Ein visuelles VR-Programmiersystem mit lokalen Constraints für die interaktive virtuelle Konstruktion

Peter Biermann

Dipl-Inform. Peter Biermann
pbierman@techfak.uni-bielefeld.de
Technische Fakultät
Universität Bielefeld

Abdruck der genehmigten Dissertation zur Erlangung des
akademischen Grades Doktor der Naturwissenschaften (Dr. rer.-nat.)

Gutachter:
Prof. Dr. Ipke Wachsmuth
Prof. Dr. Henrik Tramberend

Prüfungsausschuss:
Prof. Dr. Ipke Wachsmuth
Prof. Dr. Henrik Tramberend
Prof. Dr. Helge Ritter
Dr. Carsten Gnörlich

Tag der Promotion:
24. November 2006

Referenzexemplare gedruckt auf alterungsbeständigem
Papier nach ISO 9706

Geleitwort

Der Einbezug wissensbasierter Techniken in der Virtuellen Realität (VR) ist besonders für Anwendungen im Konstruktionsbereich zukunftsweisend, um computergraphische Entwurfsmodelle realer Objekte und deren Herstellungsprozesse bereits vor dem Bau physikalischer Produktmodelle realistisch darstellen und interaktiv verändern zu können. Mit einer solchen Zielstellung *interaktiver virtueller Konstruktion* konzipiert die vorliegende Dissertation von Peter Biermann ein speziell für Anwendungen der Virtuellen Realität vorgesehenes Programmiersystem, *VIPLIVE*. Das Programmiersystem

- orientiert sich an existierenden Metaphern zur Softwareentwicklung in echtzeitfähigen 3D-computergraphischen Umgebungen, basierend auf dem Datenflussparadigma
- bettet durch eine graphische Repräsentation der Datenflusskomponenten die Möglichkeit visueller Programmentwicklung direkt in die virtuelle Umgebung ein
- erweitert das reine Datenflussparadigma durch lokale Randbedingungen (Constraints) für die weit reichende Kopplung von Anwendungsparametern, insbesondere im Bereich der Szenengraph-orientierten rigid-body-Transformationen

Die Einbettung einer Constraintlösungsstategie in den Prozessfluss des Datenflussgraphen stellt einen entscheidenden Fortschritt gegenüber den bislang verfügbaren Ansätzen dar. Das strukturierte Design mittels deklarativer XML-basierter Beschreibungssprachen und deren Übersetzung in laufzeitkritische Komponenten erfüllt ingenieurstechnische Ansprüche; zugleich bewältigt die umfangreiche technische Umsetzung erhebliche Anforderungen an das Softwaredesign.

Ausgehend von den Forschungsarbeiten im Projekt “Virtuelle Werkstatt” an der Bielefelder Technischen Fakultät zeigt sich *VIPLIVE* praxistauglich für die Realisierung vielfältiger VR-Projekte und weist Herrn Biermann als Experten auf seinem Fachgebiet aus.

Bielefeld/Hannover, im März 2007 Ipke Wachsmuth und Henrik Tramberend

Danksagung

Die Arbeitsgruppe Wissensbasierte Systeme der technischen Fakultät, in der ich während der Entstehung dieser Arbeit Mitarbeiter war, hat mir ein angenehmes Umfeld für meine Arbeiten gegeben. Die Mitarbeiter und studentischen Hilfskräfte der Arbeitsgruppe waren stets bereit, durch ihre Unterstützung mit zum Gelingen der Arbeit beizutragen. Danke an Marc Latoschik, mit dem ich meine Ideen jederzeit diskutieren konnte, und auch an alle anderen Mitarbeiter, ohne deren Zusammenarbeit die Realisierung des hier konzipierten Systems nicht möglich gewesen wäre.

Besonderen Dank gilt Ipke Wachsmuth, der mir diese Arbeit ermöglicht hat und mich vor allem in der letzten Phase dieser Arbeit unterstützt hat. Auch Dank dafür, dass er mich von Zeit zu Zeit (zurück) auf den Pfad der Wissenschaft geführt hat. Einen Dank auch an Henrik Tramberend, dass er sich bereit erklärt hat, diese Arbeit mit zu begutachten.

Meiner Familie bin ich vor allem Dank schuldig, dass sie ohne Einschränkungen zu mir gehalten hat, auch wenn ich gegen Ende dieser Arbeit nur begrenzte Zeit für sie zur Verfügung hatte.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung und Zielsetzung

Die Unterstützung durch Computer ist unerlässlich geworden für den aktuellen Designprozess der meisten komplex aufgebauten Industriegüter. Als Vorreiter gilt hier in erster Linie die Automobilindustrie, aber auch bei der Herstellung von individuell gefertigten Produkten und auch Massengütern, laufen oft große Teile des Designprozesses bis hin zu der maschinellen Fertigung mit Hilfe von Computerprogrammen. Einer der großen Vorteile von einem computergestützten Entwurf („*Computer Aided Design*“, *CAD*) ist die Möglichkeit, mit relativ geringem Aufwand verschiedene Prototypen eines neuen oder abgeänderten Produktes erstellen zu können. Dieses *Rapid Prototyping* ermöglicht schnelle Produktzyklen und eine flexiblere Anpassung von individuellen Fertigungen schon in der Planungsphase.

Der konventionelle Einsatz von Computern bei dem Design von industriellen Produkten geschieht mit Hilfe von CAD-Programmen, welche in der Regel auf einem „normalen“ zweidimensionalen Computerbildschirm angezeigt und durch Tastatur und Maus gesteuert werden. Aber gerade für den Umgang mit komplexeren dreidimensionalen Objekten in CAD-Anwendungen hat sich diese Art der zweidimensionalen Anzeige und Steuerung, welche auch als *Desktop-Interaktion* bezeichnet wird, oftmals als zu beschränkt erwiesen. Neben der Erweiterung der Eingabegeräte im Desktop-Bereich durch intuitivere und für die dreidimensionale Eingabe ausgelegte Geräte, findet man aber auch den Einsatz von immersiver Virtueller Realität („*Virtual Reality*“, *VR*) für die Interaktion. Die dadurch aufgebauten *Virtuellen Umgebungen* („*Virtual Environments*“, *VE*) erlauben die Darstellung der Objekte im dreidimensionalen Raum und oftmals eine intuitive Eingabe über komplexere Eingabegeräte, wie z.B. den *Stylus*[1] oder Datenhandschuhe. Die Virtuellen Umgebungen kommen häufig im Bereich der Begutachtung und Anpassung der makroskopischen Struktur des Produktes vor. Für das exakte Design von kleinen und detaillierten Objekten sind die Eingabemöglichkeiten in den VEs oftmals nicht präzise genug.

Zu dem Bereich der gröberen Planung von Produkten in VR gehören auch Konstruktionsprozesse, bei denen vorgefertigte Teile angepasst und mit anderen Teilen zusammengebaut werden. Die *Virtuelle Konstruktion* stellt dabei höchste Anforderungen an das zugrunde liegende Computersystem. Die Anforderungen betreffen dabei sowohl den Bereich der immersiven Virtuellen Umgebungen (Darstellung und Eingabe-Devices), als auch den Bereich der Interaktionsmöglichkeiten und der intuitiven Bedienung der Anwendung. Während der Einsatz von VR hohe Forderungen nach Echtzeitberechnungen für eine flüssige Darstellung der virtuellen Objekte mit geringem zeitlichen Versatz zwischen Benutzereingaben und der Darstellung der virtuellen Szene stellt, wird für eine intuitive Interaktion zwischen dem Benutzer und der Anwendung ein hohes Maß von intelligentem Verhalten des Systems erwartet. Die Virtuelle Konstruktion findet sich somit im Bereich der intelligenten Virtuellen Umgebungen („*Intelligent Virtual Environments*“, *IVE*) wieder. Die Konstruktion in immersiven Virtuellen Umgebungen stellt neben den hohen Anforderungen im Bereich der Interaktion zwischen dem Menschen und dem Computer („*Mensch-*

[1] Der Stylus ist ein dreidimensionales Eingabegerät für Virtuelle Umgebungen, mit dem eine Position und Orientierung eingegeben werden kann. Zusammen mit der Möglichkeit per Knopfdruck Aktionen auszulösen, entspricht er der Maus aus der Desktop-Umgebung.

Maschine-Interaktion“, *MMI*) noch weitere, spezielle Anforderungen an das System. Hierzu gehört unter anderem die einfache Generierung der eingesetzten Bauteile und die definierte Vorgabe von Relativbewegungen verbundener Bauteile durch Einsatz von *geometrischen Constraints*.

Ein System für die virtuelle Konstruktion muss also eine Vielzahl von Bereichen der *Virtuelle Realität, Mensch-Maschine-Interaktion, wissensbasierter intelligenter Systeme,* und *Constraintlösungen* beherrschen. Für die Erstellung von immersiven Virtuellen Umgebungen gibt es aktuell eine Vielzahl von Applikationen. Aber gerade die große Anzahl von Lösungen, welche in unterschiedlichen Bereichen im Vergleich jeweils Vorteile und Nachteile bieten, erschwert die Festlegung auf ein konkretes System. Hinzu kommt, dass durch die vielen nicht standardisierten Einzellösungen der Support der eingesetzten Lösungen nicht garantiert ist, und Weiterentwicklungen zum Teil nur für neue Systeme aufgenommen werden, während ältere Tools oft nicht mehr unterstützt werden.

Für die intuitive Interaktion in der VE ist natürliche Interaktion mittels Sprache und Gestik anzustreben. Hierfür müssen die Gesten des Benutzers möglichst genau aufgenommen und erkannt werden, um dann mit der separat erkannten Sprache zu einer Anweisung verknüpft zu werden. Für die Erkennung von Gesten müssen für die vorhandenen Möglichkeiten der Aufnahme von Benutzerbewegungen und die gewünschten Interaktionsmöglichkeiten der Applikation speziell angepasste Detektoren programmiert werden, welche zum Beispiel Handstellungen oder Bewegungstrajektorien analysieren und klassifizieren. Der zusätzliche Einsatz von Spracherkennung und die Integration des sprachlichen und des gestischen Kanals zu komplexen Anweisungen an das System ermöglicht eine intuitive Benutzung des Systems während der Interaktion im virtuellen Raum. Um eine einfach zu handhabende Bedienung des Systems durch Sprache und Gestik zu ermöglichen muss das System mehr als einen einfachen Satz von Anweisungen verstehen können. Hierzu gehört neben dem flexiblen Verstehen der gesprochenen Sätze auch ein intelligenter Mechanismus zur Dereferenzierung von Objekten. Hier sollen z.B. Farben und Formen, aber auch sprachliche Umschreibungen neu entstandener Aggregate vom System erkannt und in der Anweisung des Benutzers verwendet werden können. Auch die Umsetzung der Aktionen soll nicht den direkten (unpräzisen) Eingaben des Benutzers folgen, sondern zu einer definierten Transformation der Objekte führen.

Bei Bauteilen, welche zu einem Aggregat verbunden sind, aber noch fest definierte Bewegungen erlauben, müssen zusätzlich geometrische Constraints die Transformationen der Objekte überwachen und gegebenenfalls einschränken. Eine Simulation von kinematischen Ketten ermöglicht zusätzlich die Überprüfung von Bewegungsabläufen der konstruierten Aggregate.

1.1 Probleme vorhandener Systeme und Lösungsansatz

Fast alle Tools für die Erstellung Virtueller Umgebungen sehen eine Unterstützung zur Modellierung der Geometrie in diesen Welten vor. Dadurch, dass es inzwischen aber einheitliche Datenformate zum Austausch von Geometrieinformationen zwischen verschiedenen Anwendungen und auch optimiert für den Austausch und die Darstellung im Internet gibt – z. B. die Beschreibungssprache VRML (Carey & Bell, 1997) –, wird die konkrete Modellierung der Geometrie heute in der Regel in externen Programmen durchgeführt. Dadurch sind die vormodellierten Geometrien austausch-

bar und können in unterschiedlichen Tools eingesetzt werden. Somit verlagert sich der Schwerpunkt der Frameworks für die Erstellung von VE hin zu der Platzierung und Animation von Objekten und der Interaktion mit den virtuellen Gegenständen. Für eine hierarchische Modellierung von Transformationen der Objekte hat sich als Datenformat der *Szenengraph* durchgesetzt. Die Szenengraphstruktur ermöglicht die Positionierung von Objekten relativ zueinander. Damit wirken sich Transformationen von Objekten direkt auf die Transformation der untergruppierten Szenengraphstruktur aus. Auch wenn der strenge hierarchische Aufbau der Transformationen im Szenengraphen sich für einige Anwendungen als problematisch herausgestellt hat[2], ist die Modellierung über Szenengraphen heute in fast allen Tools zur Erstellung von Virtuellen Umgebungen zu finden.

Gleichzeitig mit der Möglichkeit in virtuellen Welten interaktiv zu agieren entstand das Bedürfnis, animierte Bewegungen und Interaktionen auch komfortabel modellieren zu können. Die erste Hauptoperation ist hierbei die Veränderung der Transformationen von Objekten in der Virtuellen Umgebung. Im Szenengraph ist dieses durch die Anpassung der Transformationsmatrix der positionierenden Knoten zu erreichen. Die hierarchische Struktur des Szenengraphen stellt hierbei sicher, dass abhängige Teilobjekte mitbewegt werden. Solche Teilobjekte sind z.B. eine Schale auf einem Tisch, Räder am Auto oder sämtliche Objekte im Inneren eines fahrenden Zuges. Eine elegante Möglichkeit diese Transformationsmatrizen und andere Parameter eines Knoten im Szenengraph zu verändern ergibt sich durch den Einsatz von einem *Feldkonzept* zusammen mit einem *Eventsystem*. Dieses sieht vor, veränderbare Parameter der Knoten über Felder von außen abfragen und setzen zu können. *Feldverbindungen* erlauben hierbei eine synchrone und gerichtete Propagierung von Feldwerten. Für die Erzeugung der veränderlichen Werte wurden neue, aktive Szenengraph-Knoten erschaffen, die nicht nur der Modellierung von Positionen und den Erscheinungsbildern von Geometrien dienen, sondern solche Feldwerte gezielt erstellen und über die Zeit verändern können. Vorgefertigte Animationsknoten können z.B. Transformationsmatrizen durch eine zeitliche Interpolation zwischen vorgegebenen Stützstellen verändern und durch die Verbindung mit Transformationsknoten Objekte in der virtuellen Welt bewegen.

Der Ablauf der Weitergabe von Parameterwerten über Felder mittels Feldverbindungen baut auf den Konzepten der Datenflussprogrammierung auf. Allerdings werden von den aktuellen Tools in der Regel nur Teile dieser Konzepte übernommen, um die Feldwertpropagierung mit einem Eventsystem zu realisieren. Neben vorgefertigten Knoten für die Animation von Objekten und der Abfrage von einfachen Interaktionsevents sind oft nur Erweiterungen über Knoten möglich, welche eine Verarbeitung der Werte über Scriptsprachen bereitstellen. Eine flexible und für den Einsatz in der Echtzeitanwendung von immersiven Virtuellen Umgebungen ausreichend leistungsstarke Erweiterung der Knoten als Datenflussprogramm zur Berechnung der Feldwerte ist entweder nicht möglich, oder nur mit größerem Aufwand zu erreichen. Methoden zur direkten Steuerung des Datenflusses über die Feldverbindungen hinaus oder

[2] Ein Beispiel für Probleme mit einer vorgegebenen Szenengraphhierarchie ist die physikalische Dynamiksimulation, welche die Objekte in der virtuellen Welt jeweils für sich mit ihren globalen Transformationen betrachtet, simuliert und die Positionen der Teile auch einzeln wieder setzen muss. Die implizite Abhängigkeit der Transformationen der Objekte ist hier eher hinderlich.

der Abarbeitung der Berechnungen im Datenfluss und einer Visualisierung der Programme und Datenströme sind bisher in fast keinem VR-Tool realisiert[3].

Um eine flexible und mächtige Verarbeitung von Benutzereingaben aufzubauen, die über eine direkte Übertragung von Transformationen der Eingabegräte zu den Objekten oder über vorgefertigten Animationen hinausgeht, ist die Verschaltung eines beschränkten Satzes von Interaktionsknoten mit den Transformationen der virtuellen Objekte nicht ausreichend. Eine Realisierung komplexer Interaktionsmöglichkeiten in einer großen Programmeinheit, welche alle benötigten Daten als Eingabe erhält und als Ergebnis die resultierende Transformation über eine Feldverbindung an das betroffenen Objekt in der VE weitergibt, hat entscheidende Nachteile: Zum einem können Veränderungen im Programm nur durch die Veränderung des kompilierten (bzw. interpretierten) Codes dieses Knotens durchgeführt werden, was zur Folge hat, dass die Applikation (oder Teile der Applikation) beendet, evtl. neu kompiliert und danach wieder hochgefahren werden muss. Besonders für VR-Applikationen kann ein Neustart der Anwendung viel Zeit in Anspruch nehmen. Noch schlimmer können die Folgen bei einer fehlerhaften Programmierung sein. Hat der Fehler – z.B. in der Speicherverwaltung – einen Programmabsturz zur Folge, kann nicht nur der Neustart der Anwendung problematisch sein; auch die Suche nach dem fehlerhaften Code kann sich in komplexen imperativen Programmen als schwierig erweisen. Zum anderen sind Zwischenergebnisse der Berechnung, welche oftmals auch von anderen Teilen der Applikation benutzt werden können (und um wertvolle Rechenzeit zu sparen, dieses auch sollten), nur mit hohem Programmieraufwand aus solchen Knoten zu gewinnen. Auch das Überprüfen von Zwischenergebnissen – das unerlässlich ist, um den Ablauf komplexer Programme nachzuvollziehen und zu optimieren – ist in einem monolithischen Programm nur schwierig zu erreichen. Wenn man von der fehler- und wartungsanfälligen Möglichkeit, Teile des imperativen Codes aus den Programmen zu kopieren und in neue Programmteile einzufügen, absieht, können diese Knoten nur als Ganzes in anderen Programmen wieder verwendet werden.

Alle zuvor erwähnten Eigenschaften großer imperativ programmierter Programmknoten in VR-Tools sind in der Regel für eine Programmierumgebung untragbar, in der viele Teile der Programme prototypisch entstehen und handhabbare Möglichkeiten der Fehlersuche und Anpassung komplexer Programmteile unerlässlich sind. Die Wiederverwendung von Programmteilen von kleinsten Bausteinen bis hin zu vorgefertigten Algorithmen innerhalb eines modularen Berechnungssystems, kann die Erstellung der Programme im Rapid-Prototyping-Prozess um ein vielfaches beschleunigen.

Aus diesen Überlegungen scheint die Erweiterung der Teilkonzepte der Datenflussprogrammierung, wie sie in vielen VR-Tools in Ansätzen schon vorhanden sind, hin zu einer vollständigen Datenflussprogrammiersprache ein viel versprechender Ansatz für die Programmierung von virtuellen Welten zu sein. Die Kombination mit einer visuellen Programmierumgebung, welche die Programme und – was zum Teil noch wichtiger ist – den Ablauf und die Zwischenergebnisse während der Ausführung des Programms visualisiert, kann hierbei noch einen zusätzlichen Vorteil verschaffen.

[3] Die in Abschnitt 2.5.2 vorgestellte Programmierumgebung VEDA enthält einen ersten Ansatz für eine entsprechende Kontrolle, ist aber aufgrund von anderen Eigenschaften, die in dem Abschnitt dargestellten werden, nicht allgemein einsetzbar.

Gerade die visuellen Programmierumgebungen, welche direkt auf den Paradigmen der Datenflussprogrammierung aufsetzen, haben aktuell das höchste Potential im Bereich der visuellen Programmierung. Dieser Zusammenschluss von visueller Programmierung und Datenflussprogrammierung hat sich zum einen vorteilhaft für die Formalisierung der Semantik der visuellen Programmierung herausgestellt, zum anderen haben schon Teile der Datenflussprogrammierung als Eventsystem in viele Programmierumgebungen für die Virtuelle Realität Einzug gehalten und sich dort als sehr nützlich erwiesen.

Zusätzlich zu den Eigenschaften der Datenflussprogrammierung ist eine genaue zeitliche Einordnung von Ereignissen durch den Einsatz von Daten mit Zeitstempeln und entsprechenden Verrechnungsmethoden unerlässlich. Die exakte zeitliche Herkunft der Daten ist z.B. wichtig bei der Integration verschiedener Datenquellen oder später bei der Integration von erkannter Sprache und Gestik. Die Berechnungsmethoden sollten für einzelne Programmteile die zeitliche Exaktheit der Daten oder eine möglichst geringe Latenz der Daten für eine zeitoptimierte Darstellung in der Virtuellen Umgebung ermöglichen.

Das Weiterleiten von Werten an die Parameter der Knoten im Szenengraph nach den Regeln der Datenflussprogrammierung stellt keine komfortable Möglichkeit der Modellierung von Constraints bereit. Eine Erweiterung der Eventsystems und der Feldwertpropagierung ist notwendig, um lokale Constraints mittels der Programmknoten realisieren zu können. Ein Constraintlösungs-System, das zunächst eine lokale Lösungsstrategie verfolgt und bei Problemen, wie z.B. widersprüchlichen oder zyklischen Constraints, eine übergeordnete Lösungsstrategie einsetzt, kann eine intelligente Ablauflogik der Applikation realisieren. Besonders geometrische Constraints sind z.B. in der virtuellen Konstruktion notwendig, um das Verhalten der Bauteile geeignet definieren zu können.

Für eine natürlich motivierte Interaktion in der Virtuellen Umgebung mittels Sprache und Gestik muss die Applikation semantische Informationen über die virtuellen Objekte bereitstellen. Der Zugriff auf diese, in einer Wissensbasis repräsentierten Informationen durch im Szenegraph verankerte Strukturen kann das intelligente Verhalten der Applikation unterstützen.

Das Ziel des hier zu konzipierenden Programmiersystems ist, die genannten Nachteile vorhandener Systeme durch den Einsatz von moderner, visueller Datenflussprogrammierung und einem flexiblen Berechnungssystem aufzuheben und außerdem einen kompatiblen Mechanismus für Constraints in der virtuellen Konstruktion bereitzustellen.

1.2 Aufbau der Arbeit

Das erste Kapitel gibt eine Übersicht und nennt die Zielsetzung die Arbeit. Im zweiten Kapitel wird auf die Grundlagen und den Stand der Forschung im Bereich der visuellen Datenflussprogrammierung eingegangen. Neben der Vorstellung der Konzepte visueller Programmierung und Datenflussprogrammierung werden auch Beispiele aufgezeigt, die sich mit dieser Art der Programmierung im Bereich der 3D-Visualisierung, im Bereich von Virtuellen Umgebungen und bei der Verarbeitung von Eingabedaten befassen. Im dritten Kapitel wird ein Überblick über verschiedene Typen von Constraints gegeben, welche im Bereich der Programmierung Virtueller

Umgebungen wichtig sind, und die vorhandenen Lösungsstrategien für Constraintprobleme aufgezeigt. Das vierte Kapitel beschreibt die eigene Entwicklung eines visuellen Programmiersystems und dessen Einbindung in verschiedene VR-Tools. Kapitel 5 beschreibt das Konzept und die Realisierung von Constraints mit Hilfe einer Erweiterung der im vierten Kapitel vorgestellten Programmknoten. Besonders wird hierbei auf geometrische Constraints in Szenengraphen für die virtuelle Konstruktion eingegangen. Die Einbindung in das Szenario der virtuellen Konstruktion und der Aufbau der Strukturen für den Einsatz der visuellen Programmierung und der Constraints in der Virtuellen Umgebung werden im sechsten Kapitel aufgezeigt. Kapitel 7 enthält praktische Beispiele, in denen der Vorläufer des selbst entwickelten visuellen Programmiersystems bereits zum Einsatz kam. Sie zeigen den flexiblen und komfortablen Einsatz des Systems in externen Projekten. Ein Resümee der Arbeit wird in Kapitel 8 gezogen und es gibt dort auch einen Ausblick auf zukünftige Einsatzmöglichkeiten und Erweiterungen des Systems.

Dass der komplette Einsatz der Datenflussprogrammierung in Form einer visuellen Programmierung für die Verarbeitung und Visualisierung von Daten in einem Framework für die Erstellung Virtueller Umgebungen Vorteile hat, wird im folgenden Kapitel dargestellt. Insbesondere werden bestehende Konzepte der visuellen Programmierung für den Einsatz in einer immersiven Virtuellen Umgebung und für die Erzeugung von Programmen, mit denen ein komplexes und flexibles Interaktionsverhalten modelliert werden kann, untersucht. Die Programme sollen dabei den gesamten Bereich der Interaktionslogik implementieren, angefangen bei der Erkennung der Benutzereingaben bis hin zu den Operationen zur Veränderung der Objekte in der virtuellen Welt.

2 Visuelle Datenflussprogrammierung

Bei der Erstellung einer Programmiersprache müssen viele grundlegende Aspekte beachtet werden, welche die Art der Programmerstellung, den Einsatzzweck der erstellten Programme und den Anforderungen an die Performanz und der eingesetzten Hardware betreffen. Insbesondere gilt für visuelle Programmiersprachen ein besonderes Augenmerk der Umgebung zur Erstellung der Programme, da hier die Sprache und die Programmierumgebung sehr eng gekoppelt sind. In diesem Kapitel werden Aspekte der visuellen Programmierung und der Datenflussprogrammierung und ihrer Programmierumgebungen aufgezeigt und im Hinblick auf aktuelle Entwicklungen und Systeme auf ihren Einsatzzweck in der Programmierung von und in dreidimensionalen Virtuellen Umgebungen hin untersucht.

2.1 Visuelle Programmiersprachen

Dieser Abschnitt befasst sich mit der allgemeinen Definition visueller Programmiersprachen *(Visual Programming Languages, VPL)*. Hierbei werden die unterschiedlichen Aspekte visueller Programmierung und Vor- bzw. Nachteile zu textuellen Programmiersprachen *(Textual Programming Languages, TPL)* aufgezeigt.

2.1.1 Eigenschaften visueller Programmiersprachen

Zunächst informelle Definitionen von visuellen Programmiersprachen, wie man sie in allgemeinen Beschreibungen findet:

> „Visuelle Programmiersprachen beziehen sich auf Systeme, welche dem Benutzer erlauben, ein Programm in einer zwei- oder mehrdimensionalen Art zu spezifizieren. Textuelle Programmiersprachen werden nicht als zweidimensional angesehen, da Compiler oder Interpreter sie als langen, eindimensionalen Strom verarbeiten. […]“ – nach (Myers, 1990)

> „Die visuelle Programmierung beschäftigt sich mit der Manipulation von visueller Information oder unterstützt visuelle Interaktion oder erlaubt die Programmierung mit visuellen Ausdrücken. VPLs können weiter klassifiziert werden, bezüglich der Art und Ausmaß der visuellen Ausdrücke in symbolisch-basierte, form-basierte und diagrammatische Sprachen. Visuelle Programmierumgebungen stellen graphische oder symbolische Elemente zur Verfügung, welche vom Benutzer interaktiv manipulierbar sind bezüglich einer räumlichen Grammatik zur Konstruktion von Programmen.“ – nach (Golin *et al.*, 1990)

> „Visuelle Programmierung ist eine Programmierung in der mehr als eine Dimension für die Erzeugung von Semantik benutzt wird. Jedes potentiell signifikante mehrdimensionale Objekt oder Beziehung ist ein Token (so wie in der traditionellen, textuellen Programmierung jedes Wort ein Token ist) und der Zusammenschluss mehrer solcher Token ist ein visueller Ausdruck, wie z.B. Diagramme, Freihand-Zeichnungen, Icons usw. Falls die (semantisch bedeutsame) Syntax einer Programmiersprache visuelle Ausdrücke enthält, ist die

> Programmiersprache eine visuelle Programmiersprache (VPL)." – nach (Burnett, 1999)

Die visuelle Programmierung beschäftigt sich also mit der Programmierung über mehrdimensionale Objekte und Objektbeziehungen. Je nach Ausprägung der Sprache kann die Anordnung und Form der Objekte schon eine semantische Bedeutung tragen. Auch wenn fast alle visuellen Konstrukte auch in rein textbasierter Form repräsentiert werden – auch damit sie im Computer als Datei abgespeichert werden können –, beschreibt diese Repräsentation mehrdimensionale visuelle Konstrukte. Die Syntax der visuellen Programme gibt dabei an, welche räumlichen Beziehungen für die visuellen Ausdrücke möglich sind, welche dann z.B. anhand eines Ablaufschemas vom Computer interpretiert werden.

Zu den Zielen für den Einsatz visueller Programmierung gehört unter anderem eine andere, intuitivere Herangehensweise für die Lösung von Programmieraufgaben zu finden und die Softwareentwicklung bestimmten Zielgruppen – wie z.B. nicht professionellen Programmierern – zugänglich zu machen. In machen Bereichen kann die visuelle Programmierung auch helfen, die Zeit für die Entwicklung von Problemlösungsstrategien zu verkürzen.

Während *Native VPLs* eine inhärente visuelle Darstellung ohne eine direkte Entsprechung von textuellen Elementen haben, gibt es auch viele VPL, die als Visualisierung oder visuelles Frontend von nicht-visuellen Programmen dienen. Hier gibt es vor allem in der objektorientierten Programmierung viele Beispiele von visuellen Darstellungen z.B. von Klassenhierarchien oder Methodenabhängigkeiten. Hier zeigt sich schon das Bedürfnis, Strukturen von Programmen in einer visuell aufbereiteten Form darzustellen.

Programmierumgebungen hingegen, wie z.B. die Microsoft-Produkte *Visual Basic*, *Visual C++*, *Visual Studio (Kruglinski, 1997)* oder die *Visual Age* Produkte von IBM (IBM Inc., 1998), die eine visuelle Unterstützung z.B. bei der Erstellung einer *Graphischen Benutzeroberfläche (Graphical User Interface, GUI)* für die Erstellung textbasierter Programme bereitstellen, werden – trotz ihrer Namen – nicht zu der Klasse der visuellen Programmiersprachen gezählt.

Die visuellen Strukturen einer VPL werden – vor allem bei datenflussorientierten VPLs – häufig als Graph formalisiert. Daher werden für die formale Festlegung der *Syntax* von VPLs, entsprechend der kontextsensitiven Grammatiken für textbasierte Programmiersprachen, unter anderen *Graphgrammatiken* benutzt. Diese definieren die erlaubten Operationen bei der Modifikation der Bestandteile einer VPL (Zhang *et al.*, 2001).

Da die *Semantik* einer nativen VPL stark abhängig ist von der Struktur und den räumlichen Beziehungen der einzelnen Sprachkomponenten, ist die Definition der Semantik nicht so einfach zu formalisieren, wie in imperativen textuellen Programmiersprachen. Oft wird nur eine operationale Semantik angegeben, welche direkt auf der abstrakten Syntaxbeschreibung der Sprache aufsetzt. Bei visuellen Sprachen, die auf ein vorhandenes Programmier-Paradigma aufsetzen oder ein visuelles Frontend einer textbasierten Programmiersprache sind, kann hingegen die Semantik durch die unterliegende formale Struktur definiert werden (Erwig, 1998).

VPLs können sowohl statische (z.B. als Klassendiagramme) als auch dynamische Sachverhalte (z.B. in Ablaufdiagrammen) darstellen. Bei einigen VPLs kann die visuelle Notation komplett interaktiv aufgebaut und verändert werden. Manche Systeme können Statusveränderungen der Programme während des Ablaufs direkt in der visuellen Repräsentation anzeigen und bieten zudem die Möglichkeit, die Programme während der Ausführung zu verändern, und direkt die Auswirkungen des veränderten Ablaufs zu beobachten.

2.1.2 Kritik an Visueller Programmierung

Auch wenn eine Reihe von Aspekten für die Methoden der visuellen Programmierung sprechen, findet man – vor allem in der Aufbruchzeit – auch einige Kritik an visueller Programmierung.

In einem viel zitierten Artikel von Fred Brooks ist folgender Abschnitt über visuelle Programmierung zu lesen:

> „A favorite subject for PhD dissertations in software engineering is graphical, or visual, programming - the application of computer graphics to software design. [....] Nothing even convincing, much less exciting, has yet emerged from such efforts. I am persuaded that nothing will." – aus (Brooks, 1987)

In seinem Artikel sucht Brooks nach einer universellen Lösungsmethode (einer „Silver Bullet") für den Bereich der Softwareentwicklung und untersucht die visuelle Programmierung als einen möglichen Ansatz. Die Visuelle Programmierung kann sicherlich nicht als das Universalwerkzeug im gesamten Bereich der Softwareentwicklung eingesetzt werden. Für einige Bereiche hingegen können die Methoden der visuellen Programmierung vor allem im Zusammenhang mit der Datenflussprogrammierung (siehe Abschnitt 2.3) sehr gewinnbringend eingesetzt werden. Dieses gilt nicht nur für den Fall ungeübter Programmierer bzw. nicht direkt im der Programmierung involvierter Personen (z.B. Kunden des Softwareproduktes), welche einen viel direkteren Zugang zu den Programmabläufen bekommen, sondern auch für den professionellen Programmierer im Falle von komplexeren Abläufen.

Gerade als ordnende Ablaufkontrollsprache kommt der visuellen Programmierung eine besonders wichtige Rolle zu (Beck & Keshav, 1991). Während ein sehr kleinschrittiges visuelles Programm in der Tat schnell unübersichtlich werden kann, ist ein mehrstufiger, hybrider Ansatz – wobei z.B. eine zusätzliche textuelle Programmiersprache für die Programmierung einzelner Knoten eingesetzt wird – in der Lage den Ablauf des Programms insgesamt sehr übersichtlich darzustellen. Gerade der Ablauf von Funktionsaufrufen ist in textuellen Programmiersprachen oft unübersichtlich und kann durch die visuelle Programmierung als Ablaufkontrollsprache in machen Gebieten viel übersichtlicher gestaltet werden.

Brooks' Einwand der begrenzten Größe bzw. Auflösung der Bildschirme aus (Brooks, 1987) kann damit entkräftet werden, vor allem, wenn man zusätzlich die heutige rasante Entwicklung zu immer leistungsfähigerer und komfortabler zu programmierenden Graphik-Hardware und andererseits zu hoch auflösenden Ausgabegeräten bis hin zu immersiven, dreidimensionalen Umgebungen mit in Betracht zieht.

Ein weiterer prominenter Einwand zu der Limitierung bei der Darstellung von visuellen Primitiven kommt von Fred Lakin, der einen Ausspruch von Peter Deutsch in verkürzter Fassung als *Deutsch-Limit* bekannt gemacht hat:

> „The problem with visual programming is that you can't have more than 50 visual primitives on the screen at the same time." ("Deutsch Limit")

Hier gelten die zuvor gemachten Einwände entsprechend. So ist die Programmierung eines Betriebsystems, worauf sich der Ausspruch bezieht, im Gegensatz zu anderen Bereichen, sicherlich keine Domäne der Softwareentwicklung, wo die visuelle Programmierung ihre Stärken ausspielen kann. Das Problem der begrenzten Anzahl visueller Primitive ist aber sicherlich im Auge zu behalten. Durch den in Abschnitt 2.2 beschriebenen Ansatz der grobkörnigen Datenflussprogrammierung und einer zusätzlichen Kapselung von Subgraphen in speziellen Gruppierungsknoten, welche ihr eigenes Interface definieren können, kann dieser Problematik aber begegnet werden. Außerdem ist es auf der anderen Seite auch kaum möglich, ein etwas komplizierteres textuelles Programm, das schnell über hundert Zeilen Programmcode enthält, übersichtlich auf einer Bildschirmseite darzustellen.

Ein weiterer Einwand gegen die visuelle Programmierung ergibt sich dadurch, dass die visuellen Programmiersprachen sich gegenüber den traditionellen textbasierten Sprachen bisher nicht durchsetzen konnten. So findet man vereinzelt visuelle Programmierumgebungen, die frei oder kommerziell eingesetzt werden. Beispiele dafür sind die visuellen Programmiersysteme LabView (Labview, 2000) und ProGraph (Cox *et al.*, 1989). In der allgemeinen Anwendung findet man aber in neuerer Zeit visuelle Sprachen, mit deren Hilfe sich die Struktur und der Ablauf klassischer textueller Programme spezifizieren und modellieren lassen. Beispiele sind UML, SDL, Petrinetze, Statecharts, Datenflussdiagramme usw. Im Bereich der objektorientierten Programmierung sind hauptsächlich UML-Diagramme anzuführen, zu denen sich in der aktuellen Forschung im Bereich der visuellen Programmierung viele Abhandlungen (Dong & Yang, 2003; Kuester *et al.*, 2003; Störrle, 2004) finden lassen. Auch viele integrierte Programmierumgebungen für objektorientiertes Programmieren, wie z.B. Eclipse (Object Technology International Inc., 2001) enthalten schon Möglichkeiten zur Erstellung und Bearbeitung von UML-Diagrammen. Der begrenzte Einsatz von nativen VPLs liegt zum einem daran, dass wie zuvor erläutert diese Art der Programmierung sich nicht gut für den allgemeinen Einsatz in der Softwareentwicklung eignet. Zum anderem haben Studien gezeigt, dass für professionelle Programmierer innerhalb einiger Aufgabenbereiche die visuelle Umgebung das Programmieren verlangsamen kann (Green & Petre, 1992). Ein Hauptgrund dafür liegt vor allem in der schnelleren Erstellung von Textcode – zu einem durch das mechanisch schnellere Tippen von Text, zum anderen durch den gewohnten Umgang mit textbasierten Code – im Vergleich zu der Manipulation von visuellen Darstellungen z.B. über Menüstrukturen.

Auf der anderen Seite können VPLs in machen Einsatzgebieten überaus hilfreich sein. Dadurch, dass der Programmierer mit konkreten Objekten hantiert, welche direkt manipuliert werden können und bei den dynamischen VPLs ein sofortiges, visuelles Feedback möglich ist, haben die visuellen Programmiersprachen einen Vorteil gegenüber den rein textbasierten Sprachen (Green & Petre, 1996). So sind Programmabläufe

oft einfacher nachzuvollziehen und mit Hilfe des visuellen Debuggings Fehler im Ablauf einfacher zu finden als bei rein textbasierten Programmiersprachen.

2.1.3 Visuelle Programmierumgebungen

Ein wichtiger Faktor bei dem Einsatz einer Programmiersprache ist aber auch die Programmierumgebung mit deren Hilfe die Programme erstellt werden. Auch wenn für die Programmierung mit TPLs theoretisch ein einfacher Texteditor genügt, findet man in neuerer Zeit viele Programmiertools mit graphischer Unterstützung für TPLs (z.B.: *Eclipse* (Object Technology International Inc., 2001), *KDevelop* (KDevelop Team), *IBM VisualAge* (IBM Inc., 1998), *Microsoft VisualStudio* (Kruglinski, 1997)). Die Aufgaben dieser Programmierumgebungen sind vielfältig. In erster Linie helfen sie bei der Organisation und der initialen Erstellung von Programmdateien, Klassenstrukturen und Methoden und bei der Automatisierung des Kompilationsprozesses. Hinzu kommen Möglichkeiten, ein *graphische Benutzerinterface („Graphical User Interface", GUI)* über ein graphisches Layout zu definieren und den erzeugenden Code automatisch erstellen zu lassen. Integrierte visuelle Debugger können bei Laufzeitfehlern direkt die Problemstelle und die aktuellen Belegungen der Variablen direkt im Programmcode anzeigen. Diese graphisch unterstützten Programmierumgebungen für rein textbasierte Programmiersprachen zeigen in manchen Aspekten Ähnlichkeiten zu den visuellen Programmiersprachen, werden in der Regel aber nicht zu dem Bereich der VPL zugeordnet. Dieser Schritt wird erst z.B. durch das Hinzunehmen von visuellen Editiermöglichkeiten objektorientierter Programmkonstrukte anhand von Struktur- oder Ablauf-Diagrammen gegangen. Ein prominentes Beispiel für eine solche Diagrammsprache ist die *„Unified Modelling Language" (UML)* (Eriksson & Penker, 1998), für die in der aktuellen Forschung verschiedene Methoden der visuellen Programmierung angewendet werden (Bichler *et al.*, 2004; Dong & Yang, 2003; Ermel *et al.*, 2005; Störrle, 2004).

Bei VPLs spielt die Programmierumgebung eine noch viel größere Rolle als bei den TPLs. Eine VPL wird nicht über mit der Tastatur editierten Text, sondern über ein alternatives Interface verändert. Dem Design dieses Interfaces kommt eine besondere Bedeutung zu, da es direkt die Editier und Ausdruckmöglichkeiten der VPL bestimmt. Durch diesen direkten Zusammenhang zwischen Programmiersprache und Programmierumgebung verschwimmt die Unterscheidung zwischen den beiden. Dadurch dass außer einer einfachen Maus-basierten „Point und Klick"-Metapher keine verbreiteten Standards für die Editierung von visuellen Primitiven bestehen, wurden viele VPLs mit einer eigenen für die Sprache entwickelten Programmierumgebung ausgestattet. Da VPLs neue Anforderungen an das Benutzerinterface erschaffen, sind diese Programmierumgebungen auch ein Thema im Forschungsgebiet der *Mensch-Maschine Schnittstelle („Human Computer Interface", HCI)* geworden.

Die höchsten Anforderungen stellen dabei die VPLs mit der Möglichkeit direkten visuellen Feedbacks und der Möglichkeit, die Programme während der Ausführung zu verändern. Diese *reaktiven visuellen Programmiersprachen („Responsive VPL"* (Burnett *et al.*, 1995)) erfordern eine Umgebung, die eine gute Visualisierung der Programme und Daten und die Möglichkeit diese in Animation dazustellen bereitstellt. Das größte Potential für reaktive VPL ergibt sich im Bereich der Datenflussprogrammierung (siehe Abschnitt 2.2), wo die Datenströme zwischen den Komponenten sich direkt zur

Visualisierung in der Darstellung der visuellen Programms anbieten. Hierbei kann die Datenvisualisierung über textuelle Einblendungen in der graphischen Darstellung des Programms realisiert werden, wie z.B. in (Shizuki *et al.*, 2002), oder die Daten selber werden in eine graphische Repräsentation überführt. Ein skalarer Zahlenwert kann z.B. als Balken oder für den zeitlichen Verlauf als Funktionsplot dargestellt werden. Je nach Komplexität des Datentyps kann eine gute Visualisierung sehr viel schneller interpretiert werden als eine textuelle Darstellung. Reaktive VPLs ermöglichen eine sehr direkte Art des Überprüfens und der Fehlersuche in visuellen Programmen. In Anlehnung an die Metapher bei Textverarbeitungsprogrammen das der dargestellte Text während der Editierung auch gleich dem Endergebnis entspricht *(„What you see is What you get", WYSIWYG)* wird diese Art des Testens von Programmen auch z.B. in (Rothermel *et al.*, 1998; Burnett *et al.*, 2002) als *„What you see is what you test" (WYSIWYT)* Umgebung beschrieben. Da bei der Überprüfung und Fehlersuche die Zwischenergebnisse während des Programmablaufes sichtbar gemacht werden können und der Einfluss von Änderungen am Programm direkt sichtbar wird, kann das Finden von Fehlern und Optimieren von Programmen vor allem bei dem Umgang mit komplexen Abläufen sehr komfortabel gestaltet werden (Okamura *et al.*, 2004; Hundhausen & Brown, 2005). Gerade bei der Fehlersuche in Programmabläufen kann eine interaktive VPL viel mehr Unterstützung geben als textbasierte Programme (Wilcox *et al.*, 1997).

Auf der Suche nach Programmierumgebungen für reaktive VPLs mit vielfältigen Möglichkeiten für Visualisierung, Animation und Interaktion bieten sich in neuerer Zeit auch *Interaktive Virtuelle Umgebungen („Interactive Virtual Environment", IVE)* an. Ansätze zu der Integration von DFVPLs und Virtuellen Umgebungen findet man in (Sherman, 1993). In diesen Umgebungen können sowohl visuelle Programme als auch Daten dreidimensional dargestellt, exploriert und interaktiv verändert werden. Allerdings findet sich bis jetzt nur ein erster Ansatz einer reaktiven visuellen Programmierumgebung in IVEs (Steed & Slater, 1996), obwohl Datenflusskonstrukte schon länger in VR-Programmierumgebungen zu finden sind (siehe Abschnitt 2.5.1).

Die aktuelle Entwicklung der nativen VPLs konzentriert sich vor allem auf den Einsatz von visuellen Umgebungen für die Datenflussprogrammierung. Diese Kombination von visueller Programmierung und Datenflussprogrammierung hat viele Vorteile. Zum einen lässt sich durch das Konzept der Datenflussprogrammierung eine formale Syntax und Semantik der visuellen Programme ableiten, zum anderen bietet sich für die Formalisierung von Datenflussprogrammen durch gerichtete Graphen eine visuelle Repräsentation sehr gut an.

Es zeigt sich auch, dass die Kombination von visueller Programmierung und Datenflussprogrammierung hohes Potenzial für den Programmierer, aber vor allem auch, wie in (Baroth *et al.*, 1995) gezeigt, für der Kommunikation zwischen Programmierer und Endanwender hat. Zum Teil verwischen auch die Grenzen zwischen dem Programmierer und dem Anwender der Software, da die visuelle Programmierung einem Endanwender ohne sonstige Programmierkenntnisse das Erstellen eigener oder das Verändern vorhandener Programme ermöglicht (Burnett *et al.*, 2004).

Ein weiterer Vorteil der Datenflussprogrammierung ist die mögliche automatische Parallelisierung von Programmen, was für Entwicklung von parallelen Prozessen wichtig ist (Jagannathan, 1995a).

2.2 Datenflussprogrammierung

Dieser Abschnitt gibt einen Überblick über die Geschichte und den aktuellen Stand der *Datenfluss-Programmiersprachen (Dataflow Programming Languages, DFPL)*. In (Johnston *et al.*, 2004) sind weitere Details zu dem folgenden Thema zu finden. Aufgezeigt werden hier vor allem die Grundlagen und aktuelle Entwicklungen in den Methoden der Datenflussprogrammierung.

Die Ursprünge der Datenflussprogrammierung liegen in den 70er Jahren, wobei die ursprüngliche Motivation von Datenflussprogrammierung in der Parallelisierung von Programmabläufen lag. Hauptsächlich durch die Kritik an bestehenden Von-Neumann-Architekturen, wobei als Hauptprobleme der global Programmzähler und der global zu verändernde Speicher gesehen wurde (Ackerman, 1982; Backus, 1978). Als Alternative wurde z.B. in (Dennis, 1974; Weng, 1975) die Datenfluss-Architektur vorgeschlagen. Diese Architektur benutzt im Gegensatz zu der Von-Neumann-Architektur lokalen Speicher für die Variablen und führt Operationen aus, sobald gewisse Eingabeparameter (siehe folgender Abschnitt) gegeben sind. Da die herkömmlichen imperativen Programmiersprachen für diese neue Architektur stellenweise Probleme z.B. bezüglich Seiteneffekte und Lokalität aufwarfen, wurden teilweise durch Einschränkungen von Aspekten der vorhandenen Programmiersprachen neue DFPL entwickelt – z.B. in (Ackerman, 1982; Dennis, 1974). Während die ersten Ansätze sich wegen schlechter Performanz bedingt durch eine zu feine Granularität paralleler Instruktionen nicht durchsetzen konnten, konnte dieser Mangel durch hybride von Neumann Datenfluss-Architekturen mit einer grobkörnigeren Struktur der Parallelisierung aufgefangen werden. Hier wurden Instruktionen zu Einheiten zusammengefasst, wobei diese weiterhin nach den Regel des Datenfluss-Ablaufmodells (siehe Abschnitt 2.2.1) ausgeführt wurden und somit die Vorteile – wie z.B. die Möglichkeit der automatischen Parallelisierung – des Ansatzes erhalten blieben. Dementsprechend wurde die Datenflussprogrammierung als ein Spezialgebiet der Programmierung von parallelen Prozessen angesehen, welche zum Teil auch auf speziell für Datenflussprogramme ausgelegte Hardware – z.B. von (Dennis & Misunas, 1975; Barahona & Gurd, 1985; Gurd & Bohm, 1987; Papadopoulos, 1988) – entwickelt wurde.

2.2.1 Das Datenfluss-Ablaufmodell

Im *Datenfluss-Ablaufmodell (dataflow execution model)* werden DFPL als gerichteter azyklischer Graph *(Directed Acyclic Graph, DAG)* repräsentiert (siehe Abbildung 1 rechts). Hierbei entsprechen die Knoten des Graphen primitiven (z.B. arithmetischen oder vergleichenden) Instruktionen und die Verbindungen im Graphen repräsentieren die Abhängigkeit der Daten zwischen diesen Instruktionen. Die Datentoken fließen entlang der Pfeile, wobei Daten nicht verändert werden, sondern entsprechend einer funktionalen Abstraktion an der Ausgabe der Knoten neue Datentoken erzeugt werden. Abbildung 1 zeigt eine Gegenüberstellung einer textbasierten und einer Datenfluss-Darstellung. Der Datenfluss entspringt speziellen Knoten, welche als Datenquellen, Trigger oder Sensoren bezeichnet werden. Sie bilden das externe Interface des Systems. Bei dem datengetrieben Ablaufmodell wird die Berechnung der jeweiligen Knoten durch den Fluss der Daten angestoßen. Sobald Datentoken an einem Set von Eingabeverbindungen (*firing-set*) eines Knotens anliegen, wird der

Knoten zur Ausführung bereit. Die Ausführung der dem Knoten entsprechenden Instruktion erfolgt dann nach einer nicht bestimmten Zeit und verbraucht dabei jeweils ein Datentoken aus jeder Eingabeverbindung des firing-sets und produziert ein Datentoken auf einem Teil oder allen seiner Ausgabeverbindungen.

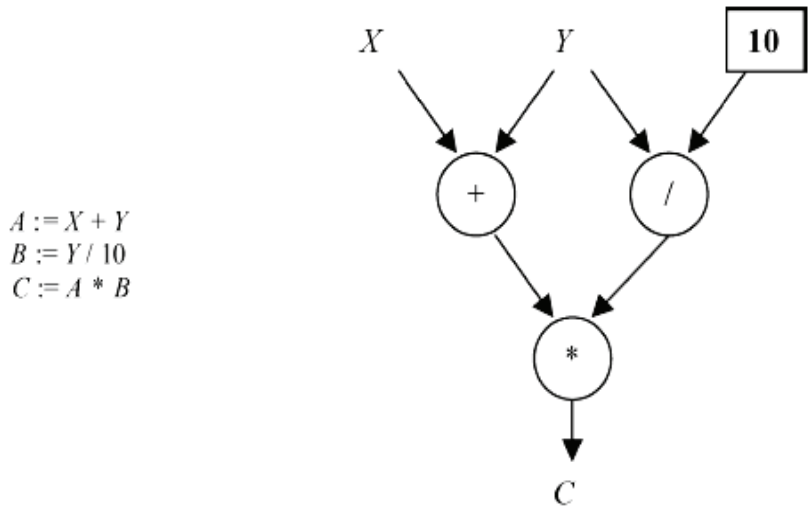


Abbildung 1:Ein einfaches sequenzielles Programm und seine Darstellung als Datenflussgraph. – aus (Johnston et al., 2004)

Da die Instruktionen zur Ausführung angemeldet werden sobald ihre Eingabeparameter zur Verfügung stehen, steht dieses Ablaufmodell im Gegensatz zu einer Von-Neumann-Architektur, wo die Instruktionen fest nach dem Programmzähler ausgeführt werden. Der Vorteil im Bereich der parallelen Programmierung ist, dass in diesem Ablaufmodell mehrere Knoten, deren Eingabeverbindungen bereit sind, gleichzeitig verarbeitet werden können, was den weit reichenden Einsatz paralleler Verarbeitung ermöglicht. Bei dem Datenflussgraphen auf der rechten Seite von Abbildung 1 werden, wenn die Eingabe an Y bereit ist, die zwei mit der Eingabe verbundenen Knoten (+) und (/) gleichzeitig zu Ausführung angemeldet und können dann parallel abgearbeitet werden. Bei der sequentiellen Schreibweise auf der linken Seite werden die Instruktionen immer der Reihe nach abgearbeitet. Im Falle des *statischen Datenflussmodells* transportiert jede Verbindung immer nur ein einzelnes Datentoken. Es ergibt sich aber die Möglichkeit, bei mehreren aufeinander folgenden Eingabetoken die Berechnung des folgenden Datensatzes schon anzufangen, sobald die erste Schicht für den aktuellen Datensatz (im Beispiel die Knoten (+) und (/)) abgearbeitet wurde. Man spricht in diesem Fall von *„Pipelined Dataflow“* (Mencer *et al.*, 2000). Wenn die Knoten unabhängig voneinander Datentoken produzieren können, müssen die Verbindungen zwischen diesen Knoten Daten in der Form von FIFO-Buffern speichern können. Bei diesem *dynamischen Datenflussmodell* wird bei Ausführung der Knoten jeweils ein Token aus der Pipeline der Eingabeverbindungen gelesen und ein Token in die Pipeline der Ausgabeverbindungen geschrieben.

Das in Abbildung 1 benutzte Modell für die Datenverbindungen entspricht dem in (Verdoscia & Vaccaro, 1998) beschriebenen direkten Verbindungen. Eine Verbindungsart in diesem Modell sind replizierende Verbindungen (*Replica links,* siehe Abbildung 2 rechts). Ein Replica-Link ist ein impliziter Knoten mit genau einer

Eingabekante und mehrere Ausgabekanten, wobei jedes ankommende Datentoken der Eingabekante auf allen Ausgabekanten repliziert wird. In Abbildung 1 wird dieses bei dem Eingabetoken Y benutzt, welches für die beiden Rechenoperationen (+) und (/) repliziert wird. Das entsprechende Gegenstück für das Zusammenfassen mehrerer Eingaben zu einer Ausgabe ist eine zusammenfassende Verbindung (*Joint link,* siehe Abbildung 2 links). Ein Joint Link hat dementsprechend mehrere Eingabekanten und eine Ausgabekante, wobei das erste Datentoken, das auf einer der Eingabekanten ankommt, auf die Ausgabekante platziert wird. Da das Verhalten von Joint links bei mehreren gleichzeitigen Datentoken an den Eingabekanälen nicht vorhersagbar ist, können durch diese Art der Verbindung undeterministische Datenflussgraphen entstehen.

Abbildung 2: Vereinigende (a) und replizierende (b) Verbindungen (Joint / Replica links) – aus (Verdoscia & Vaccaro, 1998)

Um ein deterministisches Verhalten der Datenflussprogramme sicherzustellen und gleichzeitig eine Kontrolle über den Datenfluss zu erlangen, werden deterministische Kontrollknoten, wie sie in Abbildung 3 dargestellt sind, definiert.

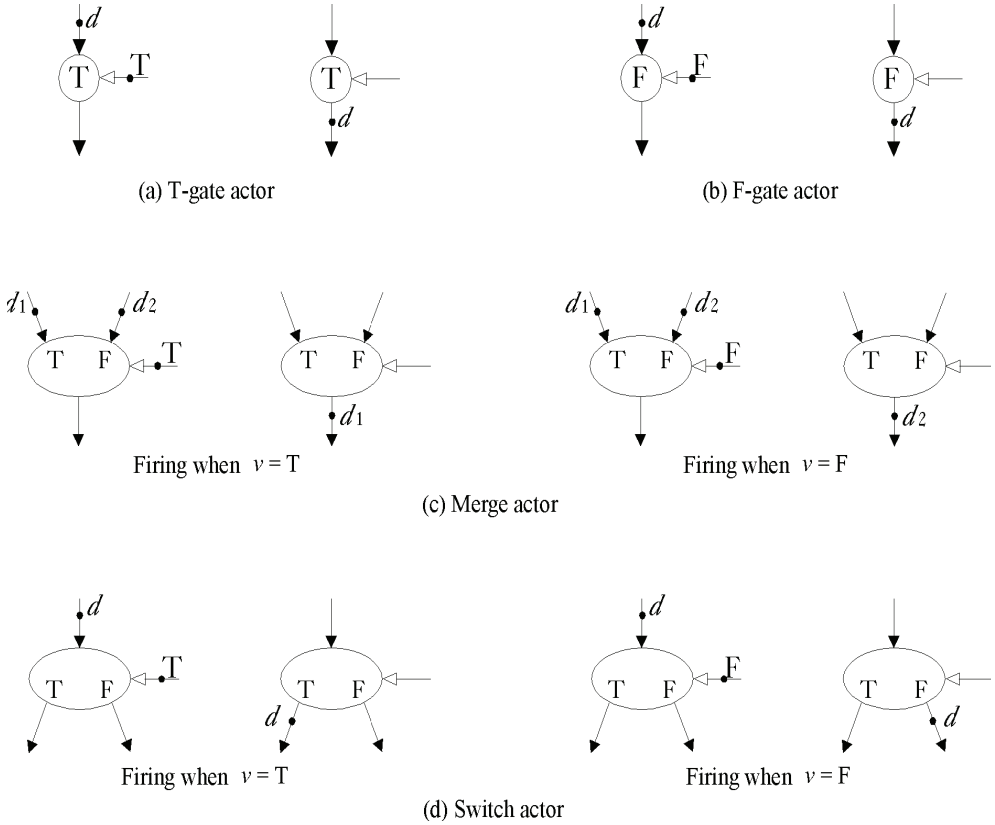
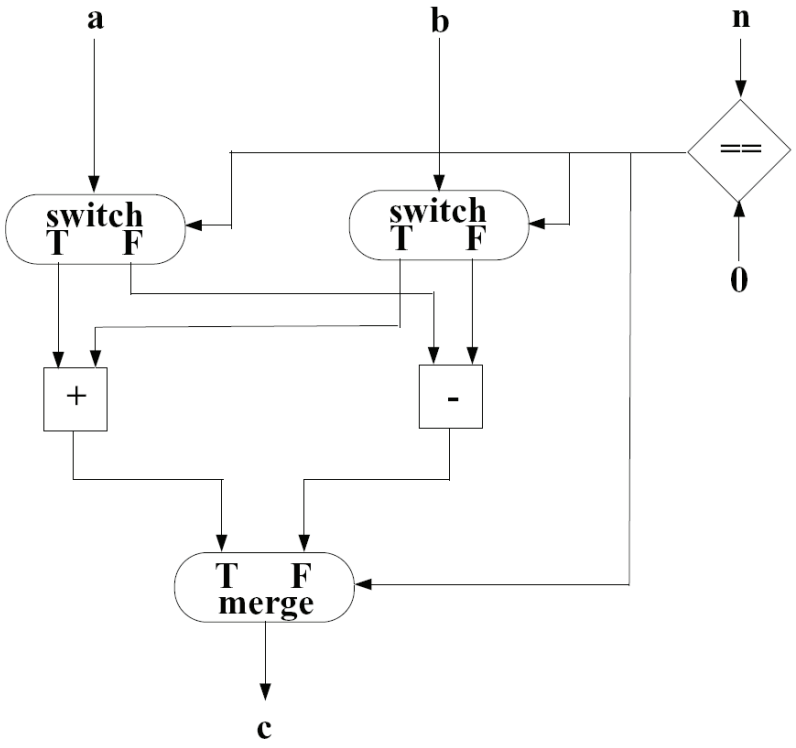


Abbildung 3: Deterministische Datenfluss-Kontrollknoten – aus (Verdoscia & Vaccaro, 1998)

Sie erlauben eine gezielte Unterbrechung oder Umleitung des Datenflusses. Je nach Ausprägung haben sie ein oder zwei Eingabe- bzw. Ausgabekanten und eine Steuer-

leitung, welche boolesche Werte transportiert. Hierbei verbrauchen nach dem klassischen Datenflussmodell die Knoten ein Datentoken von allen Eingabeleitungen und der Steuerleitung. Bei den *Gate*-Knoten entscheidet der Wert an der Steuerleitung ob das Datentoken am Eingang auf den Ausgang weitergeleitet wird oder nicht. Der *Merge*-Knoten wählt je nach Steuerwert zwischen zwei Datentoken an seinen beiden Eingängen aus und produziert das Token von dem entsprechenden Eingang auf seinem Ausgang. Ein Datentoken auf dem nicht selektierten Eingang wird dabei auch verbraucht und gelöscht. Die Merge-Knoten bilden das deterministische Gegenstück zu den Joint links, da durch das Abarbeiten der Datentoken auf allen Eingabekanten ein eindeutiger Ausgabewert sichergestellt ist. Um den Datenstrom gezielt auf zwei Ausgänge zu verteilen werden zusätzlich *Switch*-Knoten definiert. Sie verbrauchen auch ein Datentoken auf der Eingabe- und der Steuerleitung und produzieren je nach Wert auf der Steuerleitung ein Token auf der entsprechenden Ausgabeleitung.

Durch Merge-Knoten und Switch-Knoten lassen sich Kontrollstrukturen wie z.B. Schleifen oder bedingte Ausführungen in Datenflussprogrammen realisieren.



c = if n==0 then a+b else a-b

Abbildung 4 Datenflussgraph mit Steuerknoten und das entsprechende textuelle Programm– aus (Jagannathan, 1995b)

Abbildung 4 zeigt ein Datenflussnetz mit zwei Switch-Knoten, welche den Eingabestrom für zwei Rechenoperationen (+/-) bestimmen und einen Merge-Knoten, der das Ergebnis eines dieser beiden Operationen auswählt. Der Wert an der Steuerleitung wird über den Vergleich (n == 0) bestimmt. Das deterministische Verhalten der Switch- und Merge-Knoten stellt dabei sicher, dass das Ergebnis (die Datentoken am Ausgang) unabhängig von der Reihenfolge ist mit der die Datentoken an den drei Eingängen angelegt werden. Die beiden Switch-Knoten entscheiden hierbei, ob die

Eingaben **a** und **b** dem Knoten der Plus- oder dem der Minus-Operation zugeführt werden. Der Merge-Knoten wählt dann das korrekte Ergebnis aus.

2.2.2 Granularität von Datenflussprogrammen

Das Beispiel aus Abbildung 1 zeigt ein klassisches feingranulares Datenflussprogramm in dem die einzelnen Knoten atomare Instruktionen darstellen. In diesem Fall wird jede Anweisung eines einzelnen Knotens als eigener *Prozessstrang („Thread")* ausgeführt. Im Falle von sequentiellen Abschnitten im Datenflussprogramm, wo jeder Knoten zur Berechnung auf das Ergebnis der vorherigen angewiesen ist, kann die Möglichkeit der parallelen Ausführung nicht ausgenutzt werden. Hingegen kann eine übermäßige Anzahl von Threads die Performanz der Programme verschlechtern. Abbildung 5 zeigt den Zusammenhang von Granularität (d.h. Anzahl der einzelnen Threads) und Performanz. Diese Graphik zeigt eine vereinfachte Sicht, basierend auf Daten von (Sterling *et al.*, 1995). Hierbei zeigt sich, dass das Optimum der Performance in der Mitte zwischen den fein-körnigen Strukturen und den sehr grobkörnigen Varianten liegt.

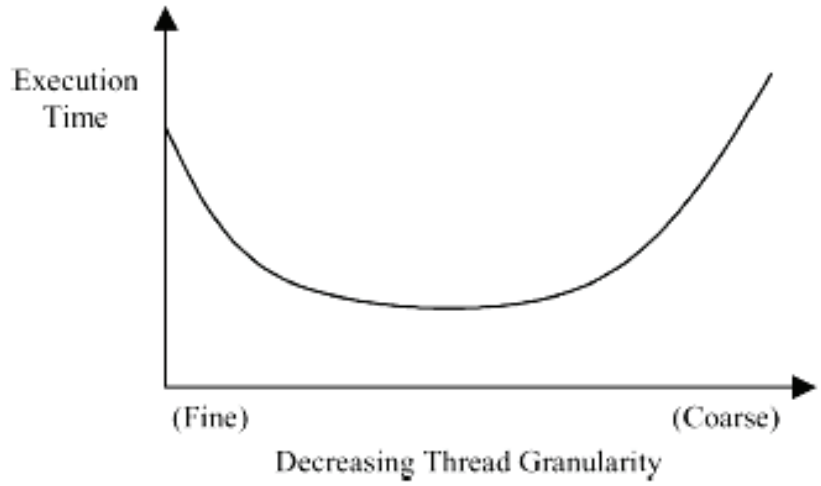


Abbildung 5: Zusammenhang von Granularität des Datenflussprogramms und Performanz – aus (Johnston et al., 2004)

Bei dem Streben hin zu gröberen Strukturen in der Datenflussprogrammierung werden zwei unterschiedliche Strategien verfolgt. In der ersten Strategie des *threadbasierten Datenflusses („Threaded Dataflow")* werden Teilgraphen des Programms, welche z.B. aufgrund vieler sequentieller Anweisungen nur wenig Parallelität erlauben, zusammengefasst und zusammen evaluiert, wenn der erste Knoten bereit ist. Hierbei werden die Knoten weiterhin als fein-körnige Struktur evaluiert, der aufwendige Eventprozess, der die Daten zwischen den einzelnen Knoten einer Gruppe weiterreicht, wird aber übergangen.

Die zweite Strategie der *grobkörnigen Datenflussprogrammierung* (*„Large-Grain Dataflow"* (Silc *et al.*, 1998)) verfolgt eine Vergröberung der Struktur durch die Zusammenfassung von Programmteilen in Sequentielle Prozesse. Hierbei werden einfache lineare Ablaufe einer feinkörnigen Struktur zusammengefasst oder direkt in einer externen klassischen befehlsorientierten Sprache implementiert. Diese Teilprogramme – auch *Macroactors* genannt – werden als Knoten im Sinne der Datenflussprogram-

mierung benutzt. Sie können als Funktionen angesehen werden, wobei die Datenflusssprache die Abhängigkeit der Funktionsaufrufe beschreibt. Da die formalen Eigenschaften des Datenflussgraphen unabhängig von der Granularität der Knoten erhalten bleiben, kann das Datenfluss-Ablaufmodell ohne Einschränkungen auf den Fall der grobkörnigen Datenflussprogrammierung übertragen werden.

Im Falle einer grobkörnigen DFPL, in der die einzelnen Knoten über eine externe sequentielle Sprache definiert werden und ganze Funktionen enthalten, kann das Datenflusskonzept als eine *Koordinierungssprache* angesehen werden. Diese Sicht unterteilt die Programmentwicklung in zwei Teile: Zum einem die Spezifikation der Berechnungen und zum anderen die Koordination des Programmablaufes. Hierbei übernimmt die externe Sprache die Definition der Berechnungen. Der in imperativen Programmiersprachen meist unübersichtliche Ablauf der Funktionsaufrufe wird durch die DFPL angegeben. Beispiele für solche Koordinierungssprachen sind VIPERS (Bernini & Mosconi, 1994) und GranularLucid (Wadge & Ashcroft, 1985).

Da eine speziell für den Threaded-Dataflow entwickelte Hardware sich gegenüber der – auf parallele Prozessoren erweiterten – Von-Neumann-Architektur nicht durchsetzen konnte, konzentrierten sich die Fortschritte in den letzen zehn Jahren auf den Bereich der grobkörnigen Datenflussprogrammierung. Der Hauptteil der Weiterentwicklung von Datenfluss-Programmiersprachen wurde in letzter Zeit aber im Berech der visuellen Datenflussprogrammierung gemacht.

2.3 Visuelle Datenflussprogrammierung

In diesem Abschnitt wird die Kombination der Methoden der Datenflussprogrammierung mit der visuellen Programmierung beschrieben.

Am Anfang der 80er Jahre findet man schon Hinweise darauf, dass die Datenflussprogrammierung nicht nur bei der Parallelisierung von Prozessen sondern durch einen visuellen Ansatz auch in anderen Bereichen des Software-Engineering hilfreich sein kann (A. L. Davis & Keller, 1982). In den 90er Jahren, als die Graphikleistung der Rechner anstieg, wurde eine Verbindung von Datenflussprogrammierung und visueller Programmierung vorangetrieben. Mit der Entwicklung von zahlreichen *visuellen Datenfluss-Programmiersprachen („DataFlow Visual Programming Language", DFVPL)* verschob sich der Fokus der Datenflussprogrammierung von der Parallelisierung von Programmabläufen hin zu dem Bereich des Software-Engineering.

Für den Bereich der visuellen Programmierung ist ein Zusammenschluss mit der Datenflussprogrammierung in vielen Bereichen sinnvoll. So bildet die Darstellung von Datenflussprogrammen über Graphen eine hervorragende Grundlage für den Aufbau einer visuellen Programmiersprache. Auch viele Probleme mit der Formalisierung visueller Programmkonstrukte können durch die Datenflussprogrammierung gelöst werden. Gerade bei der Definition der Syntax und vor allem der Semantik von visuellen Programmen, welche oft nur informell angegeben werden, bietet das Datenfluss-Ablaufmodell sehr gute formale Grundlagen (siehe Abschnitt 2.2.1).

Die schon vorhandene Visualisierung von Datenflussprogrammen als Datenflussgraph wird erweitert auf eine Interaktive Programmierumgebung, in der Datenflussgraphen editierbar sind und somit die Datenflussprogramme direkt in der visuellen Repräsentation erstellt und verändert werden können. Hierbei werden zu der einfachen Visualisierung der Knoten im Datenflussgraph (siehe Abbildung 1) zusätzlich

ihre möglichen Eingabe- und Ausgabeparameter angezeigt, um die Verbindungen für die Datentoken während der visuellen Programmierung zwischen diesen etablieren zu können. Die Parameter und Resultate der Programmknoten können dabei z.B. als eingehendes Verbindungsstück – wie z.B. in LabView (siehe Abbildung 8 im folgenden Abschnitt) – oder als Verbindungsstelle am Rand der Knoten – wie z.B. in Prograph (siehe Abbildung 9) – dargestellt werden.

Zusätzliche visuelle Konstrukte ermöglichen auch die Zusammenfassung von Programmteilen zu größeren Programmeinheiten, oder die Darstellung von alternativen Programmpfaden für bedingte Anweisungen. Der folgende Abschnitt zeigt einige Beispiele visueller Programmiersprachen, wovon viele auch auf der Grundlage von Datenflusssprachen entwickelt worden sind.

2.4 Vorhandene visuelle Programmiersprachen

Die ersten visuellen Programmiersprachen wurden zwischen Anfang der 80er und den frühen 90er Jahren entwickelt. In (Daniel, 1991) findet man eine Übersicht über diese visuellen Programmierumgebungen. Diese waren zu der Zeit aufgrund der möglichen Grafikdarstellung der Rechner weitgehend in Form von zweidimensionalen Diagrammen visualisiert.

2.4.1 Zweidimensionale visuelle Programmiersprachen

Eines der frühen visuellen Programmiersysteme war Show and Tell (Kimura & McLain, 1986). Abbildung 6 zeigt ein visuelles Programm dieser Sprache für die rekursive Berechnung der Fakultätsfunktion.

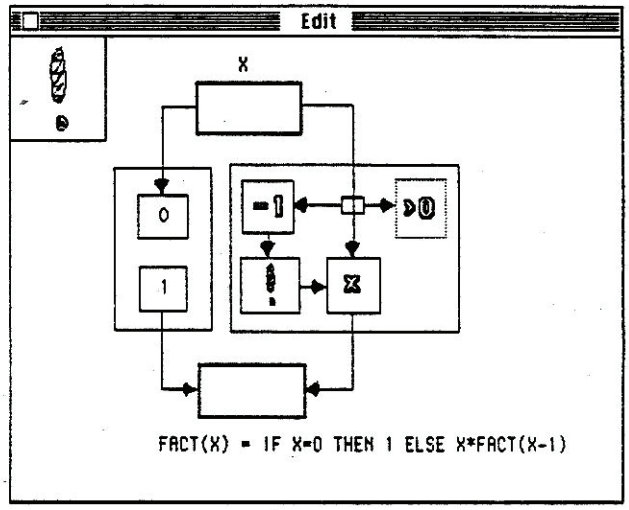


Abbildung 6: Visuelles Programm in Show and Tell – aus (Kimura & McLain, 1986)

Darstellung und Beschriftung der einzelnen Programmbausteine ist aufgrund der Einschränkungen durch die Graphikhardware zu der Zeit sehr einfach gehalten. Auch sind Teile des Programms, wie z.B. der Ablauf der bedingten Anweisung, anhand der graphischen Darstellung, teilweise nur schwierig nachzuvollziehen.

Weitere bekannte Beispiele visueller Programmiersprachen, welche zum Teil auch kommerziell eingesetzt wurden, sind LabView (Vose & Williams, 1986) und Prograph

(Cox et al., 1989). Abbildung 7 zeigt ein textuelles Programm in Basic geschrieben, das in einer Schleife iterativ die Position einer Rakete anhand der Masse der Rakete, ihres Treibstoffs und den Kräften, die auf die Rakete wirken, berechnet.

```
Fuel = 50
Mass = 10000
Force = 400000
Gravity = 32
WHILE Vdist >= 0
  IF Tim = 11 THEN Angle = .3941
  IF Tim > 100 THEN Force = 0 ELSE Mass = Mass – Fuel
  Vaccel = Force*COS(Angle)/Mass – Gravity
  Vveloc = Vveloc + Vaccel
  Vdist = Vdist + Vveloc
  Haccel = Force*SIN(Angle)/Mass
  Hveloc = Hveloc + Haccel
  Hdist = Hdist + Hveloc
  PRINT Tim, Vdist, Hdist
  Tim = Tim + 1
WEND
STOP
```

Abbildung 7: Textuelles Beispiel: Raketen-Programm in Basic – aus (Green & Petre, 1996)

Dieses Programm wurde in (Green & Petre, 1996) benutzt, um die Handhabung von textuellen und visuellen Programmen zu vergleichen. Die beiden folgenden Beispiele zeigen dasselbe Raketenprogramm in zwei verschiedenen visuellen Sprachen.

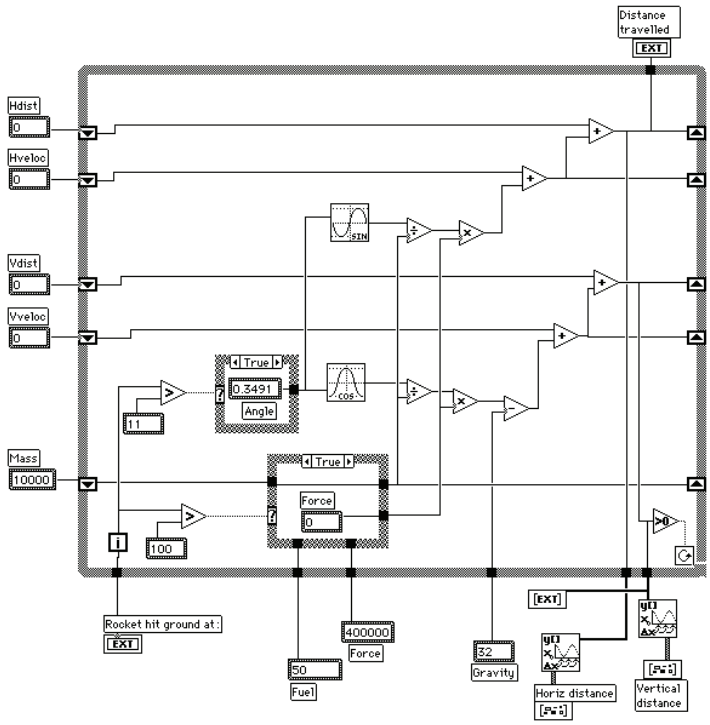


Abbildung 8: Das Raketen-Programm aus Abbildung 7 als visuelles Programm in LabView – aus (Green & Petre, 1996)

Abbildung 8 zeigt das Beispiel in der visuellen Programmiersprache LabView. Hier sind die berechneten Funktionen als Kästchen dargestellt und die Weitergabe der

Variablen als Verbindungen zwischen diesen. Im Falle von bedingten Anweisungen werden in dieser Sprache Kästchen mit zwei unterschiedlichen Inhalten für den jeweiligen Fall angezeigt. In der Abbildung sind nur die Anweisungen dafür angegeben, dass die Bedingung wahr ist. In der Programmierumgebung kann interaktiv zwischen den beiden Anweisungen für wahr und falsch umgeschaltet werden.

Abbildung 9 zeigt das gleiche Programm aus den obigen Beispielen noch einmal. Diesmal ist es in der visuellen Programmiersprache von Prograph realisiert. Im Gegensatz zu der Programmierung in LabView sind hier einzelne Programmteile geschachtelt dargestellt. Ausgehend vom Main-Programm, wo die Variablenwerte gesetzt werden, wird ein „compute“-Programm aufgerufen, das in Abbildung 9 rechts zu sehen ist. Die Funktionen aus dem „compute“-Programm sind als kleinere Programme in der Abbildung unten aufgeführt. Die ersten drei Programmteile sind jeweils in zwei Versionen vorhanden, wobei der erste Teil jeweils einer bedingten Anweisung und der zweite Teil dem alternativen Pfad entspricht.

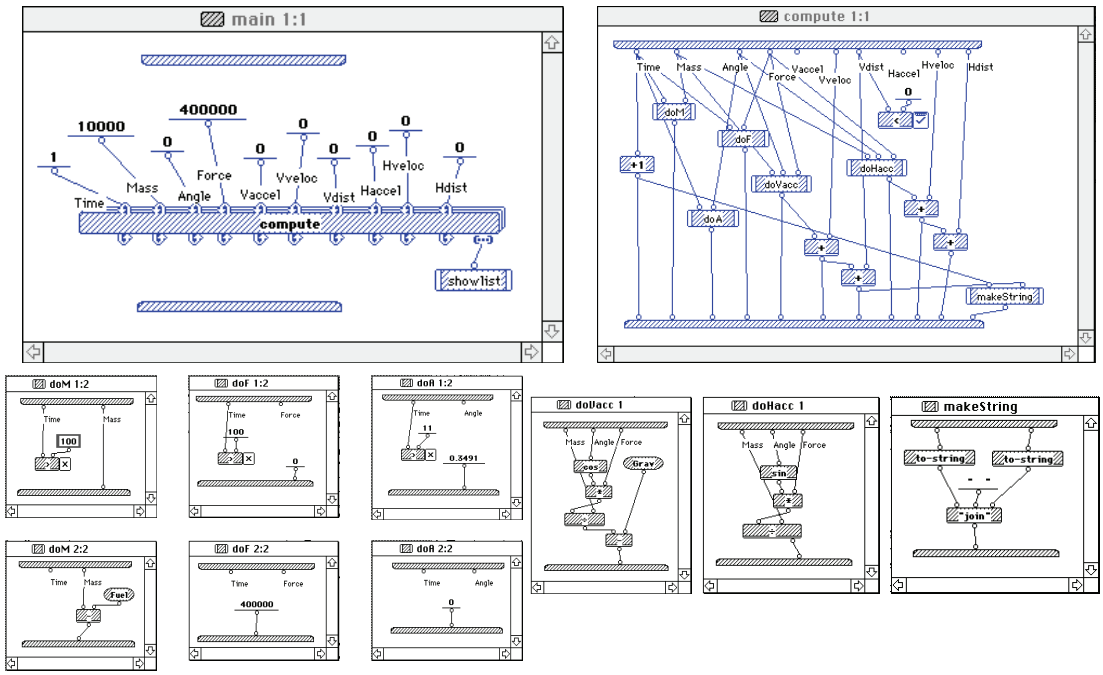


Abbildung 9: Visuelles Raketen-Programm in Prograph – aus (Green & Petre, 1996)

Eine visuelle Programmiersprache, die der Platzierung der Programmbestandteile eine große Bedeutung zukommen lässt, ist Pictorial Janus (Kahn *et al.*, 1991). In dieser Sprache werden verschiedene Verbindungsbeziehungen und räumliche Anordnungen als semantische Konstrukte in der visuellen Darstellung benutzt. Um einen Eindruck von dem Aufbau der Programme in Pictorial Janus zu geben, zeigt Abbildung 10 ein Beispiel aus dieser Programmiersprache.

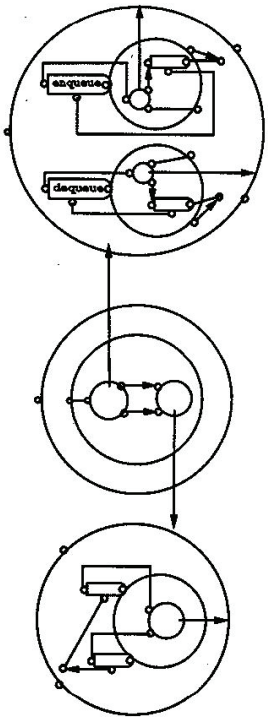


Abbildung 10: Visuelles Programm in Pictorial Janus – aus (M. A. Najork, 1996)

Abbildung 11 zeigt ein einfaches Beispiel der VPL *VIPERS* (Bernini & Mosconi, 1994), das ein Bild aus einer Datei einliest, dieses über eine Threshold-Funktion binarisiert und dann durch ein Bildbetrachtungsprogramm anzeigt.

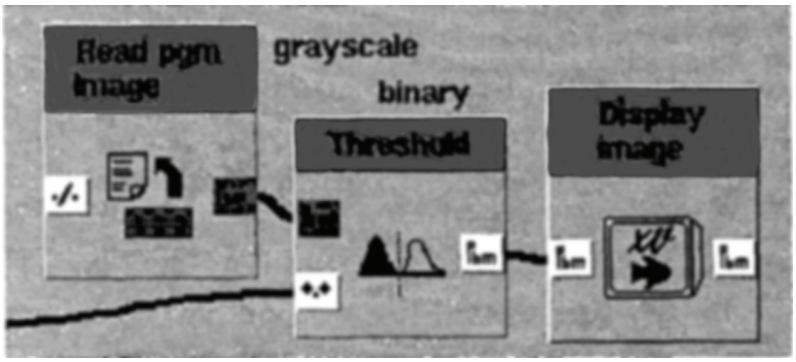


Abbildung 11:Visuelles Programm in VIPERS – aus (Bernini & Mosconi, 1994)

VIPERS ist eine modernere visuelle Programmiersprache, in der die einzelnen Programmknoten komplexere imperative Programme enthalten oder sogar externe Applikationen aufrufen können, wie in diesem Beispiel das Bildbetrachtungsprogramm „xv“, das für die Darstellung des berechneten Bildes aufgerufen wird. Als interne imperative Programmiersprache wird die Scriptingsprache Tcl/Tk verwendet.

Eine große allgemeine Verbreitung hat aktuell die visuelle Programmierumgebung „*Quartz Composer*“ von Apple (Apple Inc., 2006) da es als Entwicklungstool zu dem Betriebsystem „Mac OS X v10.4“ direkt mitgeliefert wird. Quartz Composer erweist sich als eine komfortable visuelle Programmierumgebung zur Bearbeitung und Darstellung von graphisch basierten Daten.

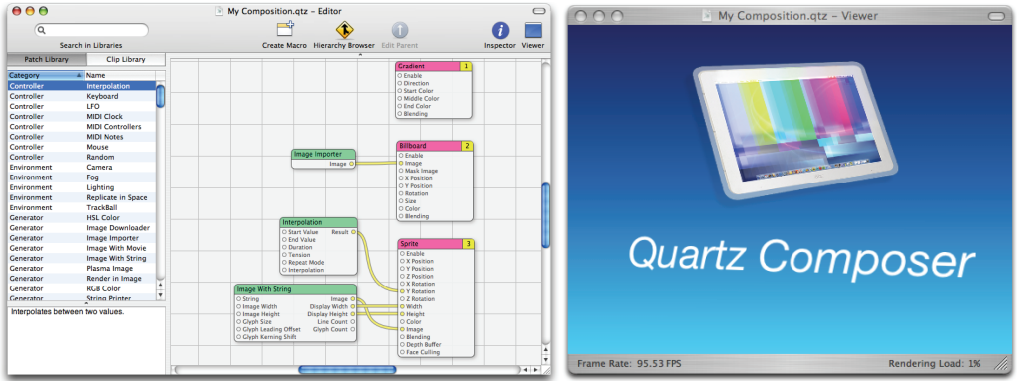


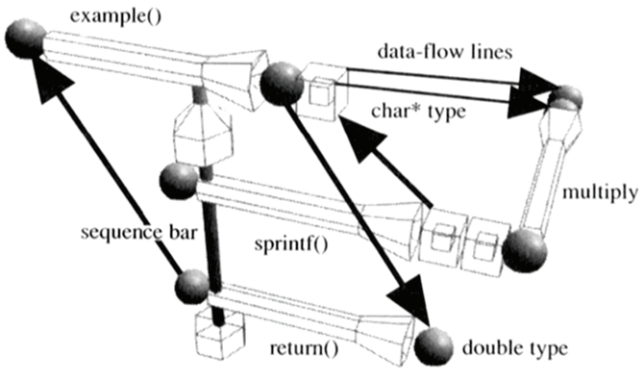
Abbildung 12: Programmierumgebung und resultierendes Bild in Quartz Composer

Abbildung 12 zeigt auf der linken Seite die visuelle Programmierumgebung von Quartz Composer. In dem dargestellten visuellen Programm wird ein Verlaufshintergrund, eine mit einem Bild texturierte Geometrie und ein sich drehender Text, der von einem Interpolator gesteuert wird, erzeugt. Auf der rechten Seite der Abbildung sieht man die resultierende Bild. In Quartz Composer stehen verschiedene Funktionen für die Bildverarbeitung, das Rendering aber auch Interaktionen mit Komponenten des Betriebssystems zur Verfügung.

2.4.2 *Visuelle Programmiersprachen in 3D*

Lingua Graphica

Lingua Graphica (Stiles & Pontecorvo, 1992) ist ein früher Ansatz einer universellen visuellen Programmiersprache, welche allgemeine C-Programme abbilden kann. Sie hat die üblichen Probleme von visuellen Sprachen, welche auf eine vorhandene imperative Programmierung aufgesetzt werden.




```
double example(double num, char *str)
  {
    sprintf(str," %f",(num*num));
    return(num);
  }
```

Abbildung 13: Beispielprogramm von Lingua Graphica und das entsprechende C-Programm – aus (Stiles & Pontecorvo, 1992)

Oftmals ist die ursprüngliche Darstellung der textuellen Programme einfacher zu erfassen als die Umsetzung in eine visuelle Darstellung. Auch die Ausnutzung einer dreidimensionalen Darstellung machen die Programme eher unübersichtlicher, da keine echte Semantik durch die Ausnutzung der dritten Dimension erreicht wird.

Abbildung 13 zeigt ein Beispielprogramm aus Lingua Graphica und das ursprüngliche textuelle Programm. Dieses Beispiel lässt schon erahnen, dass es sich bei Lingua Graphica eher um das Aufzeigen der prinzipiellen Machbarkeit der Umsetzung von textuellen nach visuellen Sprachen, als um ein praktikables Tool zur Programmierung, handelt. Besonders deutlich wird hier das Problem der Umsetzung von rein imperativen Sprachen in visuelle Konzepte. Die folgenden Beispiele zeigen, dass bessere visuelle Programmierumgebungen möglich sind, wenn auf anderen Programmier-Prinzipien z.B. der logischen Programmierung oder der Datenflussprogrammierung aufgesetzt wird.

Cube

Ein neuerer Ansatz von Programmierung in drei Dimensionen ist die visuelle Programmiersprache Cube (Marc A. Najork & Kaplan, 1991). Im Gegensatz zu dem vorherigen Beispiel baut Cube auf eine eigene von einem textuellen Programm unabhängige Semantik seiner visuellen Konstrukte auf, welche auch explizit die dritte Dimension als eine Sinn tragende Erweiterung einführt. Damit lassen sich relativ übersichtlicht durch die Schachtelung der Programmstrukturen z.B. auch rekursive Programmstrukturen darstellen. Abbildung 14 zeigt auf der linken Seite ein einfaches Programm zur Umrechnung der Einheit Celsius nach Fahrenheit mittels einer Multiplikation und einer Addition und auf der rechten Seite ein rekursives Programm zur Berechnung der Fakultätsfunktion (!).

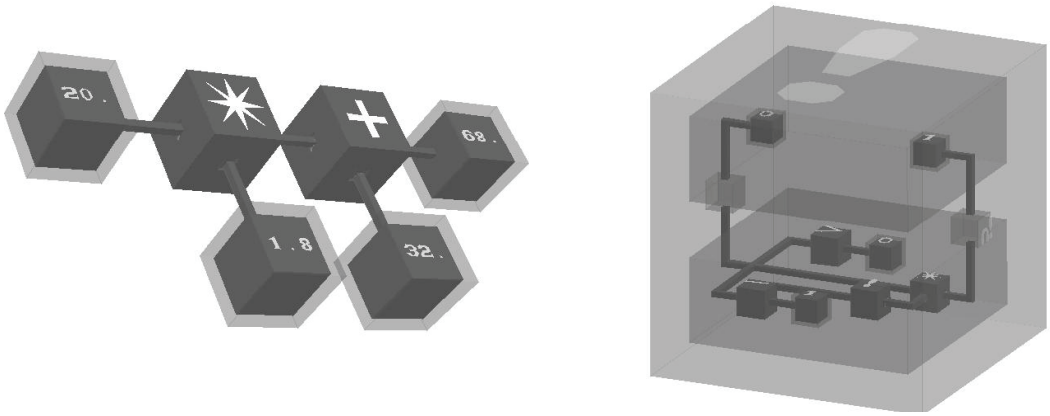


Abbildung 14: Zwei Beispiele der Programmiersprache Cube – aus (Marc A. Najork & Kaplan, 1991)

Eine Besonderheit von Cube ist, dass die Richtung des Datenflusses und damit der Rechenoperationen nicht fest vorgegeben ist, sondern je nach bestehender Belegung der Variablen (in der Abbildung als transparente Würfel dargestellt) die unbelegten Variablen nach Möglichkeit ausrechnet. Somit funktioniert das Beispiel auf der linken Seite sowohl für die Umrechnung von Celsius nach Fahrenheit, als auch umgekehrt, je nachdem ob die Variable für den Celsiuswert oder die für Fahrenheit belegt wird.

3D-PP

Die Programmiersprache 3D-PP (Oshiba & Tanaka, 1999) ist eine dreidimensionale Programmiersprache, welche auf die logische Programmiersprache GHC (Ueda, 1985) aufsetzt. Die einzelnen Programmeinheiten bestehen aus dreidimensionalen Containern, welche durch die Klauseln in der logischen Programmiersprache erzeugt werden. Hierbei entsteht für jedes Goal einer Klausel ein Container mit den aufgeworfenen Subgoals im inneren. Parameter der Goals werden als Felder auf dem Rand der Container dargestellt. Verbindungen zwischen diesen Feldern ergeben sich aus der Belegung von Parametern mit gleichen Variablen. Die Programmcontainer erlauben durch die Schachtelung, welche sich durch die Goals in den einzelnen Klauseln ergibt, eine hierarchische Einteilung der Programmeinheiten. Werden bei der Ausführung der Programme neue Subgoals aufgeworfen werden diese direkt in der Visualisierung als neue dreidimensionale Container dargestellt. Damit lassen sich das Programm selber und auch der Programmablauf einer logischen Programmiersprache visualisieren.

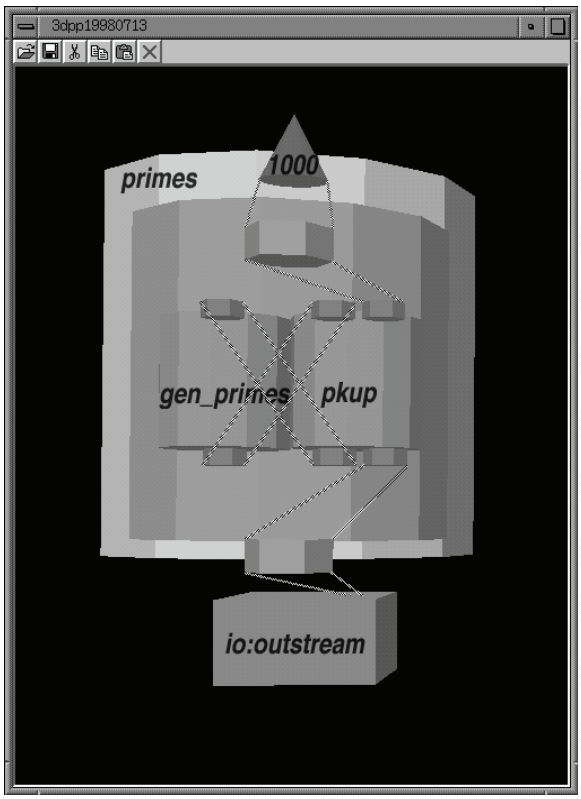


Abbildung 15: Beispielprogramm von 3D-PP zur Berechnung von Primzahlen – aus (Oshiba & Tanaka, 1999)

Abbildung 15 zeigt ein Programm von 3D-PP mit dessen Hilfe die 1000. Primzahl ausgerechnet und auf einen Ausgabestrom geschrieben wird.

SAM

Die visuelle Programmierumgebung SAM (Geiger *et al.*, 1998) wurde für die Modellierung und Simulation von Agenten und den Aufbau von Geometrie und Animationen in Virtuellen Umgebungen entwickelt. Sie ist daher nicht direkt zu dem Bereich der Datenflussprogrammierung zuzuordnen, aber sie hat zwei in dem Zusammen-

hang der vorliegenden Arbeit relevante Aspekte: Zum einem werden die Programmabläufe der Agenten ihrer internen Regeln und der von ihnen verschickten Nachrichten dreidimensional und animiert dargestellt, zum anderen implementiert sie eine abstrakte Beschreibungssprache für Szenengraphstrukturen mit denen im System die virtuelle Welt aufgebaut wird, deren Ablauf durch die Agenten bestimmt wird.

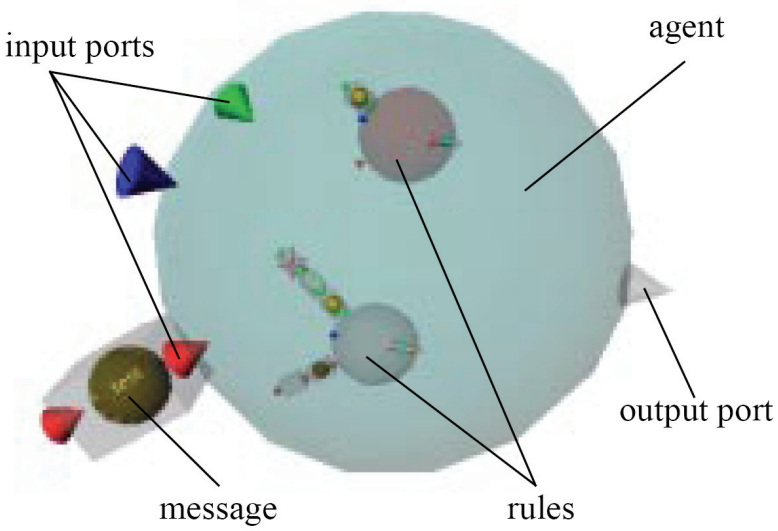


Abbildung 16: Darstellung eines Agenten, seiner Regeln und der Informationsaustausch über Nachrichten – aus (Geiger et al., 1998)

Abbildung 16 zeigt die Visualisierung eines Agenten in SAM, dessen Regeln als Objekte im Inneren der Agentenvisualisierung dargestellt werden. Verschiedene Eingabe- und Ausgabeports sind an dessen Rand platziert, an denen entsprechende Nachrichten andocken können.

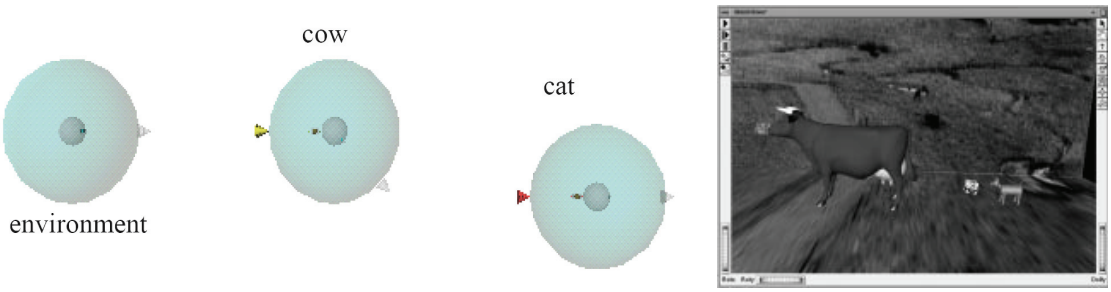


Abbildung 17: Darstellung von drei Agenten und der damit definierten Virtuellen Umgebung – aus (Geiger et al., 1998)

Abbildung 17 zeigt das konkrete Beispiel von drei Agenten, und der durch SAM aufgebauten virtuellen Welt deren Ablauf durch die Regeln der Agenten definiert wird.

2.4.3 Daten-Visualisierungstools

Bei VR-Tools, die zur Visualisierung von Datensätzen in der Virtuellen Umgebung dienen, werden die Rohdaten durch mehrere Vorverarbeitungsschritte aufbereitet. Für diese Art der Verarbeitung werden oft Datenflusskonstrukte eingesetzt. Diese einzelnen Vorverarbeitungsschritte werden dabei mit Unterstützung eines graphischen Systems zusammengebaut. Ein Beispiel hierfür sind Datenvisualisierungstools wie

IRIS-Explorer (Foulser, 1995) von SGI, IBM's *Data-Explorer* (Abram & Ternish, 1995) und *AVS* (AVS). Für ein VR-basiertes Programmiersystem sind bei den folgenden Beispielen vor allem der Aufbau von interaktiven GUIs und die Visualisierungsmöglichkeiten von komplexen Datentypen und deren Steuerung interessant.

Die Datenquellen sind bei diesen Programmen in der Regel feste oder zeitlich veränderliche Datensätze mit unterschiedlich vielen Dimensionen. Die visuellen Programme werden in 2D auf einer graphischen Bedienoberfläche zusammengestellt. Die Programmknoten, welche diese Datensätze verarbeiten, dienen dann z.B. dazu, bestimmte Daten auszuwählen, sie einzufärben, Isoflächen aus Volumendaten zu extrahieren, verschiedene Transformationen auf die Daten anzuwenden und sie schließlich darzustellen oder in einen anderen Datensatz zu exportieren.

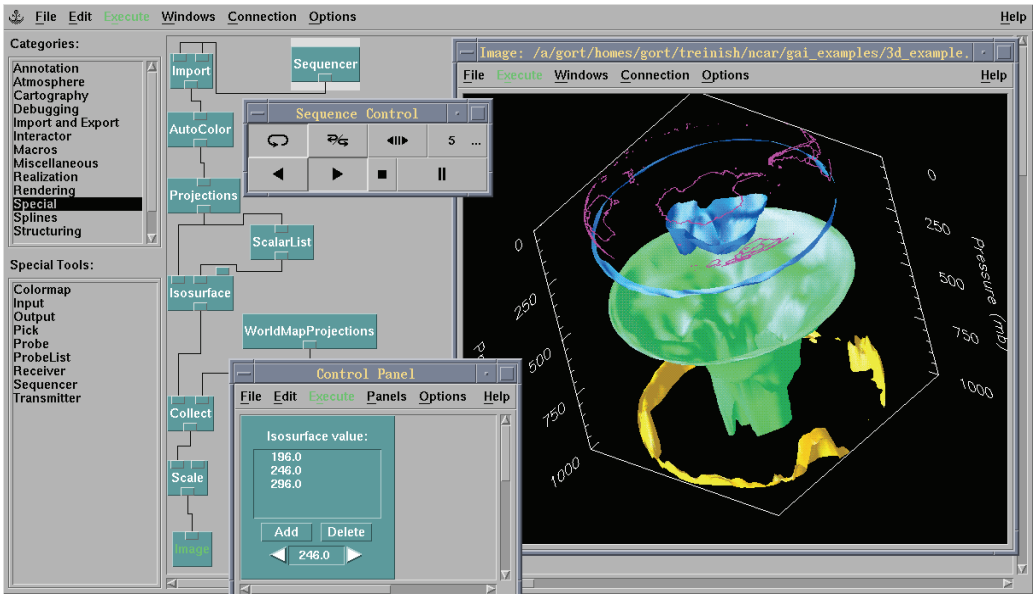


Abbildung 18: Visuelles Programm im IBM Data Explorer zur Darstellung eines komplexen Datensatzes – aus (Abram & Ternish, 1995)

Abbildung 18 zeigt ein visuelles Programm und die daraus resultierende Visualisierung im Data Explorer von IBM. Die Datenquelle ist hier ein multidimensionaler Datensatz mit zeitlichen Veränderungen. Der zeitliche Ablauf des Datensatzes wird über einen Sequenzer (im Bild hell unterlegt) gesteuert. Aus dem Datensatz werden dann drei Isoflächen extrahiert und als dreidimensionale animierte Projektion angezeigt.

Zusätzlich können interaktive Kontrollen eingebaut werden, welche Parameterwerte einzelner Recheneinheiten oder den Fluss der Daten verändern. Diese Knoten erzeugen eine graphische Benutzeroberfläche mit der das Programm gesteuert werden kann. Im Beispiel aus Abbildung 18 sind in dem Fenster „Control Panel" die Werte für Isoflächen in der Visualisierung angezeigt, welche dort auch verändert werden können.

Abbildung 19 zeigt am Beispiel des Data Explorer den Einsatz von drei Selektor-Knoten, welche den Datenfluss des Programms über spezielle Switch- bzw. Route-Knoten (im Bild hell unterlegt) steuern. In dem Beispiel aus Abbildung 18 erlaubt eine interaktiv veränderbare Liste im „Control Panel" die Festlegung der Werte für die Berechnung der Isoflächen.

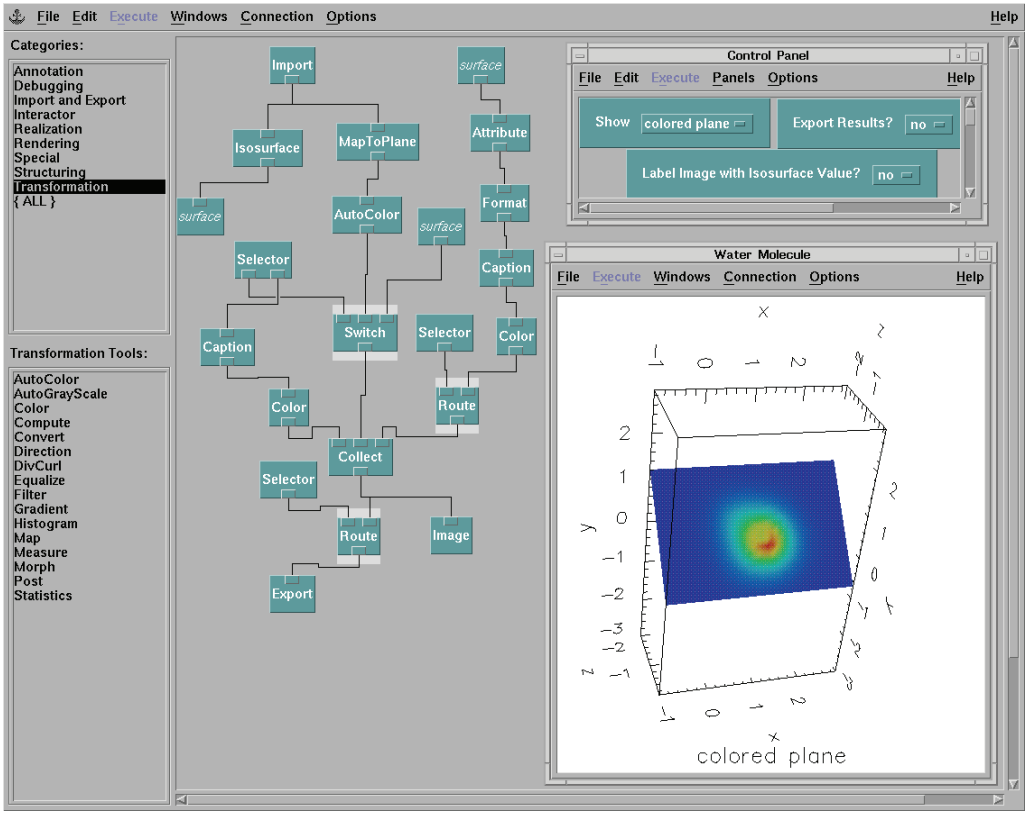


Abbildung 19: IBM Data Explorer: Kontrolle des Datenflusses durch ein GUI – aus (Abram & Ternish, 1995)

Die Programmierumgebungen der Datenflussprogramme im Bereich der Verarbeitung von komplexen Datensätzen zur wissenschaftlichen Visualisierung erlauben den Aufbau komplexer Datenflussprogramme mit einer durch ein GUI unterstützten Steuerung des Datenflusses. Die vorhandenen Programmknoten sind hauptsachlich für die Konvertierung, Filterung und Visualisierung großer Datensätze ausgelegt. Abgesehen von der dreidimensionalen Ausgabe der Datenvisualisierungen am Monitor oder einem immersiven VR-Ausgabegerät, ist der Überschneidungsbereich mit einer Programmierumgebung für immersive Virtuelle Umgebungen aber gering. Die Programme werden über einfache Benutzereingaben (mittels einer zweidimensionalen Oberfläche) erstellt und durch ein zweidimensionales GUI gesteuert. Die Verarbeitung von Benutzereingaben in den Programmen selber – was für die Modellierung der Interaktionen in der virtuellen Welt zentral ist – kommt in dem Bereich dieser Visualisierungstools nicht vor. Somit wird in diesen Tools auch keine Interaktion in der Virtuellen Umgebung ermöglicht[4].

2.5 Datenfluss in 3D und Virtueller Realität

In diesem Abschnitt wird aufgezeigt, welche Teile aus den Methoden der Datenflussprogrammierung schon in dreidimensionale Programmierumgebungen und für den Einsatz zur Modellierung virtueller Welten eingesetzt werden.

[4] Die einzige Interaktion die von manchen der Visualisierungstools mit der dreidimensionalen Darstellung vorgesehen sein kann, ist eine einfache Navigation in der erstellten Visualisierung.

Im Bereich der *Virtuellen Realität (VR)* werden Teile der Datenflussmodelle schon längere Zeit mit unterschiedlichem Fokus erfolgreich eingesetzt. Das Datenflusskonzept wird z.B. für die Vorverarbeitung von zum Teil komplexen, hochdimensionalen Daten zur Visualisierung am Bildschirm oder in der VR benutzt. Ein weiterer verbreiteter Einsatz ergibt sich im Zusammenhang mit dem Einsatz von *Szenengraphen*, welche oft zur Beschreibung von hierarchischen Strukturen in virtuellen Welten benutzt werden. Die klassische Programmierschnittstelle (*Application Programming Interface, API*) eines Szenengraphs wurde von OpenInventor (P. Strauss & Carey, 1992) eingeführt und von vielen späteren VR-Tools übernommen. Bei der Einbindung von Datenflusskonzepten in Szenengraph-APIs wird ein Datenflussgraph orthogonal zum vorhandenen Szenengraph aufgebaut, in dem interne Werte der Knoten im Szenegraph zusammen mit externen Benutzereingaben in Beziehung gesetzt werden.

Die folgenden Abschnitte geben eine Übersicht über den Einsatz von Methoden der Datenflussprogrammierung in interaktiven Virtuellen Umgebungen und zeigen einige für die vorliegende Arbeit relevante Entwicklungsumgebungen auch für die Verarbeitung von Benutzereingaben in diesen Umgebungen.

2.5.1 Szenengraph- und Datenflussbasierte VR-Tools

Viele VR-Programmierumgebungen benutzen Szenengraphen und reichern diese durch Teile von Methoden aus der Datenflussprogrammierung an. Das zentrale Konzept hierbei ist die Bereitstellung von internen Parametern der Szenengraphknoten über Felder und die Propagierung der Feldwerte über Feldverbindungen. Im Folgenden wird dargestellt, inwieweit schon Teile der Datenflusskonzepte in aktuellen VR-Tools verwendet werden.

Einer der Vorreiter und prominentestes Beispiel für die Einbindung eines Datenflussmodells innerhalb einer über einen Szenengraphen beschriebenen virtuellen Welt ist die VR-Beschreibungssprache *VRML*. Ausgehend von dem Datenformat und dem Konzept eines Szenengraph-API mit Datenflusskonstrukten aus IRIS Inventor (P. S. Strauss, 1993) entstand die Beschreibungssprache VRML, welche dann zu *VRML2.0* bzw. *VRML97* (Carey & Bell, 1997) und schließlich zu dem *X3D-Standard* (X3D Working Group, 2002) weiterentwickelt wurde. Diese mit einem Datenflusssystem angereicherten Szenengraph-Umgebungen werden im den folgenden Abschnitten als mögliche Zielplattformen für die visuelle Programmierung hin untersucht.

Open Inventor

Open Inventor (P. Strauss & Carey, 1992; Wernecke, 1994) ist eine aus dem Szenengraphentool IRIS Inventor von SGI entstandene Open-Source Entwicklung. Die Szenengraphstruktur und das Datenformat das von Inventor eingeführt wurde, ist die Grundlage vieler Szenengraphtools unter anderem auch von VRML und X3D. Neben dem Aufbau von Szenengraphstrukturen erlaubt Open Inventor für die Möglichkeit der Erstellung von direkten Manipulationsmethoden auch den Aufbau eines Datenflussgraphen.

Einfache Interaktionen werden durch vordefinierte Manipulator-Knoten realisiert. Hierbei werden einfache Benutzerevents, wie z.B. von der Tastatur oder Maus, durch spezielle Traversierungsfunktionen im Szenengraph verteilt, welche von den Manipu-

lator-Knoten aufgefangen und verarbeitet werden. Ein klassisches Beispiel für solch einen Manipulator ist ein Trackball.

Open Inventor erweitert das statische Szenengraphmodell neben den vordefinierten Manipulatoren schon durch flexible dynamische Komponenten. Dafür werden im Konzept von Open Inventor sog. *Engines* eingeführt. Die Engines verhalten sich wie Komponenten in einem Datenflussprogramm. Der Datenfluss wird hierbei über Felder zwischen den Engines weitergegeben und Feldverbindungen propagieren die Werte der Felder. Zur Berechnung der Datentoken steht eine Evaluationsfunktion zur Verfügung, welche ausgeführt wird, sobald ein Feld der Engine sich ändert. In der Evaluierungsfunktion können neu berechnete Werte auf die Ausgabefelder der Engine gesetzt werden. Dadurch lassen sich mittels dieser Knoten Datenflussnetzwerke aufbauen.

Dieses erste Konzept der Grundfunktionalität eines Szenengraph-API, zusammen mit der Möglichkeit Datenflusskonstrukte im Szenengraphen zu verankern, bildete auch die Grundlage für die Entwicklung der VR-Beschreibungssprache VRML und X3D

VRML und X3D

Die *Virtual Reality Markup Language (VRML)* wurde 1995 für die Beschreibung und den Austausch von virtuellen Welten über das Internet entwickelt, um diese direkt in einem lokalen Internetbetrachterprogramm *(„Browser")* darstellen zu können. VRML definiert die Sprachkonstrukte und ihre Interpretation für diese Welten. Die Darstellung selbst wird durch eigenständige Betrachter oder Zusatzprogramme *(„Plugins")* des Browsers übernommen. VRML hat als Beschreibungssprache einen ersten Standard für die Beschreibung von interaktiven virtuellen Welten definiert. Als Hauptproblem bei der Umsetzung in entsprechende Betrachterprogramme hat sich die Vielfalt der einzelnen heterogenen Plattformen ergeben, auf der sich je nach Fähigkeit des lokalen Systems eine möglichst große Teilmenge der VRML-Konstrukte darstellen lassen sollte. Neben dem Aufbau einer Szenengraph-Struktur für die Platzierung von Objekten, wurde in den ersten Weiterentwicklungen von VRML auch die Möglichkeit geschaffen, Werte von Objekten in diesem Szenengraph über einen Datenfluss-Mechanismus verändern zu können. Da viele dieser zusätzlichen Eigenschaften des VRML-Standards in der Weiterentwicklung X3D ausgebaut und konkretisiert wurden, wird im Folgenden auf den X3D-Standard (X3D Working Group, 2002) Bezug genommen. Neben den klassischen Knoten im Szenengraphmodell eingesetzten Knoten für Transformation, Material und Geometrie definiert X3D auch Sensor- und Scripting-Knoten, welche für das Datenflussmodell von Bedeutung sind. Sensorknoten bilden das Interface für die Eingabe z.B. von Daten von Benutzerinteraktionen. Durch Scripting-Knoten ergibt sich die Möglichkeit, den vorhandenen Satz von Szenengraph- und Sensor-Knoten mit eigenen Knoten zu erweitern. Im Zuge der Weiterentwicklung von VRML nach X3D sind diese Knoten auch in der Lage Teile des Datenflussgraphen zu bilden.

Abbildung 20 zeigt das Beispiel einer Definition und den resultierenden Szenengraph mit den entsprechenden Datenfluss-Verknüpfungen für eine einfache Animation in VRML. Der VRML-Code beschreibt eine virtuelle Szene mit einer Box, welche bei Berührung anfängt sich zu drehen. Zuerst wird ein Szenengraph mit einer einfachen Geometrie in Form einer Box aufgebaut. Diese Box ist mit einem *TouchSensor* versehen, welcher bei Berührung ein Trigger-Event aussendet. Dieser triggert einen *Time-*

Sensor, welcher dann in einem festen Intervall einen Zeitwert generiert. Ein *OrientationInterpolator* berechnet aus diesem Zeitwert eine Rotationsmatrix, welche schließlich die Transformation der Box bestimmt.

Die rechte Seite von Abbildung 20 zeigt den resultierenden Szenengraph, wobei die geraden Linien die Szenengraphstruktur und die geschwungenen Linien die Verbindungen im Datenflussgraphen darstellen.

```
#VRMLV2.0utf8
Viewpoint {position 0 0 5}
DEF BOX_POS Transform {
  children [
    Shape {
      geometry Box {}
    },
    DEF BOX_TOUCH TouchSensor {},
  ]
}
DEF BOX_TIMER TimeSensor {
  cycleInterval 5
  startTime - 1
}
DEF BOX_ENGINE OrientationInterpolator {
  key [0, .5,1]
  keyValue [ 0 1 0 0, 0 1 0 3.14, 0 1 0 6.28]
}
ROUTE BOX_TOUCH.touchTime TO
          BOX_TIMER.set_startTime
ROUTE BOX_TIMER.fraction TO
          BOX_ENGINE.set_fraction
ROUTE BOX_ENGINE.value_changed TO
          BOX_POS.set_rotation
```

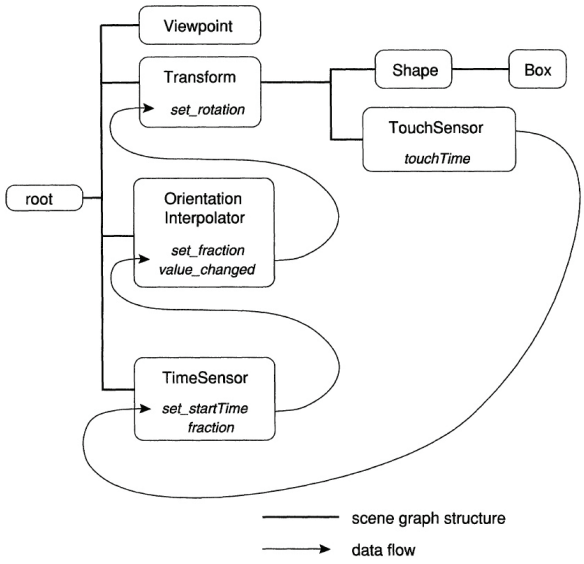


Abbildung 20: VRML-Code und Szenengraph einer einfachen Animation – aus (Steed & Slater, 1996)

Da VRML bzw. X3D den Standard für die Beschreibung interaktiver virtueller Welten darstellen und oft als Maßstab oder mögliche Zielplattform von VR-Anwendungen angesehen wird, geht der folgende Abschnitt näher auf die Verankerung des Datenflusskonzeptes in dieser Sprache ein. Die grundlegende Eigenschaft für die Realisierung eines Datenfluss-Konzeptes in X3D ist das *Event-Modell*:

Alle Ereignisse in X3D, wie z.B. zeitgesteuerte Animationen, Objekt-Auswahl, Erkennung von Benutzereingaben und Kollisionserkennung werden von *Events* gesteuert. Die Laufzeitumgebung bestimmt die Propagierung dieser Events bezüglich einer festgelegten Menge von Regeln.

Die Knoten der Datenflussprogrammierung in X3D sind eine Teilmenge der im Szenengraph vorhandenen Knoten. Sie definieren Eingabefelder, welche bei Empfang eines Events eine Aktion auslösen und Ausgangsfelder, welche ihrerseits wieder Events verschicken können. So können bestimmte Events z.B. den internen Status eines Knotens oder den Wert seiner Ausgangsfelder ändern.

Um die Ausgabeevents eines Knoten an die Eingabefelder eines anderen Knoten zu leiten, werden *Event-Routen („Routes")* definiert. Das entsprechende **ROUTE**-Statement in X3D etabliert dabei Eventstrecken, die den Verbindungen zwischen Knoten in dem Datenfluss-Ablaufmodells (siehe Abschnitt 2.2.1) entsprechen.

In dem Ablaufmodell von X3D werden die initialen Events von Sensor- oder Scripting-Knoten generiert. Diese werden dann über die definierten Routen an andere Knoten weitergegeben, welche dementsprechend neue Events generieren können, bis alle Routen betrachtet wurden. Diesen Ablauf nennt man *Event-Kaskade („event cascade")*, innerhalb der alle Events einen gleichen Zeitstempel zugewiesen bekommen.

Falls durch simultane Eventgenerierung mehrerer Sensoren mehrere Events mit gleichem Zeitstempel ankommen, generieren diese alle Events innerhalb einer Event-Kaskade. Die Reihenfolge der Bearbeitung gleichzeitiger Events wird nicht als signifikant angesehen. Jeder Knoten sollte für einen Zeitwert höchstens ein Event pro Ausgabefeld generieren. Dementsprechend darf jede Route auch nur ein Event innerhalb einer Kaskade transportieren.

Nach der Abarbeitung aller Events der initialen Event-Kaskade übernimmt ein Post-Prozess die Aktionen, welche durch die Kaskade angeregt wurden. Der Browser sollte folgende Aktionssequenz innerhalb eines Zeitwertes abarbeiten:

a: Neuberechnung der Kamera aufgrund der Position und Orientierung des Betrachterstandortes

b: Berechnung der Eingabe der Sensoren

c: Evaluierung der Event-Routen

d: Falls Events durch Schritt b oder c generiert wurden: weiter bei b

Falls Scripting-Knoten unterstützt werden, können weitere Zwischenschritte hinzukommen. Abbildung 21 zeigt das Konzept des Event- und Ablaufmodells in X3D. Der dort gezeigte Szenengraph umfasst neben den üblichen SG-Knoten auch Sensor und Scripting-Knoten. Die initialen Events der Scripting-Knoten und alle generierten Events der Knoten im Szenengraph werden über die „Execution Engine" und dem Routen-Graphen weiter zu den durch die Routen definierten Knoten weitergeleitet. Scripting-Knoten können zusätzlich zu einem selber Events generieren und zu den Knoten im Szenengraph schicken, zum anderem haben sie die Möglichkeit, bestehende Routen zu löschen oder neue aufzubauen.

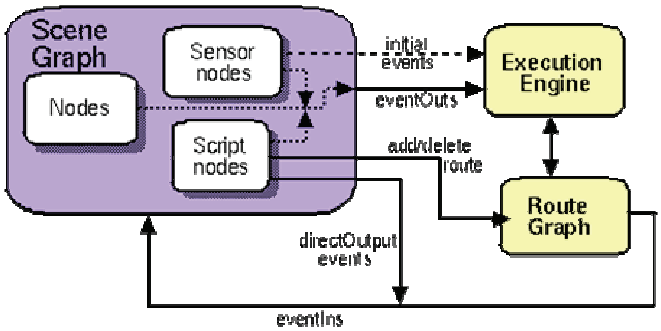


Abbildung 21: Konzept des X3D-Ablaufmodells – aus (X3D Working Group, 2002)

Event-Kaskaden können Schleifen enthalten, wobei ein Event weitere erzeugt und schließlich das ursprüngliche Event noch einmal erzeugt wird. Durch die Einschrän-

kung von einem Event pro Feld und Route innerhalb einer Kaskade ist es möglich, diese durch zyklische Abhängigkeiten entstandenen Schleifen aufzubrechen.

In dem X3D-Modell ist es generell möglich, dass Eingabefelder mehrere ankommende Routen haben. Die eventuell gleichzeitig ankommenden Events werden in beliebiger Reihenfolge abgearbeitet. Ebenso können von Ausgabefeldern mehrere Routen ausgehen. Das entsprechende Event wird dann an alle verbundenen Felder weitergegeben. Beides steht im Kontrast zu dem klassischen Datenfluss-Ablaufmodell, bei dem dieses durch spezielle Merge- bzw. Switch-Knoten erreicht werden kann (siehe Abschnitt 2.2.1).

OpenSG

Das VR-Toolkit OpenSG bietet eine Open-Source Entwicklung eines Szenegraphen unter anderem mit der Unterstützung von aktuellen Shader-Technologien (R. Fernando & Kilgard, 2003). Der Hauptunterschied zu klassischen Szenengraphtool ist das Konzept des Einsatzes von speziellen Programmkernen („*Node-Cores*“), welche die Funktionalität der generischen Knoten des Szenengraph bestimmen. Diese Node-Cores können von mehreren generischen Knoten gemeinsam genutzt werden. Dieses *Core-Sharing* ermöglicht die sonst übliche Szenengraphstruktur des gerichteten azyklischen Graphen auf eine einfache Baumstruktur zu reduzieren. Das mehrfache Instanziieren von Objekten, was bei Szenengraphen mit azyklischer Graphstruktur durch das Einhängen eines Knotens unter mehreren Elternknoten geschieht, wird bei OpenSG durch die gemeinsam genutzten Node-Cores realisiert. OpenSG erlaubt eine Verteilung des Szenengraphs auf mehrere Instanzen, welche auch auf unterschiedlichen Rechnern laufen können und dann über Netzwerk synchronisiert werden. Dieses ermöglicht eine komfortable Realisierung eines verteilen Rendersystems in Form von Renderclustern, z.B. für Mehrseitenprojektionen wie die CAVE (Cruz-Nera *et al.*, 1993).

Die Szenengraphknoten von OpenSG sind mit Feldern versehen, die die internen Parameter des Knotens setzen. Dementsprechend können auch Feldverbindungen zwischen diesen Feldern etabliert werden, um Feldwerte zwischen den Knoten zu propagieren. Allerdings ist in OpenSG kein Mechanismus für ein Eventsystem vorgesehen, dass eine Berechnung und Weitergabe von Datentoken in Form einer Datenflussprogrammierung vorsieht. Der Vorteil von OpenSG ist aber die komplette Realisierung als Open-Source Projekt ohne die Abhängigkeiten von einem kommerziellen Produkt.

Für den Einsatz als virtuelle Programmierumgebung bietet sich OpenSG als ein komfortables und Open-Source basiertes Szenegraphtool an, zusammen mit einer flexiblen Möglichkeit zum Einsatz bei Mehrseitenprojektionen. Für die Realisierung einer Datenflussprogrammierung stehen allerdings nur einfache Feldkonzepte, ohne ein für die Datenflussprogrammierung brauchbares Eventsystem, zur Verfügung.

Avango

Avango – ursprünglich benannt als Avocado (Tramberend, 1999) – ist ein VR-Toolkit, das auf *OpenGL-Performer* (Silicon Graphics Inc., 2002) von SGI aufsetzt. OpenGL-Performer bietet eine high-performance Entwicklungsumgebung für szenengraphbasierte, interaktive Virtuelle Umgebungen. Avango erweitert den Szenengraphen von Performer um einen Zugriff auf die internen Werte der einzelnen Objekte über

Datenfelder. Zusammen mit der Möglichkeit, diese Felder über Feldverbindungen miteinander zu koppeln wird auch hier ein zu dem Szenengraph orthogonaler Datenflussgraph aufgebaut. Eine Besonderheit dieses Frameworks ist die Möglichkeit, Feldverbindungen über Netzwerkverbindungen zwischen unterschiedlichen Rechnern miteinander zu verbinden und so z.B. die Daten mehrerer Instanzen eines Szenengraphs auf verschiedenen Rechnern synchron zu halten. Dieser verteilte Szenengraph ermöglicht die gleichzeitige Darstellung einer Szene auf mehreren Rechnern für ein verteiltes Rendering, welches von immersiven Virtuellen Umgebungen mit mehreren Projektionen – wie z.B. der CAVE (Cruz-Nera et al., 1993) – benötigt wird. Zusätzlich bietet Avango ein Scripting-Interface mit dem sich einfach und zur Laufzeit Änderungen an der Szenengraphstruktur über die Manipulationen von Feldwerten und Feldverbindungen vornehmen lassen. Die dafür eingesetzte Scriptsprache ist die mit der funktionalen Programmiersprache Lisp verwandte Sprache Scheme (Abelson *et al.*, 1991).

Das Ablaufmodell von Avango ähnelt in vielen Bereichen dem von X3D. Auch hier werden Knoten aus dem Szenengraph als Knoten für die Datenflussprogrammierung benutzt. Die einzelnen Knoten – in Avango auch *Feldcontainer* („*field container*“) genannt – definieren Felder, welche über Feldverbindungen gekoppelt werden können. Die Feldwerte werden direkt beim Setzen des Feldwertes an verbundene Felder propagiert. Jeder in der Datenflussprogrammierung aktive Knoten hat eine *Evaluate*-Methode, in der Berechnungen gemacht und Feldwerte gesetzt werden können. Einmal pro Zyklus zur Berechnung eines neuen Bildes (*Render-Frame*) wird die Evaluate-Methode von allen Knoten aufgerufen, deren Felder neue Werte erhalten haben. Werden dabei Feldwerte verändert und über Feldverbindungen an andere Knoten propagiert, werden diese zur Berechnung noch innerhalb des aktuellen Frames angemeldet. Das Verhalten ist somit dem Prinzip der Event-Kaskade von X3D sehr ähnlich. Allerdings sind in Avango von sich aus keine Zeitstempel der Feldwerte vorgesehen. Um die Reihenfolge der Evaluierung der Knoten z.B. bei mehreren eingehenden Feldverbindungen beeinflussen zu können, werden die Knoten zur Evaluation in Queues mit unterschiedlicher Priorität (*„evaluation bins“*) aufgeteilt. Hierbei wird sichergestellt, dass die komplette Propagierung zwischen Knoten in einer Queue mit höherer Priorität auf jeden Fall vor der Evaluierung der Knoten des höherwertigen Evaluierungssets ausgeführt wird. Während die normale Verarbeitung der Feldwerte in der Evaluate-Methode der Knoten passiert, kann auf Feldwertänderungen eines Knotens auch direkt asynchron reagiert werden. Dafür definiert jeder Knoten einen Callback der direkt bei jeder Feldwertänderung aufgerufen wird.

Sowohl VRML/X3D als auch AVANGO erlauben im Prinzip den Einsatz als visuelle Programmierumgebung. Die vorhandenen Knoten, welche für die Berechnung der Datentoken im Datenflussgraphen vorgesehen sind, stellen nur wenige vorgefertigte Möglichkeiten der Interaktion oder Animation in der Virtuellen Umgebung zur Verfügung. Eine Verschaltung der vorhandenen Knoten zu einem Programm das eine Animation oder Interaktion steuert ist möglich, aber aufgrund der begrenzten Vorgabe von Knoten, die als mögliche Bausteine einer Datenflussprogrammiersprache in Frage kommen, aber auch wegen der fehlenden Möglichkeiten zur Steuerung des Datenflusses begrenzt.

Im Gegensatz zu X3D hat AVANGO schon Grundinteraktionen in immersiven Virtuellen Umgebungen und die Anbindung von externen VR-Eingabegräten vorbereitet und bietet auch die Möglichkeit, neue Knoten für die echtzeitfähige Berechnung von Daten zu programmieren. X3D hingegen hat aufgrund der Konzipierung für den Einsatz als Austauschformat im Internet zu viele Nachteile. Zum einem erlaubt X3D aufgrund der Kompatibilitätsanforderungen zu verschiedenen, plattformübergreifenden Browsern nur die Erweiterung über die Scripting-Knoten und ist damit auf den Einsatz von Java-Script angewiesen, was in echtzeitkritischen Berechnungen für IVEs nicht brauchbar ist. Zum anderen sind die vorgesehenen Interaktionen von X3D hauptsächlich auf den Einsatz in einer 2D-Oberfläche vorgesehen und die Einbindung in eine IVE aufgrund der fehlenden möglichen Anbindung an externe Prozesse und Daten problematisch.

Die Knoten, welche die Werte für den Datenflussgraphen berechnen, werden bei den oben genannten Tools in der Regel mit in den Szenengraph gehängt. Falls sie aber selber keine aktive Funktion in der Szenengraphstruktur übernehmen, sind sie nur für die Übersicht bei der Ausgabe der Szenengraphstruktur dort aufgenommen. Es ist aber nicht vorgesehen, dass diese Knoten selbst eine Geometrie in der virtuellen Welt oder in einem eigenen Fenster erzeugen. Eine visuelle Programmierumgebung kann die Eigenschaft, dass diese Knoten sich mit im Szenengraph befinden ausnutzen, um sich selbst zu visualisieren und ein Interface zur Interaktion mit dem visuellen Programm bereitzustellen. Dabei muss aber darauf geachtet werden, dass die Geometrie, die für die visuellen Programme erzeugt wird, von weiteren Knoten, die dieser Knoten eventuell noch mit aufbaut, getrennt gehalten wird.

In allen VR-Toolkits, welche auch Möglichkeiten der Datenflussprogrammierung bereitstellen, ist die Berechnung der Datentoken direkt durch die Updaterate im Berechnungs- und Renderprozess gekoppelt. Daher ist eine zeitlich unabhängige Berechnung der Benutzer-Eingabedaten in der Granularität, wie sie durch die Datenrate der Eingabegeräte gegeben ist, nicht vorgesehen. Eine zeitlich exakte Verarbeitung von Benutzer-Eingabedaten ist aber an machen Stellen z.B. für eine exakte Analyse von Bewegungstrajektorien unerlässlich.

VEDA

Der erste (und bis dahin einzige) Ansatz der Verbindung von Datenflussprogrammierung mit einer immersiven virtuellen Programmierumgebung wird von Steed und Slater in (Steed & Slater, 1996) mit der „*Virtual Environment Dialogue Architecture*“ (*VEDA*) aufgezeigt. VEDA ist ein System zur Beschreibung von virtuellen Welten und ein erster Ansatz einer visuellen Programmiersprache innerhalb einer Virtuellen Umgebung. VEDA ist als Applikation in das *Distributed Virtual environment System* (*dVS)* (Grimsdale, 1991) eingebettet und kommuniziert mit diesem System über eine Datenbankschnittstelle. Das System erweitert dVS um ein Event-Modell, das dem von VRML 2.0 sehr ähnlich ist. Zusätzlich stellt VEDA ein graphisches Frontend in der Virtuellen Umgebung zur Verfügung, über das Knoten im Datenflussgraphen kopiert und gelöscht werden können und sich Verbindungen zwischen den Komponenten erstellen lassen. Knoten werden als dreidimensionale Objekte in der Virtuellen Umgebung dargestellt, deren Eingabe- und Ausgabeobjekte sich als einfache geometrische Objekte in der Nähe befinden. In Abbildung 22 sieht man einen einfachen Knoten, der eine logische UND-Verknüpfung darstellt. Die beiden Eingabeobjekte und das

Ausgabeobjekt des Knotens sind als Dreiecke rechts bzw. links von dem Objekt zu sehen.

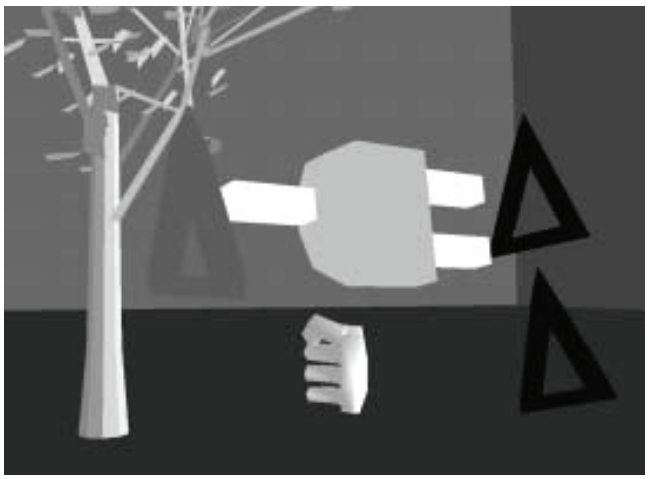


Abbildung 22: Ein "UND" -Knoten in VEDA – aus (Steed & Slater, 1996)

Abbildung 23 zeigt die visuelle Darstellung einer Verschaltung von Datenflussknoten zur Interaktion in der Virtuellen Umgebung. Hierbei wird die charakteristische Kopfbewegung während des Laufens auf der Stelle durch ein neuronales Netz detektiert, und die Position des Betrachters gegebenenfalls in Blickrichtung verschoben.

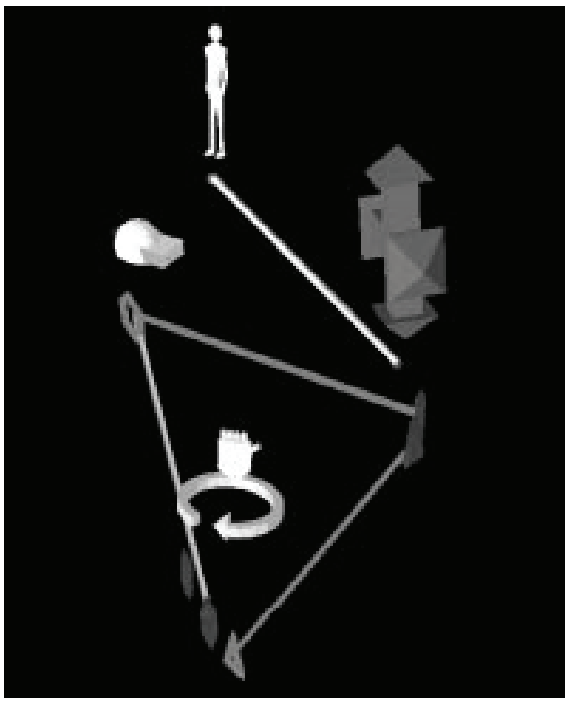


Abbildung 23: Die virtuelle Darstellung eines Datenflusskonstruktes in VEDA – aus (Steed & Slater, 1996)

Die fest vorgegebenen Komponenten in VEDA bilden eine sehr grobe Struktur von Datenflusskomponenten. Auch scheint die rein ikonische Darstellung ohne Text in der Virtuellen Umgebung – selbst wenn die Möglichkeit der Kapselung von Programmteilen gegeben ist – nicht für eine komplexere Verschaltung einer höheren Anzahl von Komponenten geeignet zu sein. Durch die externe Anbindung von VEDA an ein sehr spezielles, datenbankbasiertes VR-Tool ergeben sich neben dem auf dieses

plattformgebundene Tool begrenztes Einsatzgebiet noch weitere Probleme: Da der Update der Sensordaten und das Rendering in externen Prozessen abgearbeitet werden, kann VEDA als Client-Komponente einer Datenbank z.B. keine zeitlichen Informationen für Optimierungen bezüglich der Abfrage der Sensordaten oder dem Zeitpunkt des Renderings in seine Berechnungen einbeziehen. Um den zeitlichen Versatz von Eingabedaten und Darstellung *(„Lag“)* – der bei immersiven Virtuellen Umgebungen als kritisch anzusehen ist (siehe auch (Bryson, 1996; Kreylos *et al.*, 2001)) – und den Berechnungsaufwand möglichst gering zu halten, ist eine mit der Sensorrate oder der Darstellungsrate synchronisierte Berechnung notwendig. Der Einsatz von aktuellen VR-Tools, welche oft schon Teile einer Datenflussprogrammierung, oder zumindest die Anbindung von Datenflussprogrammen über ein Feldkonzept und Eventsystem bieten, ist hier vorzuziehen.

2.5.2 Verarbeitung von Benutzer-Eingabedaten in IVEs

Auch bei der Verarbeitung von Benutzerdaten in IVEs haben sich Datenflussprogramme etabliert. Der kontinuierliche Strom von Eingabedaten z.B. durch Positionstracker oder Datenhandschuhe eignet sich gut zur Verarbeitung durch Methoden der Datenflussprogrammierung. Im Folgenden werden vorhandene Systeme und Frameworks zur Verarbeitung von Benutzereingaben in Virtuellen Umgebungen vorgestellt.

OpenTracker

Eine Einsatzmöglichkeit ergibt sich bei der Vorverarbeitung von Trackerdaten. Trackingsysteme von IVEs sind in der Regel externe Applikationen, welche die Daten in einem eigenen Format und bezüglich eines eigenen Referenzsystems zur Verfügung stellen. OpenTracker (Reitmayr & Schmalstieg, 2001) ist ein System, das die Schnittstelle zu verschiedenen Trackingsystemen vereinheitlicht und Vorberechnungen – wie z.B. Koordinatentransformationen – durchführt.

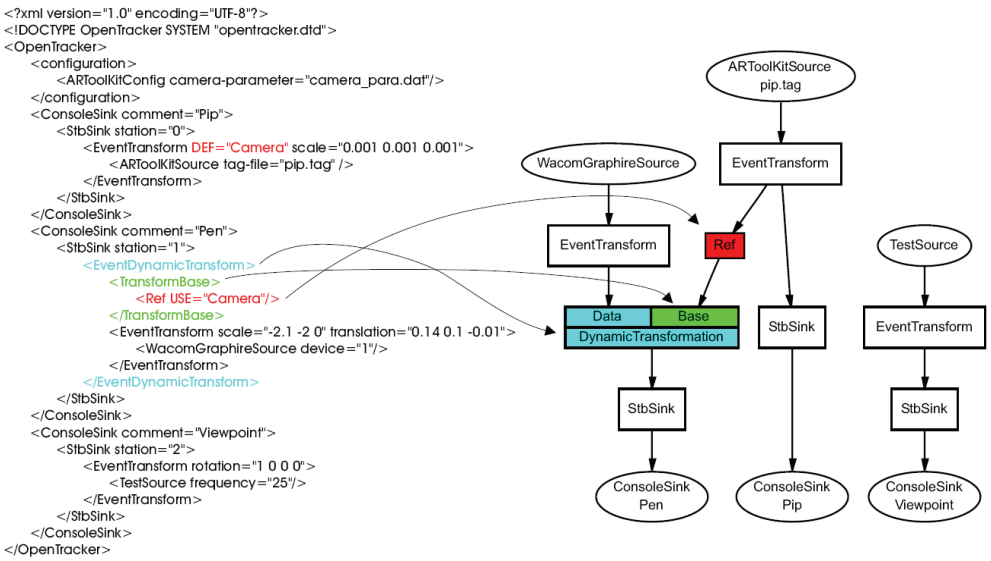


Abbildung 24:OpenTracker XML-Definition und Datenflussgraph – aus (Reitmayr & Schmalstieg, 2001)

OpenTracker bietet eine Anbindung an verschiedene 3D-Trackingsysteme, aber auch an andere Eingabegeräte wie z.B. Graphiktabletts. Das System definiert dafür ein XML-Format um die Datenflussgraphen über eine textuelle Beschreibung aufbauen zu können.

Abbildung 24 zeigt die Definition durch ein solches XML-File und den resultierenden Datenflussgraphen. Gewöhnungsbedürftig ist bei der XML-Notation, dass die äußeren Knoten in der XML-Beschreibung den Datensenken entsprechen, welche im Datenflussgraphen unten stehen. Offensichtlich ist diese Umkehrung der Notation darin begründet, dass XML-Formate nur Baumstrukturen direkt abbilden können, Datenflussprogramme aber gerichtete azyklische Graphen bilden. Da Joint-Links im Gegensatz zu Replica-Links in der Regel ausgeschlossen sind (siehe Abschnitt 2.2.1), bilden Datenflussprogramme von den Datensenken aus betrachtet, eine Baumstruktur, was die direkte Abbildung in ein XML-Format erlaubt.

OpenTracker erlaubt die Integration von Daten aus verschiedenen Eingabegeräten mit unterschiedlichen Taktraten. Es eignet sich daher sehr gut für die Vorverarbeitung von Eingabedaten für VR-Tools. Da es aber als von der VR-Applikation unanhängiger Prozess läuft, kann es keine auf den Renderzeitpunkt der Applikation bezogene Optimierung der Berechnungen durchführen.

PrOSA

Das in (Latoschik, 2001b) vorgestellte PrOSA-Framework *(PrOSA: Pattern On Sequences of Attributes)* bietet einen mehrschichtigen Ansatz zur Verarbeitung von Eingabedaten und Steuerung von Interaktionen in IVEs. Der Haupteinsatzzweck von PrOSA ist die Erkennung von Gesten des Benutzers zur multimodalen Integration mit natürlicher Sprache. Für die Erkennung der Gesten werden drei Abstraktionsebenen eingeführt.

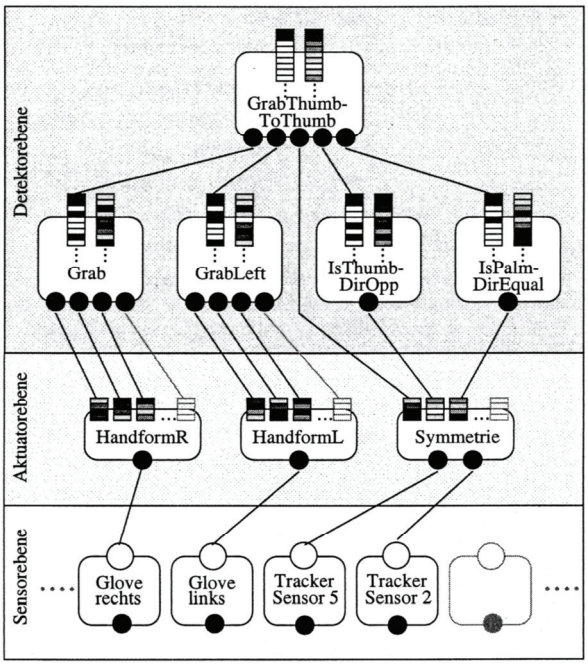


Abbildung 25: Die drei Verarbeitungsebenen in PrOSA – aus (Latoschik, 2001b)

In der untersten *Sensor-Ebene* werden die Eingabedaten des Benutzers durch Sensoren bereitgestellt. Hierbei werden die Daten der einzelnen Eingabegeräte in Datenkanälen zusammengefasst. Um die unterschiedlichen Taktungen der Eingabedaten repräsentieren zu können, werden die Daten mittels Attribut-Sequenzen zwischen den Komponenten weitergegeben. Attribut-Sequenzen enthalten eine Sequenz von Daten mit jeweils einem Zeitstempel der den Zeitpunkt der Entstehung des Datums enthält. Attribut-Sequenzen ermöglichen die möglichst exakte Zusammenführung von mehreren Datenquellen mittels linearer Interpolation.

In der zweiten Ebene, der *Aktuator-Ebene* werden die unterschiedlichen Datenquellen integriert und synchronisiert, d.h. auf eine einheitliche Taktung gebracht. Die Aktuatoren berechnen aus den ursprünglichen Daten auch die Transformation der Daten in den VR-Bezugsraum und abgeleitete Daten, wie z.B. Symmetrien, Winkel, Geschwindigkeiten und Beschleunigung. Auf dieser Ebene erfolgen alle quantitativen, numerischen Berechnungen der Werte, die in der nächsten Ebene für die symbolische Verarbeitung gebraucht werden.

Die dritte *Detektor-Ebene* verknüpft die von der Aktuator-Ebene produzierten Daten auf einem symbolischen Level miteinander. Die Detektoren in dieser Ebene suchen Muster durch die logische Verknüpfung atomarer Testbedingungen, um die Bewegungsinformationen zu klassifizieren. Zu diesen Operationen gehören vor allem Schwellwert-Berechnungen und die boolsche Verknüpfung von Bedingungen für die Klassifikation der auftauchenden Bewegungsmuster. Sind die Bewegungen des Benutzers durch die Detektoren klassifiziert, können diese, zusammen mit der Integration der sprachlichen Äußerungen des Benutzers, eine Interaktion in der Virtuellen Umgebung auslösen.

Die Übertragung auf die Interaktion in der Virtuellen Umgebung erfolgt durch *Motion-Modifikatoren*. Diese abstrahieren und diskretisieren nach der Erkennung die meist unpräzisen Benutzerbewegungen um exakte Daten für die Manipulation der Objekte bereitzustellen. Die resultierende Operation, wie z.B. die Veränderungen im Szenengraph, selber wird dann anhand dieser Daten durch *Manipolatoren* ausgeführt.

Das PrOSA-Framework enthält einen interessanten Ansatz um bei der Berechnung Daten mit einer externen Taktung eine von dem externen Takt zeitlich unabhängige Berechnung der Eingabedaten durchzuführen. Mit Hilfe von *Attribut-Sequenzen*[5] kann trotz der externen Steuerung der Berechnungen eine eigene Taktung der berechneten Daten garantiert werden.

In der Gesamtstruktur des Frameworks ist in der Aktuator-Ebene nur eine Berechnungsschicht vorgesehen, welche schon die Eingabedaten synchronisiert, auf eine einheitliche Taktung bringt und alle numerischen Berechnungen durchführt. Eine flexible visuelle Programmiersprache benötigt eine feiner unterteilte Berechnung der Eingabedaten. Auch die strikte Trennung der Berechnung von quantitativen Werten in der Aktuator-Ebene und symbolischen Verknüpfungen in der Detektor-Ebene kann und sollte bei einer visuellen Programmierumgebung flexibler gestaltet werden.

[5] Auf die genaue Art der Berechnung von Datentoken mit Hilfe von Attribut-Sequenzen wird im Abschnitt 4.4.2 eingegangen.

2.6 Zusammenfassung

Auch wenn die visuelle Programmierung sich nicht als das universelle Werkzeug im Bereich der Softwareentwicklung herausgestellt hat, wurde in Abschnitt 2.1 gezeigt, dass es doch viele Anwendungsgebiete gibt, wo sie dem Programmierer von großem Nutzen sein kann.

Abschnitt 2.2 zeigt, dass es im Bereich der Datenflussprogrammierung einen Trend von den traditionellen feinkörnigen Datenflusssprachen hin zu grobkörnigen Programmstrukturen gibt. Man erreicht dieses durch eine Zusammenfassung von sequentiellen Abschnitten oder mit Hilfe einer zusätzlichen textbasierten, imperativen Sprache innerhalb der Knoten der Datenflussprogrammiersprache. Hierbei bleiben die grundlegenden, formalen Eigenschaften der Datenflussgraphen erhalten. Die Datenflusssprache erhält den Status einer Koordinierungssprache, welche den Ablauf der Programme definiert, während die imperative Sprache die Berechnungsvorschriften angibt. Gleichzeitig wird damit von der problematischen Spezialisierung auf Hardware, welche eigens für Datenflusskonstrukte optimiert ist, Abstand genommen und der Einsatz von aktuellen Multithreading- und Multiprozessorarchitekturen für Datenflussprogramme unterstützt.

Wie in Abschnitt 2.3 gezeigt, hat sich vor allem der Zusammenschluss von visueller Programmierung und Datenflussprogrammierung als ein erfolgreicher Ansatz gezeigt. Besondere Vorteile im Bereich der visuellen Programmierung ergeben sich durch reaktive DFVPLs, bei denen der Datenfluss während des Programmablaufes visualisiert und verändert werden kann. Dieses eröffnet durch das direkte Eingreifen in laufende Programme auch äußerst komfortable Test und Debugging-Möglichkeiten. Da die Programmierumgebung für die visuelle Programmierung eine entscheidende Rolle spielt, und gerade reaktive DFVPLs hohe Anforderungen an Visualisierungs- und Interaktionsmöglichkeiten stellen, werden immersive Virtuelle Umgebungen für solche Sprachen interessant.

Andere, bereits vorhandene Ansätze visueller Programmierung in zwei und drei Dimensionen und auch Einsatz von visuellen Programmiersprachen in Visualisierungstools zeigt Abschnitt 2.4. Der professionellste Einsatz von visuellen Datenflussprogrammen findet sich in Datenvisualisierungstools, welche in Abschnitt 2.4.3 vorgestellt werden. Hier werden visuelle Programme für die Berechnung einer Visualisierung komplexer Datensätze eingesetzt. Diese Programme sind interaktiv veränderbar und können auch eine graphische Benutzeroberfläche aufbauen. Eine Anbindung an Benutzereingaben in einer Virtuellen Umgebung und für die Interaktion in VEs bieten sie aber nicht.

Im Bereich der Programmierung von interaktiven Virtuellen Umgebungen gibt es schon viele Toolkits, welche Teile der Datenflusskonzepte für die Festlegung der Abläufe in der virtuellen Welt einsetzen. Abschnitt 2.5.1 zeigt von diesen einige Beispiele. Bei den aktuellen VR-Tools ist aber, obwohl die Konzepte der Datenflussprogrammierung für die Propagierung über Feldwerte und einzelner Knoten zur Berechnung der Feldwerte aufgegriffen werden, keine Programmierumgebung für visuelle Datenflussprogramme realisiert worden. Einige VR-Toolkits – wie z.B. AVANGO – bieten aber prinzipiell die Möglichkeit durch die Erweiterung des vorhandenen Eventsystems und mit einer zusätzlichen Visualisierung der Rechenknoten eine visuelle Datenflussprogrammierumgebung aufzubauen.

Auch die Festlegung der Berechnungszeitpunkte durch die Berechnungs- und Darstellungsschleife und die dadurch abhängige Taktung der verrechneten Eingabedaten bei den betrachteten VR-Tools ist zunächst problematisch. Abschnitt 2.5.2 zeigt vorhandene Systeme für die Bearbeitung dieser Eingabedaten auf. Für eine exakte zeitliche Berechnung bei der Integration von unterschiedlichen Datenquellen bietet das PrOSA-Framework einen guten Ansatz zur Entkopplung des Berechnungstaktes und der Datenrate der berechneten Datentoken. Bei der Vorverarbeitung der Eingabedaten von Trackingsystemen in VR-Umgebungen gibt es z.B. mit OpenTracker einen guten Ansatz auf dem Konzept von Datenflussprogrammiersprachen, aber keine direkte Anbindung an ein VR-Tool für den zeitlichen Abgleich der Berechnungen mit dem Zeitpunkt der Darstellung und die Anbindung der resultierenden Operationen in der visuellen Umgebung.

Ein Ziel des in dieser Arbeit entwickelten Programmiersystems ist es, die Ansätze von Datenflusskonstrukten in aktuellen VR-Tools zu einer vollständigen Datenflussprogrammierung in immersiven Virtuellen Umgebungen auszubauen. Zusätzlich soll die Visualisierung der Abläufe und Zwischenergebnisse der visuellen Programme eine reaktive visuelle Programmiersprache erschaffen, die es erlaubt, Programmabläufe während ihrer Ausführung in der Virtuellen Umgebung zu testen, zu analysieren und zu optimieren.

Weiterhin sollen existierende Frameworks für von der Framerate der Applikation unabhängige Berechnungen in die visuelle Programmiersprache eingebunden und um die Möglichkeit einer flexiblen Anpassung auf die unterschiedlichen Erfordernisse nach Exaktheit bzw. geringe Latenzzeiten der Daten erweitert werden.

Hierbei muss die Realisierung des Systems auf die für Berechnungen in Echtzeit nötige Leistungsstärke der eingesetzten Programmiersprachen und Konzepte der Datenflussprogrammierung Rücksicht nehmen.

Eine komfortable Beschreibungssprache der Knoten soll sicherstellen, dass das System ohne großen Aufwand erweitert werden kann. Die Konzeption eines generalisierenden Layers für die Operationen mit Transformationen und Veränderungen in der Szenengraphstruktur, ermöglicht die Anbindung an verschiedene VR-Tools, ohne den Code der Programmknoten verändern zu müssen.

3 Constraints

Die Berechnung von Daten über ein Datenflussprogramm und die Propagierung der berechneten Werte an Felder der Szenengraphknoten ermöglicht schon eine weit reichende Modellierung von Interaktionen in einer szenengraphbasierten VR-Applikation. Ein zusätzliches, in dieser Arbeit entwickeltes Feature im Umgang mit Feldern und ihren Wertebelegungen ist aber auch die Überwachung und Einschränkung von Feldwerten.

Für den Umgang mit definierten Einschränkungen und Abhängigkeiten von Werten in einem Gleichungssystem existiert ein ganzer Bereich von wissenschaftlichen Arbeiten im Bereich der Lösung von Constraintproblemen. Da sich die theoretischen Grundlage bei der Formulierung und Lösung von Constraintproblemen von der Theorie der Datenflussprogrammierung unterscheidet, aber zu konzipierende visuelle Programmiersystem auch eine Behandlung von lokalen Constraints ermöglichen soll, werden in diesem Kapitel zunächst die für die im Rahmen dieser Arbeit wichtigen Grundlagen von Constraint-Lösunssystemen zusammengefasst.

Die Szenengraphstruktur in den VR-Tools impliziert schon selber eine Form von Constraints, welche die Abhängigkeit der Positionen der Objekte von den übergeordneten Transformationsknoten betreffen. Das Konzept der Feldverbindungen in aktuellen VR-Tools bildet zusätzlich eine Form von Constraints, welche die Parameter der Knoten koppeln können. Die Definition zusätzlicher Constraints kann vor allem bei dem Einsatz in Virtuellen Umgebungen das Verhalten der Anwendung komfortabel modellieren.

Für den Einsatz in der virtuellen Konstruktion werden hauptsächlich geometrische Constraints benötigt. Dieses Kapitel zeigt daher nach einer kurzen Übersicht über allgemeine Constraints und Lösungssysteme (*Constraint-Solver*) vor allem lokale geometrische Constraints für den Einsatz in szenengraphbasierten Umgebungen und für den Bereich der Festlegung von Verbindungseigenschaften der Bauteile in der in dieser Arbeit behandelten, virtuellen Konstruktion auf. Der letzte Abschnitt befasst sich mit der Einbindung von Lösungsstrategien für Constraints im Bereich der vorhandenen Programmierumgebungen für die Virtuelle Realität.

3.1 Constraint-Typen

Im Allgemeinen ist ein Constraint eine Relation über eine oder mehrere Variabeln, welche die ganze Zeit eingehalten werden soll. Die Angabe von Constraints ist in der Regel deklarativ, d.h. es wird nur angegeben, welche Bedingung erfüllt sein soll, aber nicht wie und ob dieses – z.B. bei mehreren Constraints, die sich auf dieselben Variablen beziehen – erreicht werden kann. Da auch schon einfach zu beschreibende Zusammenhänge komplizierte Lösungen erfordern können[6], werden Constraint-Lösungssysteme auf bestimmte Domänen oder Typen von Constraints eingeschränkt.

[6] Ein klassisches Beispiel für ein einfach zu formulierendes Constraintproblem, dessen Lösung sich aber als hochkomplex herausgestellt hat, ist die Lösung der Gleichung: $x^n+y^n=z^n$, $n>2$ $x,y,z \in \aleph$.

3.1.1 Funktionale Constraints

Funktionale Constraints geben einen eindeutigen Wert für eine Variable vor, falls alle anderen Variablen mit Werten belegt sind. Die funktionale Eigenschaft kann sich auch nur auf einen Teil der Variablen in der Gleichung erstrecken, wenn die beschriebene Funktion nicht bijektiv ist.

Ein einfaches Beispiel:

x=y

Dieser Constraint ist funktional für beide Variablen x und y, da die Werte direkt aus der Belegung der jeweils anderen Variablen folgen.

Weitere funktionale Constraints werden z.B. durch die Grundrechenarten plus (+) und minus (–) geformt.

So sind z.B. die folgenden Constraints:

a+b=c

g–h=i+j

funktional für alle vorkommenden Variablen.

Die Rechenoperationen mal (*) und geteilt (/) sind im Allgemeinen nicht funktional, da z.B. die Formel 0*a=b keinen eindeutigen Wert für die Variable a erzwingt. Diese Fälle lassen sich aber gesondert behandeln, so dass unter Ausschluss dieser Fälle die Operationen als funktional behandelt werden können.

Der Vorteil von funktionalen Constraints ergibt sich durch die eindeutige Belegung der Variablen, ohne dass z.B. andere Constraints in Betracht gezogen werden müssen. Dieses ermöglicht eine lokale Strategie bei der Lösung von Constraintsystemen.

Nicht bijektive Funktionen sind nicht für alle Variablen funktional. so ist z.B. der Constraint:

$$x = \sin(\alpha)$$

funktional in x, aber nicht in α.

Eine allgemeinere mathematische Formulierung von Constraints erfolgt über Relationen. Diese sind im Allgemeinen für keine der beteiligten Variablen funktional.

Einfache Beispiele: Die Einschränkung einer Variablen auf positive Werte:

$$x \geq 0$$

oder eine Variable als Obergrenze einer anderen:

$$a > b$$

Hier liegt für keine Variable ein eindeutiger Wert fest. Somit reicht für die Lösung eines Constraintproblems mit relationalen Constraints eine rein lokale Strategie in der Regel nicht aus.

3.1.2 Allgemeine Constraints

In einer allgemeineren Form kann man Parameter auf eine vorgegebene Menge von Werten einschränken:

$$x \in M$$

wobei: M = Menge der erlaubten Werte

Die Einschränkung auf einen Bereich zwischen einem Minimum und einem Maximum sieht in dieser Form so aus:

$$x \in [Min, Max]$$

Damit können einzelne Parameter z.B. noch auf Rasten einer festgelegten Schrittweite eingeschränkt werden:

$$x \in \{O + n \cdot W \mid n \in \aleph\}$$

wobei: O = fester Offset

W = Schrittweite

Bsp. Einschränkung eines Rotationswinkels auf Werte der Schrittweite W=90.

x=0, 90, 180, 270, 360, …

$$x \in \{n \cdot 90 \mid n \in \aleph\}$$

Eine ähnliche Einschränkung kann auch für Vektoren vorgenommen werden, indem sie auf eine Menge von vorgegebenen Vektoren beschränkt werden.

$$\vec{x} \in M$$

wobei: M = Menge erlaubter Vektoren

Beispiel: Einschränkung einer Drehachse auf die Hauptachsen.

$$\vec{x} \in \{(1,0,0),(0,1,0),(0,0,1),(-1,0,0),(0,-1,0),(0,0,-1)\}$$

Da die oben angegebenen Constraints aber nur eine deklarative Angabe der Einschränkungen darstellen, müssen auch Strategien entwickelt werden, welcher der möglichen Werte bei einer Verletzung der Constraintbedingung für eine Variable gesetzt wird.

3.1.3 Constraint-Propagierung

Mehrstellige Relationen ermöglichen die Kopplung mehrerer Variablen. Bei der Lösung eines Problems mit mehrstelligen Constraints müssen aber nicht nur die Werte der einzelnen Variablen in Beziehung gesetzt werden, sondern auch den Ablauf der Wertpropagierung festlegen.

Der einfachste Fall sind *gerichtete Constraints* – wie im folgenden Beispiel – zwischen zwei Parametern, wobei es einem bestimmenden (x_1) und einem abhängigen Parameter (x_2) gibt. Ändert der bestimmende Parameter seinen Wert wird dieser direkt an den abhängigen propagiert. Eine Änderung des abhängigen Parameters hat hingegen

keine Auswirkung und wird entweder direkt unterbunden oder bei der nächsten Wertpropagierung wieder überschrieben.

Oftmals ist hier einfach eine gerichtete Propagierung der unveränderten Werte gewünscht:

$$x_1 \Rightarrow x_2$$

Die Feldverbindungen bei den Szenengraphtools, wie sie in Abschnitt 2.5.1 beschrieben wurden, sind ein Beispiel für Constraints, welche so eine gerichtete Gleichheit zwischen den Feldwerten etablieren.

In der Praxis stellen bei einem Constraint-Solver mit lokaler Propagierung die einzelnen Typen von Constraints in einem System Methoden zu ihrer Bearbeitung zur Verfügung. Der Constraint-Solver wählt dann nach einer eigenen Strategie die passenden Methoden zur Lösung des Problems aus.

So wird z.B. der Constraint:

a+b=c

durch die folgenden drei Methoden repräsentiert:

a := c – b;

b := c – a;

c := a + b;

Im Bereich der Constraint-Lösungssysteme mit lokaler Propagierung gibt es Ansätze, welche speziell auf gerichtete Constraints ausgelegt sind, z.B. in (S. E. Hudson, 1991).

Sollen die Parameter sich gegenseitig beeinflussen können, muss auf andere Propagierungsstrategien zurückgegriffen werden. Ungerichtete Constraint bergen immer die Gefahr bei der gleichzeitigen Änderung mehrerer Parameter undeterministisches Verhalten zu zeigen, da das Resultat von der zeitlichen Verlauf der Wertänderung abhängig ist. Spezielle Systeme für die Behandlung von ungerichteten Constraints erlauben den Constraints verschiedene Methoden, welche das System zur Lösung des Constraintproblems heranziehen kann (Sannella *et al.*, 1993; Zanden, 1996).

Eine Lösungsstrategie ist die Ausrichtung der ungerichteten Constraints anhand der aktuellen Veränderung der Werte für die Variablen (z.B. in (Zanden, 1988)). Hierbei werden die Constraints immer so ausgerichtet, dass in der aktuellen Berechnung veränderte Variablen ihren Wert in Richtung der unveränderten Variablen propagieren. Die eindeutige Ausrichtung kann aber problematisch sein, z.B. wenn Schleifen in der Abhängigkeiten der Constraints auftreten (siehe unten).

Schränkt der Constraint beide Parameter in Abhängigkeit von einander ein, ist aber ein gerichteter Constraint nicht mehr sinnvoll, da hier der Fall eintreten kann, dass beide Werte verändert werden müssen, um den Constraint zu erfüllen. Ein Beispiel ist die Einschränkung von zweidimensionalen Koordinaten auf die Kreisfläche eines Einheitskreises um den Nullpunkt

$$\sqrt{x_1^{\,2} + x_2^{\,2}} \leq 1$$

Hier können je nach Anwendung unterschiedliche Strategien sinnvoll sein, eine erlaubte Konfiguration der Parameter herzustellen. Es kann z.B. erst ein Parameter auf seinen Definitionsbereich und dann der zweite in Abhängigkeit vom ersten eingeschränkt werden. Im Falle des oben angegeben Beispiels kann, falls der Punkt außerhalb des Kreises liegt, z.B. die Projektion des Vektors (x_1, x_2) auf den nächsten Punkt der Kreisoberfläche sinnvoll sein.

Constraint-Zyklen

Oftmals werden Constraints formalisiert, indem die Variablen als Knoten und die Constraints als Kanten in einem *Constraintgraphen* dargestellt werden. Hierbei können ungerichtete Constraints als einfache Kanten und gerichtete Constraints als gerichtete Kanten (Pfeile) dargestellt werden.

Die Formalisierung von Constraintgleichungen als Graph erlaubt die einfache Erkennung von Zyklen im Constraintgraphen.

Stellt man z.B. die folgenden (ungerichteten) Constraintgleichungen:

a+b=c

c=d

d*b=g

als Graph dar, ergibt sich ein Graph mit einer zyklischen Verbindungsstruktur zwischen den Variablen b, c und d, wie in Abbildung 26 zu sehen ist.

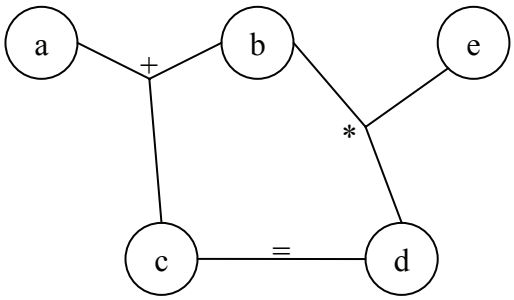


Abbildung 26: Constraintgraph mit zyklischen Constraints

Für Lösungsmethoden mit einer lokalen Propagierung sind Zyklen im Constraintgraphen ein Problem, da es im Allgemeinen nicht möglich ist, eine Lösung des Constraintproblems zu erreichen, indem man die lokalen Constraints der Reihe nach abarbeitet.

Abbildung 27 zeigt die ersten zwei Schritte im Ablauf einer lokalen Propagierung im Constraintgraphen für ein Beispiel mit den zwei gerichteten Constraints:

$$y \Leftarrow x$$

$$x \Leftarrow y \cdot y$$

Bei einer ungünstigen initialen Belegung der Variablen wird die rein lokale Propagierung der Constraints in diesem Beispiel keine der beiden möglichen Lösungen (x=y=0 oder x=y=1) finden, sondern in einer endlosen Berechnungsschleife hängen bleiben.

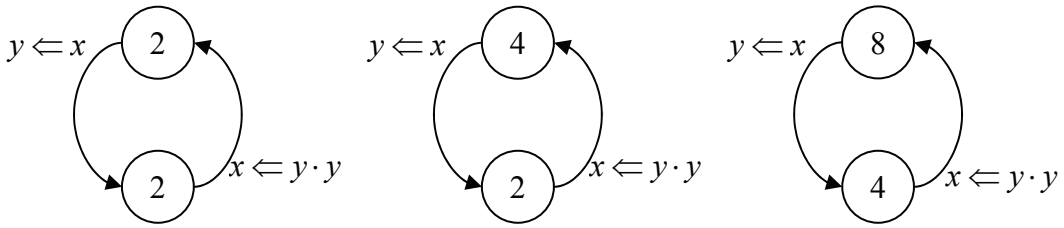


Abbildung 27: Lokale Propagierung bei zyklischen Constraints

Trotz dieser Probleme bei zyklischen Constraints hat ein lokaler Lösungsansatz entscheidende Vorteile gegenüber einer globalen Methode:

- *Effizienz*: Die lokale Propagierung kann oft effizienter passende Methoden der einzelnen Constraints zur Lösung eines Satzes von Constraints finden als globale Methoden.
- *Allgemeingültigkeit*: Constraint-Solver mit lokaler Propagierung können unterschiedliche Berechnungsmethoden der Constraints verarbeiten. Insbesondere können sie mit nicht numerischen Constraints umgehen.
- *Nachvollziehbarkeit*: Während bei manchen globalen Contraint-Solvern kaum nachzuvollziehen ist, auf welchen Weg sie eine spezielle Lösung generiert haben, kann bei der lokalen Propagierung über die Methoden, die eine Variable setzen der Wert dieser Variablen nachvollzogen werden.

Prioritäten

Da Constraints generell ohne Einschränkung gelten sollen, kann es z.B. im Falle von einem überbestimmten Constraintproblem zu unlösbaren Konfigurationen kommen. Um mit widersprüchlichen Constraints umgehen zu können, werden in Constraint-Lösungssystemen oft Prioritäten für die einzelnen Constraints vergeben – siehe z.B. (Borning *et al.*, 1992). Eine klassische Prioritätenliste, wie sie in Constraint-Lösungssystemen eingesetzt wird, ist in der folgenden Tabelle zu sehen.

Required	Höchste Priorität; Konflikte mit Constraints gleicher Priorität fatal
Strong	Höchste Priorität der optionalen Constraints; kann nur durch Constraints der Priorität „Required“ überschrieben werden
Medium	Mittlere Priorität; nicht in allen Systemen vorhanden
Weak	Niedrige Priorität; wird bei Konflikten von allen anderen Constraints außer Kraft gesetzt

Tabelle 1: Mögliche Prioritäten von Constraints

Die höchste Priorität entspricht dabei der ursprünglichen Definition von Constraints: der generellen Gültigkeit eines Constraints. Unlösbare Konflikte zwischen den Constraints dieser Klasse sind fatal, d.h. der Algorithmus zu Constraintlösung bricht in der Regel erfolglos mit der Ausgabe einer Fehlermeldung ab. Die Constraints der niedrigeren Prioritätsklassen können hingegen bei Konflikten durch stärkere Constraints außer Kraft gesetzt werden und somit trotzdem eine Lösung gefunden werden.

3.2 Constraints für Szenengraph-Transformationen

Grundlegende geometrische Constraints in VR-Umgebungen ist die Einschränkung der Transformationen, die durch die Knoten im Szenengraph realisiert werden. Eine für Szenengraphen allgemein übliche Beschreibungsform von Transformationen ist in Form von homogenen Transformationen gegeben, welche durch eine 4x4-Matrix repräsentiert werden. Diese Art der Repräsentation ermöglicht die Verkettung von Transformationen durch die Multiplikation ihrer entsprechenden Matrizen.

Da sich die Gesamttransformation eines Objektes aus der Multiplikation aller übergeordneten Matrizen in Szenengraphen ergibt, bildet die Szenengraphstruktur gerichtete Constraints der Form:

$$M_{glob} \Leftarrow M_1 \cdot M_2 \cdot \ldots \cdot M_N$$

wobei M_{glob} die globale Transformation und $M_1 \ldots M_N$ die Matrizen der Transformationsknoten im Szenengraphen – angefangen bei Wurzelknoten des Graphen bis zum Knoten des Objektes selber – sind. Ein generelles Problem der Szenengraphstruktur ist durch diesen gerichteten Constraint gegeben. Zum einem ist die resultierende Transformation des Objektes abhängig von mehreren Matrizen im Szenengraph und muss bei Bedarf jedes Mal neu berechnet werden. Ein größeres Problem ist aber die Uneindeutigkeit, wenn die globale Transformation direkt gesetzt werden soll. Hier muss bestimmt werden, welche der übergeordneten Transformationsknoten dafür angepasst werden soll, oder ob die Transformationsanteile sogar auf mehrere Knoten aufgeteilt werden müssen. Auch muss die Transformationsmatrix in das Koordinatensystem des entsprechenden Transformationsknotens überführt werden.

Diese Entscheidung, welche der übergeordneten Matrizen angepasst werden soll, kann im Allgemeinen nur mit Hilfe einer Betrachtung des gewünschten Verhaltens der Simulation getroffen werden. Ein Beispiel für eine komplexe Aufteilung der globalen Transformation bei verbundenen Bauteilen mit Hilfe von vordefinierten Constraints und die Transformation in das entsprechende Koordinatensystem der übergeordneten Transformationsknoten mittels einer geeigneten Constraint-Propagierung wird in Abschnitt 5.3.1 gegeben.

Auch können Szenengraphtools schon Transformationsknoten mit komplexen Abhängigkeiten ihrer Transformation bereitstellen. Ein Beispiel hierfür sind so genannte *Billboard-Knoten*. Diese Knoten richten die Orientierung einer vorgegebenen Achse immer in Richtung des Betrachters aus. Sie werden unter anderem dazu benutzt, texturierte Polygone so zum Betrachter auszurichten, dass die Blickrichtung immer senkrecht auf die Fläche trifft und damit das flache Polygon als Ganzes zu sehen ist.

Eine Transformationsmatrix in homogenen Koordinaten hat 4 Zeilen und Spalten und somit theoretisch 16 Freiheitsgrade. Hiervon werden für Festkörper-Transformationen von Objekten im virtuellen Raum nur 6 bzw. bei zusätzlicher Betrachtung von inhomogenen Skalierungen insgesamt 9 benutzt. Diese lassen sich für Transformationen von Objekten in drei Klassen unterteilen: Translation, Rotation und Skalierung. Die Vorgehensweise bei der Einschränkung von Transformationen ist die Aufteilung der Freiheitsgrade der 4x4-Matrix in die drei Klassen, um diese einzelnen Werte dann durch Constraints einzuschränken. Weitere Transformationen – wie z.B. Scherungen oder Projektionen, welche auch durch diese Art Matrizen beschrie-

ben werden können – werden hier nicht betrachtet, da sie für Operationen bei der Platzierung oder Anpassung von Bauteilen in der Regel keine Rolle spielen.

Auf jede der drei Klassen entfallen drei Freiheitsgrade. Um die Eigenschaften der funktionalen Constraints für diese Freiheitsgrade aufrechterhalten zu können, muss eine eindeutige funktionale Abbildung zwischen der Matrix und der Transformationsbeschreibung bestehen. Während man die Freiheitsgrade von Translation bzw. Skalierung – wie im Folgenden gezeigt – anhand der Raumachsen eindeutig auf drei skalare Werte abbilden kann, ist die Festlegung der Freiheitsgrade für die Rotation problematischer. Da Rotation und Skalierung zum Teil auf die gleichen Werte in der Matrix zugreifen, ist auch die Extraktion der einzelnen Transformationsarten aus einer 4x4-Matrix genau zu betrachten.

3.2.1 Translation

Am einfachsten ist die Zuordnung der Transformationsmatrix zu den drei Werten der Freiheitsgrade bei der Translation. Hier stehen die drei Komponenten des Translationsvektors $\vec{T} = (t_x, t_y, t_z)$ direkt an der entsprechenden Stelle in der Transformationsmatrix. Es kann somit eine einfache Abbildung vom Translationsvektor zur entsprechenden Matrix angegeben werden:

$$\mathit{TransMatrix}: \mathrm{V} \rightarrow \mathrm{M}$$

$$\mathit{TransMatrix}((t_x, t_y, t_z)) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

Da Rotationen und Skalierungen mit Zentrum im Nullpunkt nur die 3x3 Submatrix der jeweils ersten drei Komponenten der Zeile bzw. Spalte betreffen, kann man bei der Translation die Werte in der Matrix direkt ablesen und setzen. Damit ergeben sich ohne Umwege aus der Transformationsmatrix M die drei skalaren Werte für den Translationsvektor $\vec{T}$, mittels der folgenden Funktion:

$$\mathit{TransVektor}: \mathrm{M} \rightarrow \mathrm{V}$$

$$\mathit{TransVektor}\left(\begin{bmatrix} \mathrm{m}_{0,0} & \mathrm{m}_{0,1} & \mathrm{m}_{0,2} & \mathrm{m}_{0,3} \\ \mathrm{m}_{1,0} & \mathrm{m}_{1,1} & \mathrm{m}_{1,2} & \mathrm{m}_{1,3} \\ \mathrm{m}_{2,0} & \mathrm{m}_{2,1} & \mathrm{m}_{2,2} & \mathrm{m}_{2,3} \\ \mathrm{m}_{3,0} & \mathrm{m}_{3,1} & \mathrm{m}_{3,2} & \mathrm{m}_{3,3} \end{bmatrix}\right) = (\mathrm{m}_{3,0}, \mathrm{m}_{3,1}, \mathrm{m}_{3,2})$$

Dieser Translationsvektor kann dann durch einfache Vektor-Constraints eingeschränkt werden. Ein Beispiel ist die Einschränkung auf lineare Bewegungen.

Dabei gilt dann:

$$\text{LinearTrans} : \Re \times \text{V} \rightarrow \text{V}$$

$$\text{LinearTrans}(\text{f}, \vec{\text{d}}) = \text{f} \cdot \vec{\text{d}}$$

wobei

$\vec{\text{d}} \in \text{V}$ Ein auf die Länge 1 normierter Richtungsvektor

$\text{f} \in \Re$ Der Faktor für die Länge der Translation

Damit können die drei Freiheitsgrade durch die vorgegebene Richtung der Translation auf einen Freiheitsgrad reduziert werden, der die Weite der Translation in diese Richtung beschreibt.

3.2.2 Skalierung

Bei der Skalierung von Zentrum aus entlang der drei Hauptachsen des Koordinatensystems stehen die Komponenten des Skalierungsvektors $\vec{S} = (s_x, s_y, s_z)$ auf der Diagonale der 3x3-Submatrix.

ScaleMatrix: V→M

$$ScaleMatrix((s_x, s_y, s_z)) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Bei der Berechnung des Skalierungsvektors aus einer Transformationsmatrix ist zu beachten, dass die Skalierung durch eine eventuelle Rotation in der oberen 3x3-Matrix überlagert sein kann. Um aus so einer kombinierten Transformation die Skalierung zu extrahieren kann man über die Längen der Basisvektoren in der 3x3-Matrix gehen:

ScaleVektor: M → V

$$ScaleVektor(\begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} \\ m_{1,0} & m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,0} & m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,0} & m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix})$$

$$= (\|(m_{0,0}, m_{0,1}, m_{0,2})\|, \|(m_{1,0}, m_{1,1}, m_{1,2})\|, \|(m_{2,0}, m_{2,1}, m_{2,2})\|)$$

Eine Skalierung vom Nullpunkt des Koordinatensystems aus lässt sich also gut durch die Angabe der drei Werte auf der Diagonalen parametrisieren, bzw. bei kombinierten Transformationen durch die Länge der Basisvektoren gewinnen. Somit werden diese Werte des Skalierungsvektors auch für die den Constraints zu unterziehenden Freiheitsgrade benutzt.

Ähnlich der linearen Translation kann man auch eine Skalierung mit einem Freiheitsgrad – dem Faktor f – für eine Skalierung entlang eines beliebigen (normierten) Richtungsvektors $\vec{d}$ angeben. Der zugehörige Skalierungsvektor errechnet sich dann wie folgt:

$$\text{LinearScale} : \Re \times V \rightarrow V$$

$$\text{LinearScale}(f, \vec{d}) = (1 + (f-1) \cdot d_x, 1 + (f-1) \cdot d_y, 1 + (f-1) \cdot d_z)$$

wobei

$\vec{d} = (d_x, d_y, d_z) \in V$ Ein auf die Länge 1 normierter Richtungsvektor

$f \in \Re$ Der Faktor für die Skalierung

Die Abbildung einer beliebigen Skalierung vom Nullpunkt des Koordinatensystems lässt sich also komfortabel mit der Angabe des Skalierungsvektors $\vec{S}$ beschreiben. Seine Komponenten bilden bei der Skalierung die drei Freiheitsgrade ab, welche durch funktionale Constraints eingeschränkt werden können. Durch die „LinearScale"-Funktion kann bei vorgegebener Richtung eine Einschränkung auf einen Freiheitsgrad vorgenommen werden, der den Faktor der Skalierung enthält.

Die Skalierung von einem beliebigen Skalierungszentrum aus kann zu der Translation des Zentrums in den Ursprung des Koordinatensystems, gefolgt von der Skalierung und der inversen Translation zerlegt werden. Die Kombination resultiert in einer Skalierung mit einer von der Skalierung und Skalierungszentrum abhängigen Translation. Die durch die drei Komponenten des Skalierungszentrums gegebenen Parameter dieser Art der Skalierung werden bei der Betrachtung von Skalierungsconstraints als konstant angenommen. Die eindeutige Zerlegung einer kombinierten Transformation in die Skalierungsfaktoren und einer Translation kann bei einer solchen Skalierung nur bei einem bekanntem Skalierungszentrum geschehen.

3.2.3 Rotation

Beliebige Rotationen um den Nullpunkt des Koordinatensystems sind leider nicht so einfach auf ihre drei Freiheitsgrade abzubilden.

Die einzige Beschreibung, die Rotationen direkt auf drei skalare Werte abbildet, ist die Angabe über *Euler-Winkel.* Diese sind aber durch eine nicht eindeutige Abbildung auf die entsprechenden Werte und Probleme bei der Verrechnung von mehreren Rotationen nicht einfach zu handhaben. Durch diese Beschreibung lassen sich aber Transformationen z.B. einfach auf Rotationen um eine der drei festen Raumachsen einschränken. Aufgrund der Problematik der Mehrdeutigkeit der Parameter ist z.B. die Einschränkung auf die Drehung um eine beliebige Achse oder die Einschränkung mehrer Parameter auf definierte Winkel nur schwer zu realisieren.

Quaternionen sind eine Verallgemeinerung der komplexen Zahlen. Sie bieten eine eindeutige Repräsentation von Rotationen. Jedes Quaternion ist durch vier reelle Komponenten (*w, x, y, z*) eindeutig bestimmt. Sie werden über Linearkombination mit den vier Basiselementen des Vektorraums der Quaternionen (*1, i, j, k*) kombiniert zu:

$$q = w + i \cdot x + j \cdot y + k \cdot z$$

Sie sind zwar mathematisch sehr gut handhabbar (Interpolation, Eindeutigkeit), sind aber nicht intuitiv zugänglich. Quaternionen bestehen aus vier Werten (bei drei Freiheitsgraden), welche untereinander abhängig und nicht einzeln interpretierbar

sind. Daher lassen Quaternionen direkt keine Aufteilung der Rotation in Freiheitsgrade zu, welche sinnvoll einzeln durch Constraints eingeschränkt werden können.

Die Darstellung der Kombination aus *Drehwinkel und Rotationsachse* ermöglicht auch keine gut handhabbare Abbildung auf die drei Freiheitsgrade. Bei einem Freiheitsgrad in der Winkelangabe verbleiben die restlichen zwei für den dreikomponentigen Richtungsvektor der Rotationsachse. Richtungsvektoren können ohne Informationsverlust auf eine beliebige Länge normiert werden. Dadurch sind die drei Komponenten untereinander abhängig und ermöglichen somit nur zwei statt drei Freiheitsgrade. Der Vorteil dieser Darstellung liegt aber in der Möglichkeit, die Drehachse als einen Parameter entweder als konstant vorzugeben oder mit Constraints speziell für Richtungsvektoren zu belegen. Für den Drehwinkel kann dann als variabler Freiheitsgrad ein eigener Constraint definiert werden.

Daher wird im vorliegenden Ansatz die Rotation zweistufig behandelt. Die Abbildung der Matrix auf ihre rotatorischen Freiheitsgrade erfolgt über ein Quaternion $q = w + i \cdot x + j \cdot y + k \cdot z$ das normalisiert sein muss, d.h.:

$$\sqrt{w^2 + i^2 + j^2 + k^2} = 1$$

$$RotationsMatrix : \mathrm{Q} \rightarrow \mathrm{M}$$

$$RotationsMatrix(w + i \cdot x + j \cdot y + k \cdot z) =$$

$$\begin{bmatrix} 1-2(y^2+z^2) & 2\cdot(x\cdot y - z\cdot w) & 2\cdot(x\cdot z + y\cdot w) & 0 \\ 2\cdot(x\cdot y + z\cdot w) & 1-2(y^2+z^2) & 2\cdot(y\cdot z - x\cdot w) & 0 \\ 2\cdot(x\cdot z - y\cdot w) & 2\cdot(y\cdot z + x\cdot w) & 1-2(y^2+z^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

und entsprechend:

$$RotationsQuat : \mathrm{M} \rightarrow \mathrm{Q}$$

$$RotationsQuat\begin{pmatrix} m_{0,0} & m_{0,1} & m_{0,2} & 0 \\ m_{1,0} & m_{1,1} & m_{1,2} & 0 \\ m_{2,0} & m_{2,1} & m_{2,2} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} =$$

$$(w + i\frac{m_{2,1} - m_{1,2}}{4w} + j\frac{m_{0,2} - m_{2,0}}{4w} + k\frac{m_{1,0} - m_{0,1}}{4w});$$

$$\text{wobei} : w = \frac{\sqrt{1 + m_{0,0} + m_{1,1} + m_{2,2}}}{2}$$

Im zweiten Schritt werden die berechneten Quaternionen einem Constraint mit Hilfe der Darstellung über Drehwinkel und Rotationsachse unterworfen. Dafür braucht man folgende Abbildungen zwischen Quaternionen und dieser Darstellung:

$$RotationsAchse : \mathrm{Q} \rightarrow \mathrm{V}$$

$$RotationsAchse(w + i \cdot x + j \cdot y + k \cdot z) = \begin{pmatrix} \frac{x}{\sqrt{1-w^2}} \\ \frac{y}{\sqrt{1-w^2}} \\ \frac{z}{\sqrt{1-w^2}} \end{pmatrix}$$

$$RotationsWinkel : \mathrm{Q} \rightarrow \Re$$

$$RotationsWinkel(w + i \cdot x + j \cdot y + k \cdot z) = 2 \cdot \mathrm{acos}(w)$$

und entsprechend:

$$RotationsQuat : \mathrm{V} \times \Re \rightarrow Q$$

$$RotationsQuat((\mathrm{x}, \mathrm{y}, \mathrm{z}), \alpha)$$

$$= \cos\frac{\alpha}{2} + i \cdot \mathrm{x} \cdot \sin(\frac{\alpha}{2}) + j \cdot \mathrm{y} \cdot \sin(\frac{\alpha}{2}) + k \cdot \mathrm{z} \cdot \sin(\frac{\alpha}{2})\ ;$$

wobei der Vektor (x, y, z) normiert sein muss, d.h. $\mathrm{x}^2 + \mathrm{y}^2 + \mathrm{z}^2 = 1$

Rotationen um einen beliebigen Aufsatzpunkt werden durch ein zusätzliches *Rotationszentrum* angegeben. Ähnlich wie bei der Skalierung, kann dieses durch die Translation vom Zentrum in den Ursprung, Drehung um den Ursprung und der inversen Translation beschrieben werden. Entsprechend ist auch das Rotationszentrum für die Einschränkung mittels Constraints als konstanter Parameter anzusehen.

3.2.4 Kombinierte Transformationen

Um aus einer beliebigen aus Rotation, Skalierung und Translation zusammengesetzten Matrix die einzelnen Komponenten zu errechnen, kann man inkrementell vorgehen: Nachdem man die Rotationskomponente errechnet hat und mit Hilfe des Rotationszentrums die durch diese Rotation verursachte Translation bestimmt hat, können diese Anteile aus der ursprünglichen Matrix herausgerechnet werden. Verfährt man mit der Skalierung entsprechend, bleibt von der ursprünglichen Matrix nur noch der dann einfach zu bestimmende Anteil der Translation übrig.

Ein interessanter Aspekt bei der Kombination von Translation und Rotation ergibt sich bei paralleler Auswertung von dem durch die Quaternion vorgegebene Rotation und der Translation bei einem asymmetrischen Skalierungszentrum. Das standardmäßige Vorgehen wäre zuerst die Rotation der überwachten Matrix zu berechnen und einzuschränken und Translation entsprechend dem Rotationszentrum anzugleichen. In dem Fall würde eine reine Translation ohne Rotation durch den Constraint komplett unterbunden. Wird jedoch die versuchte Translation durch die asymmetrische Rotation zuerst möglichst gut angenähert, resultiert sie in einer Rotation, welche das Zentrum des Objektes in Richtung der Translation verlagert. Abbildung 28 zeigt die beiden Lösungen für den Standardfall a) und der Anpassung über die Translation b).

Mit dieser Art der Constraintbehandlung können sich sehr intuitive Anpassungen der Konfigurationen bei Interaktion mit den Bauteilen ergeben (siehe auch Abschnitt 6.5).

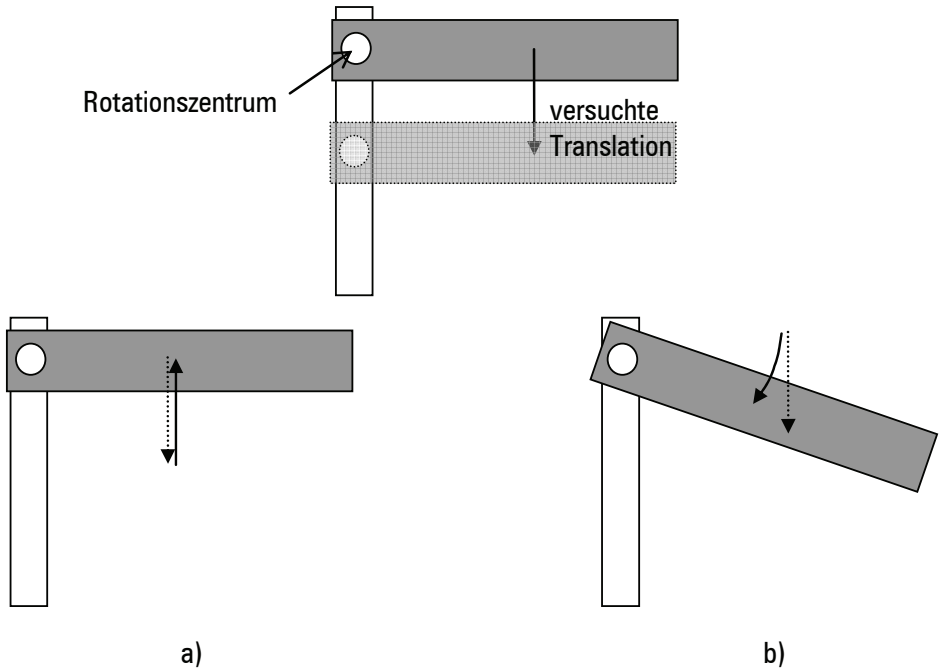


Abbildung 28: Unterschiedliche Lösungen eines Rotationsconstraints bei einer reinen Translation

3.2.5 Verbindungsconstraints

Bei einer nicht festen Verbindung von zwei Bauteilen können verschiedene rotatorische und translatorische Freiheitsgrade frei bleiben, welche auch miteinander gekoppelt sein können. Eine Übersicht über die möglichen Belegungen der Freiheitsgrade geben die *kinematischen Paare*.

Einfache kinematische Paare

Kinematische Paare (Turner *et al.*, 1992) sind in Klassen eingeteilt, welche der Anzahl der Freiheitsgrade der Relativbewegung der verbundenen Bauteile entsprechen. Bei der einfachen Verbindung von zwei Bauteilen treten nicht alle theoretisch möglichen Kombinationen von Freiheitsgraden auf. Insbesondere ist mindestens ein translatorischer Freiheitsgrad eingeschränkt, da sonst keine Verbindung der Bauteile aufrechterhalten würde.

Stehen keine Freiheitsgrade zur Verfügung, ist die Verbindung der Bauteile fest, d.h. es ist keine Relativbewegung möglich. Ohne Freiheitsgrade in der Rotation gibt es nur den Fall eines einzelnen translatorischen Freiheitsgrades. Dieses bezeichnet man als *Prismatic-Pair (P)*. Ein einzelner FG in der Rotation ergibt ein *Revolute-Pair (R)*. Kommt ein translatorischer FG hinzu ergibt sich bei paralleler Ausrichtung der Translationsachse und der Rotationsachse ein *Cylindrical-Pair (C)*. Sind diese beiden FG gekoppelt ergibt sich eine Schraubverbindung, das *Helical-Pair (H)*, stehen die Achsen der beiden FG hingegen senkrecht zueinander spricht man von einem *Cam-Pair (Ca)*. Zusammen mit einem weiteren translatorischen FG kommt es zu dem Fall zweier frei miteinander verbundener Flächen: einem *Plane-Pair (Pl)*. Von den anderen höheren Graden der kinematischen Paare wird im Folgenden nur noch das

Spheric-Pair (S) betrachtet, bei dem alle drei Rotationen, aber keine Translation freigegeben ist.

Tabelle 2 zeigt eine Übersicht der kinematischen Paare gelistet nach der Anzahl der rotatorischen, bzw. translatorischen Freiheitsgrade.

Translatorische FG Rotatorische FG	0	1	2	3
0	Fest	P	-	-
1	R	C,Ca	Pl	-
2	Sl	Ss	Cp	-
3	S	Sg	Sp	-

Tabelle 2: Übersicht der Kinematischen Paare nach ihren translatorischen und rotatorischen Freiheitsgraden

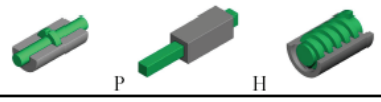
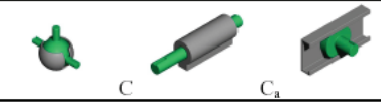
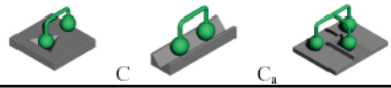
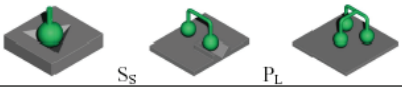
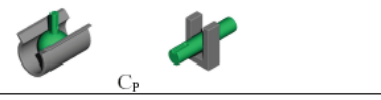
Class	Degrees of Freedom	Name and Symbol	Diagram
I	1	Revolute – R Prism – P Helical – H	 Form-Closed  Force-Closed
II	2	Slotted Spheric - S_L Cylinder – C Cam - C_a	 Form-Closed  Force-Closed
III	3	Spheric Pair – S Sphere Slotted Cylinder - S_S Plane Pair - P_L	 Form-Closed  Force-Closed
IV	4	Sphere Groove - S_g Cylinder Plane Pair - C_P	 Form-Closed  Force-Closed
V	5	Sphere Plane - S_P	 Form-Closed  Force-Closed

Abbildung 29: Graphische Übersicht der einfachen Kinematischen Paare – aus (Wharton & Singh, 2001)

Eine komplette Übersicht der einfachen kinematischen Paare ist in Abbildung 29 zu sehen. Zusätzlich sind in der rechten Spalte der Abbildung Visualisierungen von Bauteilen zu sehen, die diese kinematischen Paare bilden. Die Diagramme zeigen jeweils ein Beispiel der kinematischen Paare für den formschlüssigen Fall, wobei die Relativbewegungen komplett durch die Form der Bauteile vorgegeben sind, und den kraftschlüssigen Fall, wo die Verbindung durch eine Kraft der Bauteile zueinander aufrechterhalten und durch die Form eingeschränkt wird.

Ports

Als abstrakte Formalisierung einer Verbindungsstelle von Bauteilen wurde von Wachsmuth (Wachsmuth & Jung, 1996) das Konzept der *Verbindungs-Ports* entwickelt. Ports können Verbindungsstellen und ihre Eigenschaften, wie die möglichen Freiheitsgrade der Verbindung, die möglichen Verbindungsarten und auch die Geometrie für diese Verbindungsstelle definieren. Eine Ontologie von Porttypen wurde in (Kopp, 1998) ausgearbeitet (siehe Abbildung 30). Sie bildet die verschiedenen Verbindungsstellen von Bauteilen ab.

Bauteile können durch diese Ports formal mit Verbindungsstellen angereichert werden. Hierbei sind die Verbindungsstellen in der Regel durch die Geometrie der Bauteile vorgegeben. Es können aber auch Ports definiert werden, die keine direkte Entsprechung mit der Form der Objekte haben. Für die teilautomatische Generierung von Ports wurde in (Biermann, 2001; Biermann et al., 2002) ein System vorgestellt, das die Geometrie von Bauteilen weitgehend automatisch analysiert und mögliche Ports des Bauteils vorschlägt.

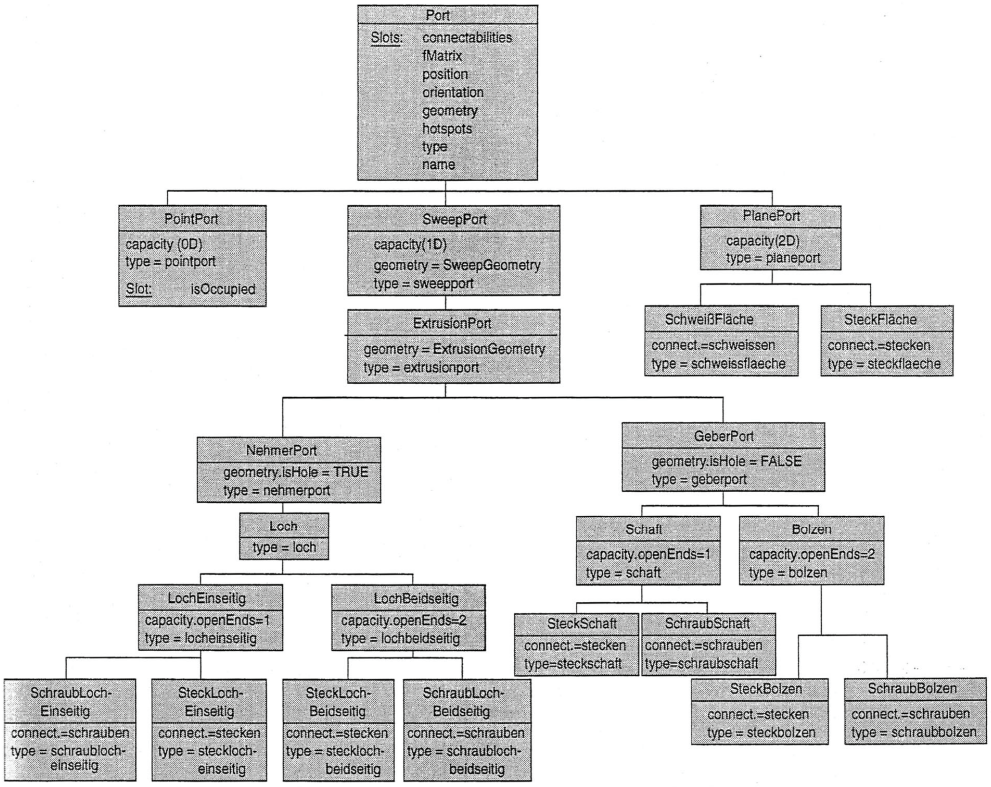


Abbildung 30: Taxonomie der Porttypen – aus (Kopp, 1998)

Die erlaubten Relativbewegungen – Translationen bzw. Rotationen – von verbundenen Ports werden in Form von Freiheitsmatrizen angegeben.

Erweiterte Freiheitsmatrizen

Die *erweiterten Freiheitsmatrizen* basieren auf den logischen Freiheitsmatrizen von Roth (Roth, 1994). Diese definieren für jede der drei Raumachsen die erlaubte Translation bzw. Rotation für die jeweilige Achse in positiver und negativer Richtung. Die Erweiterung von (Kopp, 1998) sieht statt der qualitativen booleschen Informationen für die erlaubten Bewegungen numerische Werte vor, welche quantitativ die Werte der Translation bzw. Rotation eingrenzen. Die Matrix besteht somit aus 3 mal 4 also insgesamt 12 Werten: Jeweils die positiven und negativen Maximalwerte für Translation und Rotation in einer Zeile mit insgesamt 3 Zeilen für jede Achse.

Erweiterte Freiheitsmatrizen bestehen aus einer 3x4-Matrix A und einer Menge von Kopplungen C:

$$M_f := (A, C)$$

wobei:

- $A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}, a_{ij} \in \Re^+ \cup \{\text{FREE}, \text{BLOCKED}\}$
- $C = \{c_1, c_2, ...c_n\}, n \geq 0$

  wobei:

  - $c = (< i_1, j_1 >, < i_2, j_2 >, r)$

    wobei: $i_1, i_2 \in \{1,2,3\}$, $j_1, j_2 \in \{1,2,3,4\}$, $< i_1, j_1 > \neq < i_2, j_2 >, r \in \Re$

  Jeweils eine lineare Kopplung zwischen den Parametern der Matrix mit dem Faktor r

In den Zeilen der Matrix stehen zuerst die Maximal- und Minimalwerte der Translation, dann die der Rotation. Die erste Reihe enthält die Werte für die X-Achse, die folgenden für die Y-Achse und die letzte die Werte für die Z-Achse:

$$\begin{pmatrix} T(+x) & T(-x) & R(+x) & R(-x) \\ T(+y) & T(-y) & R(+y) & R(-y) \\ T(+z) & T(-z) & R(+z) & R(-z) \end{pmatrix}$$

Durch erweiterte Freiheitsmatrizen lassen sich viele der im vorherigen Abschnitt aufgeführten kinematischen Paare modellieren, indem bei den Stellen der Freiheitsmatrizen, welche den Freiheitsgraden der Rotationen bzw. Translationen der einzelnen Achsen für den positiven und negativen Fall der Wert „FREE“ eingetragen wird, während die anderen Parameter ohne Freiheiten mit dem Wert „BLOCKED“ belegt werden.

Durch die zusätzliche Angabe von Kopplungen der Parameter können einfache, lineare Abhängigkeiten zwischen den Parametern der Matrix definiert werden. Für

das Helical-Pair besteht z.B. eine Abhängigkeit zwischen der Translation und der Rotation, welche durch jeweils eine Kopplung der positiven bzw. negativen Werte der Translation und der Rotation in der Matrix realisiert wird.

Probleme ergeben sich z.B. bei kinematischen Paaren mit zwei rotatorischen Freiheitsgraden. Zum einen führt bei Rotationen eine Vertauschung der Reihenfolge der Rotationsachsen, für welche die Rotation freigegeben wird, bei quantitativen Einschränkungen zu unterschiedlichen Ergebnissen, wobei bei den erweiterten FM keine Reihenfolge der Rotationsachsen vorgegeben werden kann. Zudem kann nach einer Rotation um eine Achse für die inkrementelle Beschreibung durch die FM eine Rotation um die durch die Drehung verlagerte Rotationsachse, welche dann frei im Raum liegen kann und nicht mehr parallel zu einer Raumachse ist, durch Freiheitsmatrizen nicht mehr beschrieben werden. Auf die Probleme der eindeutigen Aufteilung von beliebigen Rotationen auf ihre drei Freiheitsgrade wurde schon in Abschnitt 3.2.3 eingegangen.

Somit lässt sich durch die erweiterten Freiheitsmatrizen die Modellierung eines Großteils der kinematischen Paare in kompakter, deklarativer Form beschreiben. Sie können in einzelnen Fällen bei kombinierten Rotationen aber unzureichend sein.

3.2.6 Abstrakte Geometrische Constraints

Die direkt durch die Verbindung von Bauteilen aufgebauten Constraints bestimmen die Relativbewegungen der Bauteile untereinander. Zusätzliche, abstrakte geometrische Constraints können darüber hinaus die Einhaltung globaler Vorgaben überwachen, oder die Bewegung von Bauteilen während der Interaktion steuern.

Planungs-Constraints

Über Planungs-Constraints ist es möglich, einen groben Plan des zu erschaffenen Aggregates zu erstellen, indem der Konstrukteur abstrakte Vorgaben für dieses Aggregat vorgeben kann.

Soll z.B. der Radstand eines komplett montierten Autos auf eine bestimmte Länge eingeschränkt werden, ohne dass dieses schon durch konkrete Bauteile wie Fahrwerk, Achsen, usw. festgelegt ist, kann ein abstrakter Constraint einen definierten Abstand zwischen den Rädern festlegen.

Diese abstrakten Constraints haben aber eine geringere Priorität (zu Constraint-Prioritäten siehe auch Abschnitt 3.1.3) als die Constraints, welche durch die Verbindungen und Bauteileigenschaften definiert sind. Daher können die abstrakten Constraints grobe Vorgaben für die relative Transformation zweier Bauteile machen, werden aber durch eine konkrete Bauteilverbindung eventuell außer Kraft gesetzt.

Falls die Transformationen von Bauteilen durch eine konkrete Bauteilverbindung nicht mit dem abstrakten Constraints vereinbar sind, wird dieser Constraint überschrieben. Abstrakte Constraints können aber weiterhin für den Konstruktionsprozess sinnvoll sein, da sie Auskunft über Konflikte mit der groben Planung des Aggregates geben. Um eine Lösung der durch die abstrakten Constraints aufgeworfenen Konflikte zu finden, können aus der Differenz zwischen den aktuellen und den definiteren Transformationen qualitative und quantitative Merkmale für ein eventuelles Re-Design des gebauten Aggregates gewonnen werden.

Interaktions-Constraints

Während der Interaktion können auch geometrische Constraints aufgebaut werden, um durch eine Kopplung z.B. die Position von Bauteilen mittels einer Positionsänderung eines getrackten Manipulators anzupassen.

Eine Art dieser Constraints setzt die globalen Positionen zweier virtueller Objekte in Relation. Für die Interaktion sind diese Constraints in der Regel gerichtet vom Manipulator zum manipulierten Objekt. Sie können aber durch andere Constraints eingeschränkt werden. Diese Constraints binden z.B. die Bewegung eines Objektes an die Bewegung der Hand des Benutzers für eine direkte „Drag and Drop“ Manipulation. Dabei sind unterschiedliche Formen der Bindung der Bewegung zwischen zwei virtuellen Objekten möglich:

Direkte Kopplung

Rotation und Translation werden zusammen direkt im Koordinatensystem des Manipulators an das manipulierte Objekt weitergegeben, d.h. der Mittelpunkt der Rotation liegt in dem Manipulator. Dieses kommt vor allem für die Manipulation von Objekten im Greifraum zum Einsatz, da hier der Eindruck entsteht, man hätte das Objekt in der Hand. Hierbei wird direkt die Differenz der globalen Transformation der Hand and die globale Transformation des Objektes propagiert. Da hierfür aber ausgewählt werden muss, welche Matrizen in den übergeordneten Transformationsknoten des Objekts im Szenengraph verändert werden müssen, wird wie bei den Portverbindungen aus Abschnitt 5.3.4 die Transformation mittels weiterer Constraints auf diese Matrizen aufgeteilt.

Translation und Rotation getrennt

Translation und Rotation der Bewegung werden getrennt behandelt. Während die Translation wie im ersten Fall direkt an die globale Transformation des Objekts weitergegeben wird, wird der Rotationsanteil umgesetzt in eine Rotation um den Mittelpunkt des manipulierten Objektes. Dieser Fall ist besser für die Manipulation entfernter Objekte geeignet, da bei der direkten Kopplung die Rotation des Manipulators durch die große Entfernung des Rotationszentrums und den entsprechend großen Hebel sehr starke Bewegungen des Objektes zur Folge hat.

Nur Translation

Die Rotation kann während der Manipulation auch gesperrt werden, falls für definierte Bewegungen von Objekten keine Veränderung der Ausrichtung durch die Interaktion gewünscht wird. Weitere Constraints während der Interaktion können zusätzlich die Translation der Objekte auf eine Ebene oder sogar auf eine lineare Bewegung einschränken.

Komplexere Verknüpfungen können z.B. den Abstand der Hände an die Skalierungsparameter eines Objektes binden. Bei dieser Art von Constraints können dann auch, wie in Kapitel 5 gezeigt, mehrere Rechenknoten aus der visuellen Programmierung beteiligt sein. Beispiele für solche komplexen Constraints während der Interaktion sind in Abschnitt 6.5.1 zu finden.

3.2.7 Nicht-geometrische Constraints

Der Haupteinsatzzweck von Constraints ist im Szenario der Virtuellen Konstruktion die Überprüfung und Propagierung geometrischer Constraints. Da aber durch Constraints im Allgemeinen beliebige Parameter überwacht und miteinander in Beziehung gesetzt werden können, sind natürlich auch Einschränkungen und Kopplungen außerhalb der geometrischen Beziehungen der Objekte möglich. Diese Kopplungen können z.B. einfache Material oder Beleuchtungsparameter – wie z.B. Helligkeit oder Farbwerte – betreffen, oder auch komplexe Parameter, welche die abstrakten Eigenschaften von Objekten modellieren. Ein Beispiel für komplexe Parameter wären z.B. Strukturparameter einer Objekttextur (Körnigkeit, Winkel, Relief der Struktur, usw.), welche Auswirkungen auf das Erscheinungsbild des dazustellenden Materials haben.

Zum Beispiel ist so auf sehr einfache Weise ein Dimmer zu realisieren, welcher über den Drehwinkel eines Reglers oder der Position eines Schiebers in der virtuellen Szene die Helligkeit des Umgebungslichtes bestimmt.

3.2.8 Semantische Constraints

Neben den Constraints, welche durch die Beziehung und den Abgleich von Feldwerten ausgedrückt werden können, müssen bei der virtuellen Konstruktion auch Constraints berücksichtigt werden, die Einschränkungen auf einem höheren, semantischen Level beschreiben. Diese semantischen Constraints betreffen z.B. die Art der Bauteile, die Typen und Ausprägungen der Verbindungsstellen und den Zustand einer Interaktion. Die Informationen hierfür werden über ein semantisches Netz verwaltet und im Szenengraph über spezielle Schnittstellenknoten – den *Semantic-Entities* (siehe Abschnitt 6.1) – zur Verfügung gestellt. Über diese Knoten können auch Informationen über die Bauteile in das semantische Netz eingetragen werden. Daher werden bei dem Aufbau der Bauteile über das externe Datenformat (siehe Abschnitt 6.2) diese Semantic-Entities auch mit angelegt und mit den entsprechenden Informationen gefüllt. Diese Informationen werden zum einem zur Konfiguration der Constraint Mediatoren verwendet, zum anderen können mit ihnen komplexere semantische Abhängigkeiten überprüft werden. Ein Beispiel dafür ist die Verbindung von zwei Bauteilen. Hierbei müssen die Art der Verbindungsstellen auf ihre Kompatibilität überprüft werden. Dieses betrifft z.B. den Typ der Verbindungsstelle, aber auch sekundäre Eigenschaften, wie z.B. die Überprüfung des Radius eines Loches und des entsprechenden Bolzens. Diese semantischen Constraints von Bauteilverbindungen werden während des Aufbaus einer Verbindung kontrolliert, indem die Informationen aus den Semantic-Entities der beteiligten Verbindungsstellen gelesen und auf ihre Kompatibilität überprüft werden.

3.3 Geometrische Constraints in VR-Umgebungen

Bei der Realisierung einer eingeschränkten Bewegung zwischen Bauteilen in einer Virtuellen Umgebung werden zwei Hauptansätze verfolgt: Eine physikalisch-basierte Modellierung oder die Modellierung über Geometrische Constraints.

Bei der physikalisch-basierten Modellierung, wie zum Beispiel in (Zachmann, 1998; Mirtich & Canny, 1995; Baraff, 1994) werden physikalische Kräfte aufgrund von

erkannten Kollisionen zwischen den Teilen und die resultierenden Bewegungen der Teilstücke berechnet. Obwohl diese Art der physikalischen Dynamikberechnung für einfach geformte Objekte (wie z.B. kugelförmige Partikel) oder nur gelegentliche Kollisionen von Teilen mit angepassten Berechnungen und heutiger Hardware in Echtzeit möglich ist (Zachmann, 2000), ist dieser Ansatz für Objekte, welche über eine komplex geformte Oberfläche in einem ständigen Kontakt mit anderen, verbundenen Bauteilen stehen, nur für eine geringe Anzahl von Objekten geeignet (T. Fernando *et al.*, 2000). Der Vorteil dieser Methode ist der geringe Aufwand bei der Modellierung der Verbindungsarten, da sich sämtliche Constraints aus der Geometrie der Bauteile ergeben. Auf der anderen Seite überwiegt der Nachteil durch einen hohen Berechnungsaufwand und mögliche numerische Instabilitäten bei der Berechnung der Kräfte im Falle von mehreren Kontaktflächen. Ein Beispiel für eine für die physikalische Simulation unpassende Verbindung ist ein einfaches Scharniergelenk. Während eine Modellierung der erlaubten Relativbewegung der beteiligten Bauteile durch einen einfach zu berechnenden geometrischen Rotationsconstraint modelliert werden kann, wäre die Berechnung der erlaubten Bewegung durch die Kollision der Geometrien im Bereich der Scharnierverbindung aufgrund der komplexen Geometrie und der dadurch an mehreren Stellen gleichzeitig auftretenden Kollisionen aufwendig zu berechnen, oder führt, z.B. bei einer unsauberen oder zu groben Tesselierung, zu Problemen bei den numerischen Verfahren.

Bei der Modellierung über Geometrische Constraints werden die Bauteile durch Modellierung ihrer möglichen Bewegung exakt positioniert. Aktuelle Ansätze dieser Modellierung definieren entweder Gleichungen, welche die eingeschränkte Bewegung der Bauteile beschreiben, oder definieren Konstruktionsschritte, welche durch die Reduzierung der Freiheitsgrade die Relativbewegung zweier Verbindungspartner schrittweise einschränken. Für die Lösung der Gleichungssysteme aus dem ersten Fall existieren mehrere Ansätze. Neben rein numerischen Ansätzen, welche die Gleichungen über iterative Verfahren lösen (Lin *et al.*, 1981; Light & Gossard, 1982) existieren auch symbolverarbeitende Methoden. Der heutzutage am meisten benutzte Ansatz geht über den Aufbau von Constraintgraphen (Beispiele in (Sannella, 1993; Serrano & Gossard, 1992); siehe auch Abschnitt 3.1.3).

3.4 Zusammenfassung

Alle vorgestellten Ansätze werden über einen externen Prozess (sog. *Constraintsolvern*) realisiert, welcher einen Datenaustausch zwischen diesem Prozess der VR-Applikation notwendig macht und einen Abgleich der definierten Constraints und den Aktionen in der Virtuellen Umgebung erfordert. Zum anderen bilden die Szenengraphhierarchie und die Feldverbindungen in den Szenengraphtools (siehe Abschnitte 2.5.1 und 3.2) durch die Kopplung von Werten der Szenengraphknoten auch eine Form von Constraints, welche parallel zu den externen Constraints existieren. Daher muss auf den Einsatz von Feldverbindungen oder dem Aufbau von Szenengraphhierarchien bei externen Constraintsolvern für die von den Constraints beeinflussten Teile verzichtet werden, um Konflikte zwischen den internen und externen Constraints zu vermeiden. Die zweite Möglichkeit, die Abhängigkeiten, welche durch den Einsatz einer Szenengraphhierarchie und dem Aufbau von Feldverbindungen entstehen, auch im externen Constraintsolver nachzubilden, ist in der Regel wegen des aufwändigen und fehleranfälligen Abgleich der internen und externen Constraints nicht praktikabel.

Die in dieser Arbeit konzipierten Constraints sollen sich, im Gegensatz zu den externen Constraintsolvern, kompatibel in die Szenengraph- und Datenflussstrukturen aktueller VR-Tools einbinden lassen. Dafür werden diese lokalen Constraints direkt in Form von Programmknoten im Szenengraph platziert. So können die über Felder bereitgestellten Parameter der Knoten im Szenengraph direkt eingeschränkt und ihre Werte im Szenengraphen propagiert werden.

Diese Constraints sollen vor allem im Bereich der Virtuellen Konstruktion geometrische und Verbindungsconstraints modellieren, wobei sie auch Transformationen wenn nötig auf mehrere Knoten im Szenegraph aufteilen können. Dabei sollen die Constraints mächtig genug sein auch die Simulation von gekoppelten Getrieben und kinematischen Ketten realisieren zu können.

Die Einbettung lokaler Constraints in das Konzept der Programmknoten der Datenflussprogrammierung erlaubt, berechnete Daten aus dem Datenflussprogrammen mit in die Constraints einzubeziehen. Für die Definition der Constraint-Knoten kann dadurch auch – mit geringen Erweiterungen – die Beschreibungssprache der Programmknoten übernommen werden.

Ein lokales, über das Eventsystem gesteuertes Constraint-Lösungssystem soll zusammen mit einer globalen Kontrolle, die auch semantische Informationen über die Objekte in der virtuellen Welt einbeziehen kann, die Behandlung von komplexen Zusammenhängen ermöglichen. Durch diese Art von Constraints werden die allgemeine Ablauflogik der Applikation, direkte Interaktionen der Benutzer und auch die Simulation von kinematischen Ketten modelliert, auch wenn Teile davon schon durch die Szenengraphhierarchie oder durch vorhandene Datenflusskonstrukte der VR-Tools realisiert sind.

4 VIPLIVE: Eine interaktive visuelle Datenfluss-Programmiersprache für immersive Virtuelle Umgebungen

In diesem Kapitel wird die selbst entwickelte visuelle Programmierumgebung für die Virtuelle Realität *VIPLIVE* (***V**isual **I**nteractive **P**rogramming **L**anguage for **I**mmersive **V**irtual **E**nvironments*) beschrieben. Neben der formalen Syntax der visuellen Programme wird auch das Ablaufschema der Recheneinheiten und die Abstraktionsschicht für den Einsatz in unterschiedlichen VR-Tools vorgestellt.

4.1 Einordnung des Systems

Das in dieser Arbeit vorgestellte System ist in den Grundzügen als DFVPL einzuordnen, weicht in einigen Methoden aber bewusst von der klassischen Datenflussprogrammierung ab.

Das Hauptprinzip des Systems entspricht dem einer *grobkörnigen datenflussorientierten visuellen Programmiersprache*. Sie baut einen gerichteten azyklischen Graphen auf, deren Rechenknoten nicht nur primitive Instruktionen darstellen, sondern auch komplexere Berechnungen anstellen können, die durch eine imperative Sprache definiert werden. Wie in Abschnitt 2.2.2 gezeigt hat sich diese Vorgehensweise als aktuell bester Ansatz für Datenflusssprachen herausgestellt. Auch die Datentypen des Datenflussprogramms werden in dieser externen Sprache angegeben. Die visuelle Sprache bildet hier in erster Linie eine Koordinierungssprache für den Ablauf der Programme.

Um die Berechnungen in den einzelnen Knoten definieren und die Bibliothek von Rechenknoten einfach erweitern zu können, wird die Ein- und Ausgabeschnittstelle der einzelnen Rechenknoten über eine speziell dafür entworfene *XML-basierte Beschreibungssprache* definiert, wobei die *Recheninstruktionen direkt in C++* angegeben werden. Die Datenschnittstelle der einzelnen Recheneinheiten ist über Felder realisiert, welche über *Feldverbindungen* – wie in datenflussorientierten VR-Tools und z.B. in der Beschreibungssprache VRML und X3D üblich – miteinander verknüpft sind. Bei der Realisierung der Berechnungsabläufe und der Wertpropagierung über Feldverbindungen können die vorhandenen Datenfluss-Funktionalitäten der eingesetzten VR-Tools flexibel übernommen oder erweitert und angebunden werden.

Um eine einheitliche Anbindung an das Szenengraphen-Interface verschiedener möglicher VR-Tools zu erreichen, wird für die Manipulationen im Szenengraphen und Datenflussgraphen und für Berechnungen mit VR-Typischen Datentypen (Vektoren, Matrizen, Quaternionen etc.) ein neu entwickeltes *universelles Szenengraph-API* eingesetzt, das speziell für die Kapselung der Basisoperationen bei der Interaktion mit Szenengraphstrukturen entwickelt wurde. Durch dieses API werden die Funktionen des konkret eingesetzten VR-Tools abstrahiert und eventuell fehlende Funktionalitäten implementiert. Stellt das eingesetzte VR-Tool kein ausreichendes Eventsystem für die Datenflussprogrammierung zur Verfügung kann dieses auch mit einer zusätzlichen Entwicklung – dem *Feld-Event-Layer* –zur Verfügung gestellt werden. Eventuell vorhandene Funktionalitäten wie Felder, Feldverbindungen oder Callbacks zur Berechnung werden von diesem Layer gekapselt.

Einzelne visuelle Programmabschnitte können hierarchisch über *Subgraphen (Knoten-Container)* zusammengefasst und mit einem eigenen Interface gekapselt werden und tragen so zur Übersichtlichkeit und der Wiederverwendbarkeit von Programmteilen bei.

Der Hauptunterschied zu den klassischen DFVPLs ist die Struktur des Datenstromes und der zeitliche Ablauf des visuellen Programms. In dem vorgestellten System werden die Rechenknoten mit einer *externen Prozessrate* evaluiert, was im Gegensatz zu der klassischen Datenflussprogrammierung steht, wo jeder Programmteil für sich zu Ausführung bereit ist, sobald ausreichend Eingabedaten vorhanden sind. Der externe Prozesstakt ist durch die Renderschleife der VR-Applikation gegeben[7], um die Daten möglichst exakt zu der Berechnung des nächsten Renderframes bereit zu haben.

Um trotzdem die Verarbeitung einer von dem externen Prozesstakt *unabhängige Datenrate* zu ermöglichen, können pro Berechnungsschritt je nach Anforderung mehrere Datentoken verrechnet werden. Dazu werden die Datentoken, die während eines Frames aufgelaufen sind, für die Berechnung mit ihren zeitlichen Informationen in speziellen Datenstrukturen gespeichert.

Die Token im Datenfluss sind mit *Zeitstempeln* versehen, welche den Zeitpunkten der Datenaufnahme entsprechen. Diese Zeitstempel werden bei der Verarbeitung des Datenflusses berücksichtigt und geben bis hin zum Ergebnis der Berechnungen die exakte zeitliche Information der verarbeiteten Daten. Das Verfahren macht die zeitliche Exaktheit der Daten unabhängig von dem zeitlichen Ablauf des Datenflussprogramms und ermöglicht darüber hinaus Aussagen über die durchschnittliche *Latenzzeit* der berechneten Daten.

Verschiedene *Berechnungsmodi* erlauben hierbei eine unterschiedliche Gewichtung im Hinblick auf entweder niedrige Latenzzeiten oder eine möglichst präzise Berechnung der Daten. Um je nach Anforderung eine möglichst effiziente oder präzise Berechnung zu erlauben, können Datenrate und Berechnungsmodus in jedem Programmabschnitt unabhängig voneinander definiert werden.

Die Einbettung des visuellen Programmiersystems in eine *VR-Umgebung* ermöglicht die dreidimensionale animierte Darstellung der visuellen Programme und vielfältige Visualisierungsmöglichkeiten der Ergebnisse und Zwischenergebnisse in Echtzeit. Wie in Abschnitt 2.1.3 gezeigt, ist damit auch eine sehr komfortable und effiziente Fehlersuche in komplexeren visuellen Programmen möglich.

Durch das Interface in der Virtuellen Umgebung können sowohl verschiedene *animierte Visualisierungen* ausgewählt und während der Interaktion angezeigt werden, als auch bestehende Programme *während der Ausführung überprüft, verändert und erweitert* werden. Damit erfüllt das System alle Voraussetzungen für eine reaktive, visuelle Programmierumgebung. Um auch eine komfortable Möglichkeit zu bieten, über eine text-basierte Beschreibung die Programme zu erstellen, wird auch für die visuelle Programmstruktur ein XML-Format definiert.

---

[7] Bei einem Einsatz der Programmierumgebung außerhalb einer VR-Anwendung, kann der externe Takt auch durch andere Events bestimmt werden. In jedem Fall gibt der externe Takt – unabhängig von der Taktung der Eingabedaten – die Zeitpunkte vor, zu denen die jeweils möglichst aktuellen Daten berechnet werden sollen.

4.2 Hierarchischer Ansatz von Programmier-Ebenen

Das System verfügt über einen dreistufigen Ansatz von Programmier-Ebenen:

Gekapselte, lineare Abläufe können direkt in der imperativen Programmierung der Rechenknoten realisiert werden. Die freie Programmierung der Knoten über die Programmiersprache C++ zusammen mit dem speziell für Berechnungen von VR-typischen Datentypen und Szenengraphmanipulationen ausgelegten Interface-Layer USG (siehe Abschnitt 4.12) erlauben die Realisierung von komplexen Berechnungen und Manipulationen in einzelnen Knoten. Ein Hauptproblem der Datenflussprogrammierung – die schlechte Übersichtlichkeit durch sehr kleinschrittige Datenflussgraphen – wird dadurch vermieden. Durch die Kapselung über die XML-Beschreibungssprache können neue Rechencontainer sehr komfortabel dem System zugefügt werden.

Die Ebene der Datenflussprogrammierung dient in erster Linie als Koordinierungssprache (Gelernter & Carriero, 1992) der einzelnen Recheneinheiten. Diese ist in der Lage einen sehr intuitiven Zugang zu dem Ablauf des Programms zu geben. Während bei komplexen imperativen Programmen häufig Ablaufdiagramme mit Hilfe von UML-Tools erstellt werden, welche eine Übersicht der Aufrufreihenfolge einzelner Funktionen komfortabel darstellen, bildet der visualisierte Datenflussgraph selbst schon eine automatisch erstellte und editierbare Übersicht über den Programmablauf.

Besonders für Benutzer, die wenig Programmiererfahrung oder wenig Erfahrung mit der Verarbeitung vielschichtiger Sensordaten zur Erkennung und Interaktion in Virtuellen Umgebungen haben, kann diese abstrakte Sicht auf den Programmablauf hilfreich sein (Green & Petre, 1996). Durch eine große Anzahl vordefinierter Rechencontainer für den Einsatz zur Erkennung von Benutzereingaben und resultierenden Veränderungen in der virtuellen Szene, können auch unerfahrene Benutzer durch visuelle Programmierung einen relativ schnellen Zugang zu dem an sich sehr komplexen Bereich der Programmierung von Interaktionen in virtuellen Welten finden. Auch für erfahrene Programmierer erlaubt diese Schicht eine große Flexibilität bei der Veränderung und Erweiterung bestehender Programme, da diese direkt während der Ausführung überprüft und angepasst werden können.

Durch die dritte Ebene – der Zusammenfassung von Subgraphen in Knotencontainer mit eigenem Interface – können Teilabschnitte des visuellen Programms gekapselt werden. Die Container können aber jederzeit ihren gekapselten Subgraphen visualisieren, um z.B. Korrekturen an dem Unterprogramm vornehmen zu können. Dieses kann zu einer wesentlichen Verbesserung der visuellen Darstellung des Programms führen, da komplexe Subgraphen mit eventuell vielen Rechenknoten zu einer visuellen Einheit mit einer definierten Funktionalität zusammengefasst werden können. Damit reduziert diese Ebene die Anzahl der gleichzeitig präsentierten visuellen Primitive erheblich. Zusätzlich vereinfacht die Art der Kapselung von Subgraphen in Knoten mit eigenem Interface die Möglichkeit der Wiederverwendung von Programmteilen.

4.3 Definition der Syntax des visuellen Programms

Ein visuelles Programm wird als gerichteter Graph formalisiert.

Definition 4.1 (**Visuelles Programm**): Ein Visuelles Programm vp ist ein gerichteter Graph

$$vp:=(N,E)$$

wobei:

- $N = P \cup F$

  Menge von *Knoten* im Programmgraphen

  wobei:

  - P
    Eine Menge von *Programmknoten*
  - F
    Eine Menge von *Feldern*

- $E = E_{Assign} \cup E_{Connect}$

  Menge von *Kanten* zwischen den Knoten

  wobei:

  - $E_{Assign} \subseteq (F \times P) \cup (P \times F)$

    Eine Menge von Zuweisungen von Feldern zu Programmknoten

    Es gelte: $\forall f \in F; p,p' \in P; p \neq p' \quad (f,p) \in E \Rightarrow (f,p') \notin E$

    $\forall f \in F; p,p' \in P; p \neq p' \quad (p,f) \in E \Rightarrow (p',f) \notin E$

    Jedes Feld darf nur einem Programmknoten zugewiesen sein

  - $E_{Connect} \subseteq (F \times F)$

    Eine Menge von Feldverbindungen

    Es gelte: $\forall f^1, f^2, f^3 \in F; f^2 \neq f^3 \quad (f^2,f^1) \in E \Rightarrow (f^3,f^1) \notin E$

    Jedes Feld ist von maximal einem Feld ausgehend verbunden

Ein visuelles Programm besteht damit aus Programmknoten, denen über die Kanten des Graphs Felder zugeordnet sein können. Die Felder wiederum können durch Feldverbindungen miteinander verknüpft werden. Durch die Definition ist sichergestellt, dass jedes Feld nur einem Programmknoten zugeordnet ist und Feldverbindungen 1-zu-N-Verbindungen sind. Von einem Feld können also mehrere Feldverbindungen ausgehen, aber es darf nur eine ankommen. Dadurch realisieren Feldverbindungen die in Abschnitt 2.2.1 vorgestellten Replica Links, welche die Vervielfältigung von Feldwerten ermöglichen. Die Einschränkung auf ein eindeutiges Ausgabefeld bei den Feldverbindungen verhindert die Realisierung von Joint links, welche bei der Feldwertpropagierung ein undeterministisches Verhalten des Programms zur Folge

haben können, da der Wert des verbundenen Feldes aufgrund von Laufzeitunterschieden der einzelnen Programmknoten nicht mehr eindeutig bestimmt ist.

Definition 4.2 (**Feldarten**): Je nach Verbindung im Programmgraphen ergeben sich unterschiedliche Feldarten:

- $F_{In} = \{f \in F \mid \exists p \in P \ (f,p) \in E \wedge \forall p' \in P \ (p',f) \notin E\}$
  Eingabefelder von Programmknoten
- $F_{Out} = \{f \in F \mid \exists p \in P \ (p,f) \in E \wedge \forall p' \in P \ (f,p') \notin E\}$
  Ausgabefelder von Programmknoten
- $F_{Data} = F_{In} \cup F_{Out}$
  Datenfelder von Programmknoten fassen die Eingabefelder und Ausgabefelder zusammen
- $F_{IO} = \{f \in F \mid \exists p, p' \in P \ (f,p) \in E \wedge (p',f) \in E\}$
  Bidirektionale *Parameterfelder* von Programmknoten

Durch die Graphstruktur des visuellen Programms werden also unterschiedliche Feldfunktionen unterschieden. Eingabefelder liefern den Programmknoten Datentoken, welche in der Regel über die Feldverbindungen von Ausgabefeldern anderer Programmknoten propagiert werden. Die bidirektionalen Parameterfelder erlauben dem Programmknoten sowohl Lese- als auch Schreibzugriff auf die Feldwerte.

Die Unterteilung in Eingabefelder und Ausgabefelder erlaubt eine weitere Einschränkung der Feldverbindungen:

Definition 4.3 (**Gerichtete Feldverbindungen**):

Es gelte:

$$\forall f \in F_{In}, f' \in F_{Out} \ (f,f') \notin E$$

Feldverbindungen von Eingabefeldern zu Ausgabefeldern sind nicht erlaubt.

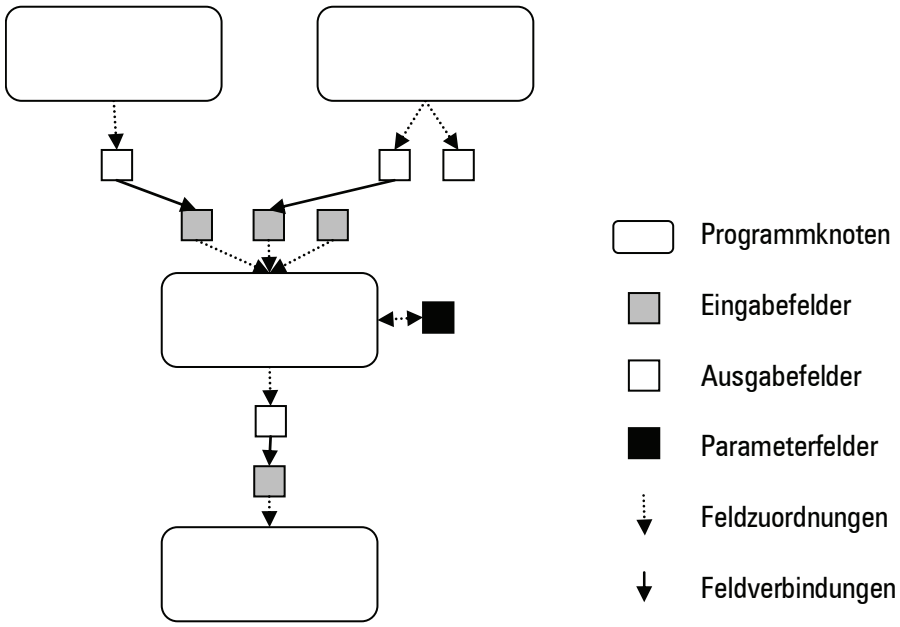


Abbildung 31: Graphdarstellung eines visuellen Programms

Abbildung 31 zeigt vier Programmknoten mit ihren zugeordneten Feldern, die über Feldverbindungen verbunden sind. Die Darstellung dient zur Veranschaulichung der Graphstruktur des visuellen Programms.

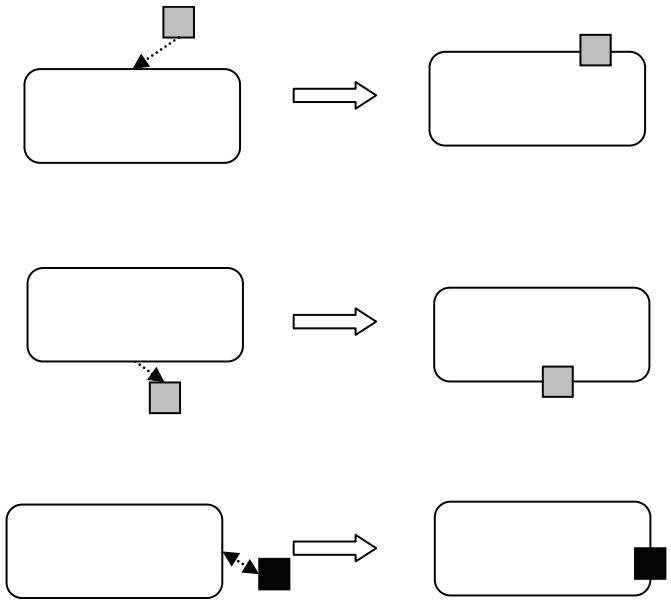


Abbildung 32: Visuelle Grammatik zur vereinfachten Darstellung der Graphstruktur

Abbildung 32 zeigt eine visuelle Grammatik mit deren Hilfe die Felder zur besseren Übersicht direkt auf der Kante des zugeordneten Programmknotens dargestellt werden können. Hierbei werden Eingabefelder im oberen Bereich, Ausgabefelder im unteren Bereich und Parameterfelder an der Seite des Programmknotens gruppiert. Abbildung 33 zeigt das so vereinfachte visuelle Beispielprogramm aus Abbildung 31.

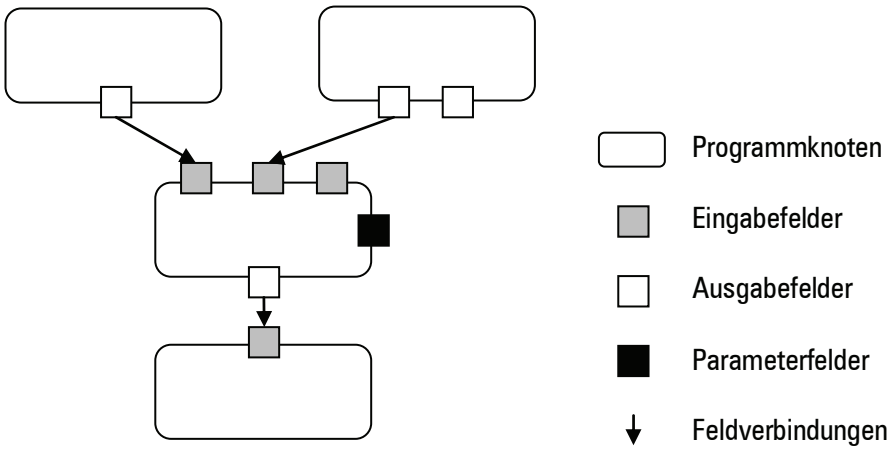


Abbildung 33: Vereinfachte Darstellung eines visuellen Programms

Da in der vereinfachten Darstellung die Felder nicht mehr als eigenständige Knoten des Graphen auftreten, legt die Definition eine direkte Verbindungsbeziehung der Programmknoten nahe.

Definition 4.4 (**Verbindung von Programmknoten**): Seien $p,p' \in P$ zwei Programmknoten dann gelte:

$$p \text{ ist } \textit{direkt verbunden} \text{ mit } p': \; p \rightarrow p'$$

genau dann, wenn

$$\exists f \in F_{Out}, f' \in F_{In} \;\; (p,f) \in E \wedge (f,f') \in E \wedge (f',p') \in E$$

Die indirekte Verbindung wird rekursiv über die direkte Verbindung definiert:

$$p \text{ ist } \textit{verbunden} \text{ mit } p': \; p \rightarrow^{*} p'$$

genau dann, wenn

$$p \rightarrow p' \;\; \vee$$
$$\exists p'' \;\; p \rightarrow p'' \wedge p'' \rightarrow^{*} p'$$

Das heißt zwei Programmknoten sind direkt verbunden, falls mindestens ein zugewiesenes Ausgabefeld des Elternknotens mit mindestens einem dem Kindknoten zugewiesenen Eingabefeld verbunden ist. Sie sind verbunden, falls es eine Kette von direkt verbundenen Programmknoten gibt, mit denen sie direkt verbunden sind. Zu beachten ist, dass bidirektionale Felder von der Betrachtung der Verbindungsbeziehungen ausgeschlossen sind. Sie spielen eine Sonderrolle bei der Verschaltung von Programmknoten, wie in Abschnitt 4.4.1 beschrieben.

Mittels dieser Definition können zyklische Verbindungen im Programm ausgeschlossen werden:

Definition 4.5 (**Azyklischer Programmgraph**): Seien $p,p' \in P$ zwei Programmknoten,

dann gelte:

$$\forall p,p' \in P \;\; p \rightarrow^{*} p' \Rightarrow p \neq p'$$

Ein Programmknoten darf nicht mit sich selbst verbunden sein

Die zusätzlich zu den Eingabefelder und Ausgabefeldern definierten Parameterfelder (F_{IO}) können auch über Feldverbindungen gekoppelt werden, sind aber nicht an den Aufbau azyklischer Strukturen gebunden. Dadurch erlauben sie Feedback-Verbindungen ohne gegen die Anforderung eines azyklischen Programmgraphen zu verstoßen.

Definition 4.6 (**Knotentypen**): Die Menge der Programmknoten P unterteilt sich in drei Mengen von Knotentypen:

$$P = P_{Sensor} \cup P_{Compute} \cup P_{Actor}$$

wobei:

- $P_{Sensor} = \{p \in P \mid \forall f \in F^{In}(f,p) \notin E \wedge \exists f \in F^{Out}(p,f) \in E\}$
  die Menge der *Sensorknoten* ohne Eingabefelder
- $P_{Compute} = \{p \in P \mid \exists f \in F^{In}(f,p) \in E \wedge \exists f \in F^{Out}(p,f) \in E\}$
  die Menge der *Rechenknoten* mit Eingabefeldern und Ausgabefeldern
- $P_{Actor} = \{p \in P \mid \exists f \in F^{In}(f,p) \in E \wedge \forall f \in F^{Out}(p,f) \notin E\}$
  die Menge der *Aktorknoten* ohne Ausgabefelder

Eine genauere Betrachtung der Knotentypen und ihrer Anwendungen findet sich in Abschnitt 4.5.

4.4 Ablaufmodell und Datenstrukturen

Im Gegensatz zu dem allgemeinen Ansatz der visuellen Datenfluss Programmierung verfolgt dieser Ansatz eine Spezialisierung auf die Verarbeitung eines kontinuierlichen Datenstromes – wie z.B. aus Trackerdaten oder Daten aus Datenhandschuhen – mit entsprechenden Rechenknoten. Dementsprechend sind zum Teil übliche Steuerungsknoten wie z.B. Schleifen innerhalb der visuellen Programmierung nicht implementiert. Sollten iterative oder rekursive Konstrukte benötigt werden, können diese in der imperativen Programmierung der Rechenknoten (siehe Abschnitt 4.2) realisiert werden. Diese Designentscheidung wurde getroffen, da sich zum einem iterative und rekursive Programmkonstrukte meist nur unübersichtlich als visuelles Programm darstellen lassen, und da zum anderen innerhalb dieser Konstrukte die Zeitinformationen der Datentoken besonders zu behandeln sind.

Die zweite Spezialisierung betrifft die zeitliche Verarbeitung der Daten. Da das vorgestellte System darauf ausgelegt ist, in eine VR-Applikation eingebettet zu werden, ist neben der Taktung der Eingabedaten vor allem die Darstellungsrate der einzelnen Renderframes von Bedeutung. Um Rechenaufwand und Verzögerungszeiten *(Latenz)*

zwischen der Datenaufnahme und der Darstellung in der Virtuellen Umgebung möglichst gering zu halten und gleichzeitig eine möglicht präzise zeitliche Verrechnung der Daten zu ermöglichen, sind die Berechnungszeitpunkte der Rechenknoten an die Framerate der VR-Applikation gekoppelt. Um die Präzision der Eingabedaten zu erhalten werden diese mit möglichst exakten Zeitstempeln des Aufnahmezeitpunktes Sequenzen gespeichert und weitergegeben. Dieses erlaubt auch die genaue zeitliche Zuordnung der Ergebnisse zu den Eingangsdaten ohne den Einfluss der benötigten Rechenzeit und die Integration unterschiedlich getakteter Datenquellen.

Da im Allgemeinen die Forderung nach präziser Verrechnung und geringen Rechenaufwand, bzw. geringer Latenzzeit gegeneinander stehen, können einzelne Programmteile je nach Anforderung in verschiedenen *Rechenmodi* betrieben werden. Die Rechenmodi entscheiden zwischen einer Berechnung mit geringer Latenz – z.B. falls die Daten für die direkte Manipulation von Objekten in der Virtuellen Umgebung benötigt werden – oder einer genauen Berechnung (ohne Extrapolation) z.B. für eine Trajektorienanalyse oder für die spätere Integration mit einem sprachlichen Kanal.

Dadurch ergeben sich unterschiedliche Aufrufe der Rechenmethoden für die einzelnen Rechenknoten. Je nach Modus werden diese nur einmal pro Renderframe oder für jedes Datentoken in der Sequenz von Daten für den aktuellen Frame aufgerufen.

Die Integration unterschiedlicher Datenquellen mit zum Teil unterschiedlicher Taktung kann in jedem einzelnen Rechenknoten erfolgen. Dadurch entfällt ein allgemeiner Vorverarbeitungsschritt für die Integration der Daten (wie z.B. bei dem PrOSA-Framework; siehe Abschnitt 2.5.2) und es ergibt sich eine große Flexibilität bei der Aufteilung der Programmstränge je nach gewünschter (zeitlicher) Präzision der Daten oder geringen Latenz- und Berechnungszeiten.

4.4.1 Feldwerte und Feldverbindungen

Um die Datenströme zwischen den Programmknoten zu verbinden, wird das Konzept der Datenfelder, bzw. Feldverbindungen benutzt. Dieses Konzept hat sich gerade im Bereich der Graphik- und VR-Tools durchgesetzt (vgl. Abschnitt 2.4).

Datenfelder erlauben einen geregelten Zugriff auf veränderbare Parameter einzelner Einheiten. Das Feldkonzept wird im VR-Beschreibungssprachen wie VRML/X3D eingesetzt, aber auch viele aktuelle VR-Tools (wie z.B. AVANGO, OpenSG, …) bieten Feldkonzepte als Schnittstelle für die die Parameter der Einheiten.

Die Zuordnung von Feldern zu den enthaltenden Werten wird durch die folgende Definition formalisiert.

Definition 4.7 (**Feld**): Ein Feld F vom Typ T besteht aus einem Zweiertupel:

$$F=(V, N)$$

wobei:

- $V:F\rightarrow T$
  die Wertefunktion des Feldes vom Typ T
- $N:F\rightarrow ASCII^*$
  der Name des Feldes

Zusätzlich sei: F^T die Menge aller Felder vom Typ T.

Parameterfelder und Datenfelder der visuellen Programme in VIPLIVE enthalten unterschiedliche Arten von Daten. Während die Werte der Datenfelder aus einer Sequenz von zeitgestempelten Datentoken bestehen, enthalten die Parameterfelder einfache Datentypen ohne zeitliche Information. Eine besondere Art von Feldern bilden die *Triggerfelder*. Sie haben keine Wertfunktion, sondern werden nur für die Propagierung von Feld-Events benutzt. Durch sie können in der visuellen Programmierung Berechnungen von Programmknoten angestoßen werden, ohne dass Feldwerte propagiert werden.

Definition 4.8 (**Triggerfeld**): Ein Triggerfeld f ist nicht typisiert und besitzt keine Wertzuweisung:

$$f:=(N)$$

wobei:

- $N:F\rightarrow ASCII^*$
  Name des Feldes

Zusätzlich sei: $F^{Trigger}$ die Menge aller Triggerfelder

Der Vorteil von Triggerfeldern gegenüber normalen Feldern mit Feldwert ist, dass sie von jedem Feld eines beliebigen Typs ausgehend verbunden werden dürfen. Da die anderen Felder Werte eines bestimmten Typs beinhalten, welche später durch die Feldverbindung zugewiesen werden, muss in dem Fall der Typ der zwei verbundenen Felder übereinstimmen.

Definition 4.9 (**typisierte Feldverbindungen**):

Es gelte:

$$\forall f, f' \in F \ (f, f') \in E \Rightarrow \exists t \ \ (f \in F^t \wedge f' \in F^t) \vee f' \in F^{Trigger}$$

Ein Feld darf nur zu einem Feld des gleichen Typs oder zu einem Triggerfeld verbunden werden.

Die Feldverbindungen zwischen den Feldern erlauben hierbei die Synchronisation von Feldwerten. Feldverbindungen sind gerichtet, d.h. eine Verbindung aktualisiert

die Feldwerte des Zielfeldes in Abhängigkeit zum Ausgangsfeld, falls dieses seinen Wert ändert, aber nicht umgekehrt.

Definition 4.10 (**Wertpropagierung durch Feldverbindungen**): Seien f1, f2 ∈ F zwei Felder vom gleichen Typ *T*,

dann wird der neue Feldwert V' von f2 wie folgt berechnet:

$$V'(f2)=V(f1) \text{ falls: } (f1,f2) \in E$$

Feldverbindungen propagieren die Feldwerte ihrer Verbindungspartner entlang der Verbindung.

In X3D bzw. VRML und vielen VR-Tools werden die Feldverbindungen benutzt, um die Parameter von Knoten im Szenengraph z.B. an Interpolatoren oder Aktuatoren zu koppeln und darüber Einfluss auf die Virtuelle Umgebung nehmen zu können.

Im Fall der visuellen Programmierung erlaubt das Feldkonzept die Weitergabe der Datentoken bzw. Feldwerte und somit die Verschaltung der einzelnen Rechenknoten.

Datenverbindungen und Steuerverbindungen

Für die Verbindung der Programmknoten über Felder gibt es, je nach Feldart zwei unterschiedliche Datentypen, für die unterschiedliche Regeln gelten.

Zum einem gibt es die *Datenverbindungen* der Datenfelder. Die Daten, die durch diese Verbindungen transportiert werden, bestehen aus Sequenzen von zeitgestempelten Daten. Sie ermöglichen damit eine unabhängige zeitliche Auflösung bei der Verrechnung der Daten (siehe Abschnitt 4.4.2). Für diese Art der Verbindungen werden Zyklen im Programmgraphen nicht erlaubt, da die Berechnung der einzelnen Daten innerhalb einer Sequenz in einem Block erfolgt und Daten mit Zeitstempeln, welche innerhalb des gerade berechneten Frames liegen, nachträglich nicht berücksichtigt werden können. Datenverbindungen werden von Dateneingabefeldern (F_{in}) zu den Datenausgabefeldern (F_{out}) der Rechenknoten entsprechend der Syntax des visuellen Programms erstellt.

Damit trotzdem Werte, die in tieferen Schichten des Programmgraphen berechnet werden, einen Einfluss auf obere Programmteile nehmen können, werden zusätzlich *Steuerverbindungen* eingeführt. Diese Art der Verbindung transportiert nur framebasierte Daten ohne zeitliche Information. Für sie sind aber rückwärts verkettete Verbindungen erlaubt. Steuerverbindungen koppeln z.B. an spezielle Steuerknoten – z.B. Switch- oder Merge-Knoten – wie sie aus dem Ablaufmodell für Datenflusssprachen (siehe Abschnitt 2.2.1) bekannt sind. Da bei diesen Steuerknoten die Steuerinformationen nur für Daten gelten können, welche von ihm noch nicht bearbeitet wurden, kann die genaue zeitliche Information, wie sie durch die Zeitstempel in den Sequenzen der Eingabedaten zur Verfügung steht, bei rückwärtigen Verbindungen nicht benutzt werden. Diese Steuerverbindungen werden zwischen den Parameterfeldern (F_{IO}) der Rechenknoten erstellt, da diese Felder nur einfache framebasierte Werte ohne zeitliche Information enthalten.

4.4.2 Datenstrukturen

Die Datenstrukturen und Zugriffsmethoden für die Verarbeitung und Integration von zeitgestempelten Datentoken der Datenfelder innerhalb der Programmknoten bauen auf dem Konzept des im PrOSA-Frameworks (Latoschik, 2001b) vorgestellten Abstrakten Aktuators auf (vergleiche auch Abschnitt 2.5.2).

Die im folgenden Abschnitt dargestellten Konzepte unterteilen und erweitern die bisher im Abstrakten Aktuator zusammengefassten Konzepte.

Das Konzept des Abstrakten Aktuators in PrOSA ist darauf ausgelegt, alle numerischen Berechnungen, welche für die symbolische Detektor-Ebene gebraucht werden, innerhalb einer Schicht zu berechnen. Im Gegensatz dazu muss das vorliegende Konzept, aufgrund der Berechnung mittels mehrer Programmknoten, einen möglichst performanten und einfach auf verschiedene Feldtypen abzubildenden Austausch der Daten über Feldverbindungen berücksichtigen. Daher wird die reine Datenhaltung der Datentoken und ihrer zeitlichen Informationen, welche über die Feldverbindungen transportiert werden, von dem Konzept der Zugriffsfunktionen getrennt.

Um die framebasierte Berechnung und Weitergabe der Daten in die Konzeption der Datentypen aufzunehmen, werden die Datentoken nicht wie bei PrOSA in einer zeitabhängig wachsenden Menge in den *Eingabekanälen*, sondern in Form von framebasierten Ausschnitten aus dieser Menge vorgehalten. Dieses Vorgehen spezialisiert das von dem PrOSA-Framework allgemeine mathematische Konzept der Datenhaltung unter Einschränkung der Allgemeingültigkeit des Ansatzes, ermöglicht aber eine direkte Umsetzung in der Implementierung mit möglicht generischen Datentypen für den Austausch der Datentoken und ihrer Zeitstempel und somit den einfachen Einsatz in unterschiedlichen VR-Tools.

Die *Synchronisationsfunktion* des Abstrakten Aktuators wird in das Konzept der Datenströme aufgenommen und durch die Einführung der selbst entwickelten Rechenmodi variabel erweitert. Dadurch können – je nach Anforderung an Latenzzeiten und Exaktheit der Daten – auch Berechnungszeitpunkte erlaubt werden, welche die Extrapolation der Werte in den Datenkanälen erfordern oder auf einen Berechnungszeitpunkt pro Renderframe limitiert werden.

Die in PrOSA vorgesehenen *Normierungsfunktionen* der Aktuatoren entfallen, da die Normierung der Eingangsdaten bei Bedarf durch eigenständige Rechenknoten vorgenommen werden kann.

Die *Berechnungsfunktion* für die Ausgabewerte wird durch die Rechencallbacks der Programmknoten (siehe Abschnitt 4.4.7) realisiert.

Die Weitergabe der Daten über die mit Bezeichnern versehenden *Attributsequenzen* wird durch die Verbindungen von bezeichneten Ausgabefeldern, welche die berechneten Datentoken enthalten, realisiert. Auch hier konkretisiert das vorliegende Konzept die allgemeine Weitergabe von Attributen als den Austausch von Datentoken über Feldverbindungen.

Datensequenzen

Um eine von der Framerate der Applikation unabhängige Berechnung des Eingabestromes an Daten ermöglichen zu können, aber auch die Möglichkeit zu haben, mit

nur einem Wert pro Renderframe zu rechnen, werden als Dateneinheiten zwischen den Programmteilen *Datensequenzen (DS)* eingesetzt. Durch die Aufteilung in framebasierte Einheiten, welche mehrere mit Zeitstempeln versehende Datentoken enthalten können, ist die Möglichkeit gegeben, diese Daten innerhalb der DS in einem durch eine externe Taktung gesteuertes Datenflussprogramm exakt einzeln abzuarbeiten.

Diese externe Taktung ergibt sich bei einer VR-Applikation sinnvollerweise durch die aktuelle Framerate der Darstellung. Damit ist der Übergang von einer Datensequenz mit eigener Taktung zu einer Berechnung mit einem Datentoken pro Frame sehr einfach zu erreichen. In diesem Fall müssen die im aktuellen Frame aufgelaufenen Daten zu einem Wert zusammengefasst werden. Dabei kann je nach Einsatzzweck der Daten ein möglichst präziser Mittelwert, das aktuellste Datum im Datenkanal oder aber auch eine Extrapolation der Daten auf den Zeitpunkt der Darstellung des Frames – z.B. für die Vorhersage von zukünftigen Positionen im Raum (*Motionprediction*) – sinnvoll sein.

Eine Datensequenz enthält die Menge aller Datentoken, die innerhalb des gerade zu bearbeiteten Frames aufgelaufen sind. Die Datentoken bestehen immer aus einem Zeitstempel und einem Wert, dessen Typ dem der DS entspricht. Zusätzlich enthalten sie die Zeitpunkte von Beginn und Ende des Datenframes. Diese Zeitpunkte geben das Intervall an in dem die aktuellen Daten aufgelaufen sind, und müssen nicht mit der Taktung der Berechnung übereinstimmen. Der Endzeitpunkt des Datenframes gibt das Ende des abgetasteten Zeitraumes an. Nachfolgende Datentoken müssen zeitlich hinter diesem Zeitpunkt liegen. Die Datensequenz beinhaltet zusätzlich noch weitere Informationen über die enthaltenen Daten. Hierzu gehören die von den Eingabedaten erwartete Framerate, die Interpretation der Daten, der Berechnungsmodus mit dem diese Daten entstanden sind und Priorität der Daten bezüglich eines Masterkanals.

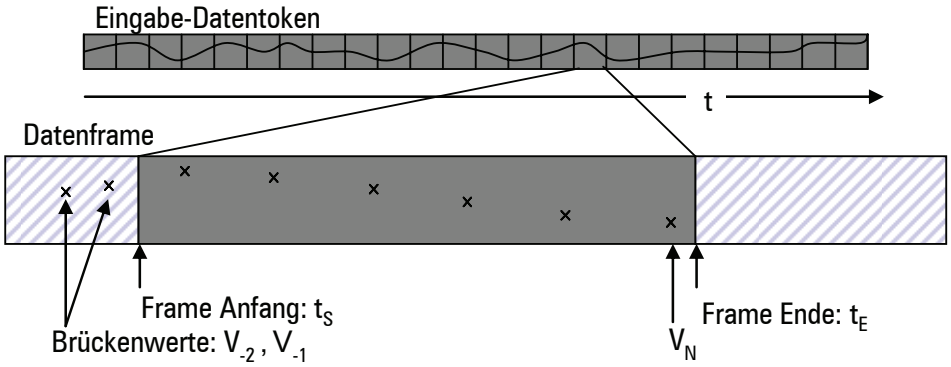


Abbildung 34: Eine Datensequenz

Datensequenzen enthalten alle Informationen der aufgenommenen Daten, der zeitlichen Anordnung und eventuell benötigte Zusatzinformationen. Da Datensequenzen in der Implementierung der visuellen Programmierung von den Programmknoten erzeugt und zwischen diesen übertragen werden, enthalten sie nur die benötigten Informationen über die Daten.

Definition 4.11 (**Datensequenz**): Eine Datensequenz vom Typ *T* enthält alle Daten und zeitlichen Informationen eines Datenframes. Sie besteht aus vier Komponenten.

$$DS=(t_S, t_E, D, I)$$

Wobei:

- $t_S \in \Re$: Zeitpunkt zum Beginn des Frames
- $t_E \in \Re$: Zeitpunkt zum Ende des Frames
- $D=\{(V_{-2}, t_{-2}), (V_{-1}, t_{-1}), \ldots (V_N, t_N)\}$ Menge von 2er Tupeln zeitgestempelter Datentoken

  wobei:

  $V_i \in T$ Wert vom Typ *T*; i = -2…N

  $t_i \in \Re$ Zeitpunkt der Aufnahme von V_i; i = -2…N
- $I = (\lambda, m_i, m_s, p)$ 4-Tupel von Zusatzinformationen zu den Daten

  $\lambda \in \Re$: die erwartete Datenrate

  $m_i \in \{Discrete, ExtraConst, ExtraLinear\}$: der Interpolationsmodus

  $m_s \in \{Const, Immediate, Sequence, Precise\}$: der Synchronisationsmodus

  $p \in \aleph$: die Master-Priorität

  Für die Zeitpunkte der Daten gilt:

  $t_{-2} \leq t_{-1} \leq t_S < t_0 < t_1 < \ldots < t_N \leq t_E < t_{N+1}$

  Zusätzlich gilt für die zeitlich folgende Datensequenz

  $ds'=\{t_S', t_E', \{(V_{-2}', t_{-2}'), \ldots (V_N', t_N')\}, I'\}$:

  $t_S' = t_E$

  $(V_{-2}', t_{-2}') = (V_{N-1}, t_{N-1})$

  $(V_{-1}', t_{-1}') = (V_N, t_N)$

  $I' = I$

Das ersten beiden Datentoken der Datensequenz (V_{-2}, t_{-2}) und (V_{-1}, t_{-1}), liegen zeitlich vor dem Beginn des Frames und entsprechen den beiden letzten Datentoken, die vor dem aktuellen Frame aufgezeichnet wurden. Diese zusätzlichen Daten sind für jede DS definiert. Zu Beginn werden die beiden Werte auf den Initialwert der DS gesetzt und die Zeitstempel auf -1 und 0. Die beiden Datentoken ermöglichen die lineare Interpolation über den gesamten zeitlichen Bereich der Sequenz bis hin zum aktuellsten Datentoken und die lineare Extrapolation darüber hinaus. Außerdem ermöglichen sie die Abfrage des aktuellsten Datenwertes, falls innerhalb des Frames kein neues Datentoken aufgelaufen ist.

Die Methoden für den einheitlichen Zugriff und typspezifische Interpolationsberechnungen werden durch Datenkanäle bereitgestellt.

4.4.3 Datenkanäle

Ein *Datenkanal* (*DC*) kapselt die für den aktuellen Frame gültige DS und stellt außerdem einheitliche Methoden zur Anfrage von Daten innerhalb des durch die DS definierten Datenframes zur Verfügung.

Definition 4.12 (**Datenkanal**): Ein Datenkanal vom Typ *T* kapselt eine Datensequenz des gleichen Typs und stellt die Funktionen zu Zugriff der Werte zur Verfügung.

$$DC=(DS,\ T,\ T^{Max},\ V_{Value}^{InterpolationMode},\ V_{Cur},\ V_{Avg})$$

Wobei:

- $DS = \{t_S, t_E, \{(V_0, t_0), \ldots (V_N, t_N)\}, (\lambda, m_i, m_s, p)\}$
  die gekapselte Datensequenz mit N+1 Dateneinträgen
- $T: DS \rightarrow \{\Re\}$
  Menge der Zeitstempel der aktuellen Daten (ohne Brückenwerte)
  $T(\{t_S, t_E, \{(V_{-2}, t_{-2}), \ldots (V_N, t_N)\}, (\lambda, m_i, m_s, p)\})=\{t_i \mid i \in [0..N]\}$
- $T^{Max}: DS \rightarrow \Re$
  Der späteste Zeitpunkt für Berechnungen ohne Extrapolation

$$T^{Max}(\{t_S, t_E, \{(V_{-2}, t_{-2}),...,(V_N, t_N)\}, (\lambda, m_i, m_s, p)\}) = \begin{cases} t_E & \text{für } m_i = \text{Discrete} \\ t_N & \text{sonst} \end{cases}$$

- $V_{Value}^{InterpolationMode}: DS \times R \rightarrow T$
  Funktion zur Abfrage des Wertes zu einem Zeitpunkt t der Datensequenz $ds=\{t_S, t_E, \{(V_{-2}, t_{-2}),...,(V_N, t_N)\}, (\lambda, m_i, m_s, p)\}$:
  für den Fall m_i = Discrete:

$$V_{Value}^{Discrete}(ds, t) = \begin{cases} \text{undef.} & \text{für } t < t_S \vee t > t_E \\ V_N & \text{für } t_N \le t < t_E \\ V_i & \text{für } t_i \le t < t_{i+1} \end{cases}$$

  für m_i = *ExtraConst*

$$V_{Value}^{ExtraConst}(ds, t) = \begin{cases} \text{undef.} & \text{für } t < t_S \\ I_{lin}(V_i, V_{i+1}, \frac{t - t_i}{t_{i+1} - t_i}) & \text{für } t_i < t \le t_{i+1} \\ V_N & \text{sonst} \end{cases}$$

  wobei: $I_{lin}(A,B,w)$: eine lineare Interpolationsfunktion mit dem Gewicht $w \in [0..1]$ passend zum Typ *T* und der Interpretation d_s der gekapselten Datensequenz

Definition (**Datenkanal**) [cont.]

für m_i = *ExtraLinear*

$$V_{Value}^{ExtraLinear}(ds,t) = \begin{cases} \text{undef.} & \text{für } t < t_S \\ I_{lin}(V_i, V_{i+1}, \frac{t-t_i}{t_{i+1}-t_i}) & \text{für } t_i < t \le t_{i+1} \\ E_{lin}(V_{N-1}, V_N, \frac{t-t_N}{t-t_{N-1}}) & \text{sonst} \end{cases}$$

wobei: $I_{lin}(A,B,w)$: die lineare Interpolationsfunktion wie oben

$E_{lin}(A,B,w)$: eine lineare Extrapolationsfunktion mit dem Gewicht $w > 0$ passend zum Typ *T* und der Interpretation d_s der gekapselten Datensequenz

- V_{Cur}: DS→ *T*

  Abfrage des aktuellsten Wertes der Datensequenz

  $V_{Cur}(\{t_S, t_E, \{(V_{-2}, t_{-2}),...,(V_N, t_N)\}, (\lambda, m_i, m_s, p)\}) = V_N$

- V_{Avg}: DS→ *T*

  Abfrage des zeitlich gewichteten Durchschnitts über den aktuellen Datenframe

  $$V_{Avg}(\{t_S, t_E, \{(V_{-2}, t_{-2}),...,(V_N, t_N)\}, (\lambda, m_i, m_s, p)\}) = \frac{(t_1 - t_S)\cdot V_0 + \sum_{i=1}^{N-1}(t_{i+1} - t_i)\cdot V_i + (t_E - t_N)\cdot V_N}{t_E - t_S}$$

Die verschiedenen Abfragefunktionen haben unterschiedliche Vor- und Nachteile. Die Auswahl der passenden Abfrage über den Interpolationsmodus m_i hängt vom Datentyp, dessen Interpretation und dem gewünschten Anwendungsverhalten ab:

$V_{Value}^{Discrete}(ds,t)$: Aktueller Wert einer Datensequenz ds zum Zeitpunkt t:

Diese Art der Anfrage geht davon aus, dass ein Datentoken der Datensequenz solange gültig ist, bis er von einem aktuelleren Datum abgelöst wird. Interpolation findet nicht statt. Dies ist vor allem für boolsche und andere diskrete Werttypen sinnvoll. Der Zeitraum für Anfragen ohne Extrapolation ist hierbei das Ende der Datensequenz, da durch die Definition sichergestellt ist, dass bis es bis dahin keinen aktuelleren Wert als V_N gibt.

$V_{Value}^{ExraConst}(ds,t)$: Lineare Interpolation mit konstanter Extrapolation für $t > t_N$

Der gültige Zeitraum für die Anfrage interpolierter Daten geht von dem Zeitpunkt des Brückenwertes t_0 bis zum Zeitstempel des aktuellsten Datentoken. Bei Anfragen für Zeitpunkte nach dem letzen Datentoken wird der aktuellste Wert V_N zurückgegeben. Hierbei können sich je nach Lage der Framegrenzen unterschiedliche Berechnungswerte ergeben.

Die lineare Interpolationsfunktion $I_{lin}(A,B,w)$ sieht z.B. für skalare Werte aus wie folgt:

$$I_{lin}(A,B,w) = w \cdot A + (w-1) \cdot B$$

Für komplexere Datentypen mit Interpolation muss eine spezielle Interpolationsfunktion angegeben werden. Hierbei spielt auch die Interpretation der Daten eine Rolle. Während z.B. Vektoren, welche eine Position im Raum beschreiben, ähnlich der skalaren Interpolation berechnet werden können, sollten Richtungsvektoren auf andere Weise interpoliert werden. 4x4-Matrizen die als Transformation interpretiert werden, werden in ihre einzelnen Komponenten Translation, Rotation und Skalierung zerlegt, welche dann einzeln interpoliert werden, anstatt der Interpolation der einzelnen Matrixwerte, wie man sie bei einem unabhängigen Wertearray vornehmen würde.

$V_{Value}^{ExtraLinear}(ds,t)$: Lineare Interpolation und Extrapolation

Hierbei werden nach dem letzten Datentoken V_N werden die Daten linear extrapoliert. Nicht jede Art von Eingangsdaten eigenen sich aber für eine aussagekräftige Extrapolation. Bei stark verrauschten oder sonst störungsanfälligen Daten kann die konstante Extrapolation bessere Daten liefern. Auch hier können je nach Lage der Framegrenzen bei Extrapolation unterschiedliche Wertverläufe entstehen.

Die lineare Extrapolationsfunktion $E_{lin}(A,B,w)$ sieht für skalare Werte aus wie folgt:

$$E_{lin}(A,B,w) = B + w \cdot (B-A)$$

4.4.4 Datenströme

Um mehrere Datenkanäle, welche unterschiedliche Taktungen haben können, bei der Berechnung im Datenflussprogramm zusammenfassen zu können wird das Konzept des Datenstromes eingeführt.

Definition 4.13 (**Datenstrom**): Ein Datenstrom fasst mehrere Datenkanäle zusammen und stellt ein passendes zeitliches Raster für die Berechnungen zur Verfügung. Er besteht aus einem 4-Tupel:

$$DS=(DC,\ DC^m,\ M_c,\ T_c)$$

wobei:

- $DC = \{DC_1, \ldots, DC_N\}$

  Menge der zusammengefassten Eingabekanäle

- $DC^m \in DC$

  der Masterkanal

- $M_c \in \{precise, sequence, immediate\}$

  Der Berechnungsmodus

- $T_c : M_c \times DC \times DC^m \rightarrow \{\Re\}$

  die Synchronisationsfunktion, welche in Abhängigkeit vom aktuellen Berechnungsmodus, der aktuellen Menge von Eingabekanälen $dc = \{dc_1,\ldots,dc_n\}$ und dem aktuellen Masterkanal k^m die Zeitpunkte für die Berechnungen im aktuellen Frame liefert.

  $T_c\ (sequence, dc, dc^m) = \{t \mid t \in T(ds^m)\}$

  $T_c\ (immediate, dc, dc^m) = \{t \mid t \in T(ds^m) \wedge \forall t_i \in T(ds^m) : t \geq t_i)\}$

  $T_c\ (precise, dc, dc^m) =$

  $$\{t \mid t \in (T(ds^m) \cup T^{Store}) \wedge t \leq \min(T^{Max}(ds_1),\ldots, T^{Max}(ds_N))\}$$

  wobei:

  - ds^m die Datensequenz des Masterkanals dc^m, und

    ds_i die Datensequenz des entsprechenden Eingabekanals dc_i

  - T^{Store}

    Die Menge der noch nicht berechneten Zeitschritte, welche ausgehend von $T^{Store} = \varnothing$ iterativ berechnet wird:

    $$T^{Store} = \{t \mid t \in (T^{Store} \cup T_{K_m}) \wedge t > \min(T^{Max}_{K_1},\ldots, T^{Max}_{K_N})$$

Ein Eingabe-Datenstrom fasst bei jedem Programmknoten die eingehenden Datenkanäle zusammen und stellt ein einheitliches Interface zum Zugriff auf die Daten zur Verfügung. Hierbei müssen die unterschiedlichen Taktungen der einzelnen Kanäle

berücksichtigt werden und ein passendes zeitliches Raster für den Wertezugriff festgelegt werden. Dabei wird durch den Berechnungsmodus die gewünschte zeitliche Präzision bei der Verrechnung der Daten berücksichtigt (siehe Abschnitt 4.4.5). Der Datenstrom eines Programmknotens bestimmt anhand der Synchronisationsfunktion ob, bzw. wie oft und mit welchen Daten die Berechnungsroutine dieser Einheit im aktuellen Frame aufgerufen werden muss und damit, wie viele Daten in den aktuellen Ausgabestrom geschrieben werden. Falls im Precise-Modus die Daten aus dem aktuellen Datenframe nicht komplett verarbeitet wurden, weil z.B. die aktuellen Daten aus einem Kanal noch fehlen, können auch Berechnungszeitpunkte in T^{Store} vorgemerkt werden. Für diese Berechnungen müssen die schon für diesen Zeitraum vorliegenden Datenwerte aus den Datenkanälen gespeichert werden.

Ein Datenkanal des Stromes wird als Masterdatenkanal ausgewählt und bei der Zusammenführung von mehreren Kanälen zu einem Datenstrom das zeitliche Raster für den Datenstrom übernommen. Dieses entspricht der gekoppelten festen Synchronisation der Abstrakten Aktuatoren. Für die Daten im Masterkanal hat das den Vorteil, dass keine zusätzliche Latenzzeit durch ein Resampling entsteht und die Datenwerte direkt ohne Interpolation übernommen werden können. Um für den Fall, dass mehrere Kanäle als Master ausgezeichnet sind, einen Master auswählen zu können, werden für die Kanäle Masterprioritäten vergeben. Hierbei bekommen die Kanäle mit wichtigen Daten oder sehr konstanten Datenraten (wie z.B. Daten aus einem optischen Trackingsystem) die höchsten Prioritäten als Masterkanal.

Für jeden Programmknoten wird ein Masterkanal bestimmt. Spezielle Resampling-Knoten (siehe Abschnitt 4.5.2) können aber eine eigene Taktung für den Datenstrom vorgeben und somit als Masterkanal für die folgenden Programmzweige die Datenrate bestimmen. Dadurch wird die ungekoppelt feste Synchronisation der Abstrakten Aktuatoren abgebildet. Um die Probleme eines kompletten Resamplings (Erhöhung der Latenz und Rundungsungenauigkeiten durch Interpolation) zu umgehen, können die Resample-Knoten die Daten eines Masterkanals auch ausdünnen und z.B. nur jeden zweiten oder dritten Wert und seinen Zeitstempel übernehmen. Damit kann bei Programmteilen, welche nicht die volle zeitliche Auflösung zur Berechnung benötigen, Rechenzeit eingespart werden.

4.4.5 Berechnungs-Modi

Da die Forderungen nach möglichst geringen Latenzzeiten und möglicht präziser Berechnung gegeneinander stehen und auch eine möglicht geringe Verarbeitungszeit des gesamten Programms anzustreben ist, müssen die verschiedenen Programmteile unterschiedlich behandelt werden.

So erfordern manche Programmteile die volle zeitliche Auflösung und Präzision der Eingabedaten. Als Beispiel ist hier die Analyse von Bewegungstrajektorien zu nennen, wo z.B. bei einer Abtastrate der Trackerdaten von 60 Hertz durch das Überspringen von Datensamplen aufgrund eines Resampling erhebliche Störungen bei der Erkennung innerhalb von schnellen Bewegungen entstehen können. Auch bei der späteren Integration von durch ein visuelles Programm detektierter Gestik und der erkannten Sprache des Benutzers (siehe Abschnitt 6.5.2) sind präzise Zeiten der erkannten Features entscheidend. Bei Teilstücken des visuellen Programms, das z.B. eine interaktive Manipulation in der Virtuellen Umgebung steuert, ist nicht die absolute Präzi-

sion der Daten, sondern vielmehr eine geringe Latenzzeit entscheidend für eine gute Handhabung des Systems. Hierbei sollten immer aktuellste (evtl. auch extrapolierte) Daten der Berechnung zugrunde liegen, wobei die Berechnung von Datensamples, welche nicht für die aktuelle Darstellung benötigt werden, übersprungen werden können.

Eine geringe Latenzzeit der Eingangsdaten und eine geringe Verarbeitungszeit dieser Daten sind vor allem bei der direkten Manipulation von Objekten wichtig. Bei der Integration von Gestik und Sprache hingegen sind vor allem die korrekten Zeitangaben der berechneten Daten von Interesse, da bei der Integration die zeitliche Abfolge von erkannten Gesten und der erkannten Sprache eine bedeutende Rolle spielt (Latoschik, 2001b).

Um den verschiedenen Anforderungen an Präzision, Frameraten und Latenzzeiten gerecht werden zu können, werden die Berechnung der Daten in den Datensequenzen der Datenkanäle in verschiednen Modi betrieben:

Precise-Modus

Bei diesem Modus werden die Daten erst dann verrechnet, wenn alle benötigten Datenströme ein aktuelles Datum enthalten. Das heißt, hier ist keine Extrapolation der Daten erlaubt. Beim Zugriff auf Daten zu einem Zeitpunkt im Datenkanal wird ein möglichst präziser Wert durch Interpolation errechnet. Bei Datenkanälen ohne Interpolation ist immer ein Zugriff auf gültige Daten bis zum Ende des Datenframes möglich. Außerdem ist für den Masterkanal, dessen Zeitstempel und Werte für die Berechnung übernommen werden, keine Interpolation notwendig. Daher wird dieser Modus nur angewendet, falls mindestens ein Slave-Kanal des Datenstromes einen zu interpolierenden Datentyp enthält. Für diesen Modus müssen die Datenkanäle die noch nicht benutzten Datentoken zwischenspeichern, bis alle benötigten Daten anliegen und die Verrechnung durchgeführt werden kann. Da eine Abschätzung noch nicht vorhandener Daten z.B. durch Extrapolation nicht durchgeführt wird, wird durch diesen Modus die präziseste Verrechnung der Eingabedaten erreicht. Dies geht im Allgemeinen auf Kosten einer höheren Latenzzeit, da eventuell nicht die aktuellsten Daten in die Berechnung einfließen.

Sequence-Modus

Hier werden die Daten bis zum aktuellsten Datum des gesamten Datenstromes berechnet. Daher müssen bei der Zusammenführung von mehreren Datenkanälen die Daten der evtl. noch fehlenden Kanäle extrapoliert werden. Dieses kann zu einer ungenaueren Berechnung bei der Interpolation der Einzeldaten und zu einer Abhängigkeit der Werte von der aktuellen Framerate führen (siehe Abschnitt 4.4.2). Der Modus ermöglicht aber im weiteren Datenfluss den Zugriff auf möglichst aktuelle Daten. Für den Fall, dass die Daten nicht interpoliert werden, ist das Ergebnis der Berechnungen mit dem Precise-Modus identisch.

Immediate-Modus

Beim Immediate-Modus wird maximal ein neuer Wert pro Applikationstakt berechnet. Die Berechnung ist dadurch abhängig von der externen Framerate, was für präzise Berechnungen sowohl in der zeitlichen Granularität als auch – wie bei dem

Sequence-Modus – bei der Präzision der Daten von Nachteil sein kann, aber eine effiziente Berechnung mit geringen Latenzzeiten erlaubt.

Der Immediate-Modus ist speziell für die Berechnung von Daten für die Visualisierung in der Virtuellen Umgebung gedacht, da hier die Daten mit möglichst geringer Latenzzeit und nur einmal pro Frame benötigt werden. Je nach Höhe des Verhältnisses von Datenrate zu Framerate kann mit diesem Modus – im Vergleich zu den vorherigen beiden – Rechenzeit eingespart werden, was für das Ziel niedriger Latenzzeiten und hoher Frameraten entscheidend sein kann. Da die Berechungsrate an die Framerate gebunden ist, ergibt sich bei der Berechung ein von der Gesamtlast des Systems abhängiges Level-Of-Detail Verfahren. Während bei den Modi mit konstanter Datenrate bei sinkender Framerate der Applikation mehr Berechnungen pro Frame durchgeführt werden müssen, werden im Immediate-Modus bei sinkender Framerate der Applikation insgesamt automatisch weniger Berechnungen durchgeführt, was zur Stabilisierung der Datenrate beitragen kann. Ist die Datenrate hingegen niedriger als die Framerate, müssen innerhalb der Frames, in denen keine neuen Eingangsdaten aufgelaufen sind, auch keine neuen Berechnungen gemacht werden. Dieser Modus kommt auch bei der Konvertierung von framebasierten Daten zu Datensequenzen zum Einsatz.

Const-Modus

Für konstante Daten, die nur bei Neukonfigurationen oder während des Ablaufes gar nicht verändert werden, wird der Const-Modus benutzt. Die initialen DS eines Feldes werden mit diesem Modus erzeugt und auch das Setzen von Dateneingabefeldern auf konstante Werte generiert Datensequenzen im Const-Modus.

4.4.6 Ablaufschema der Rechenknoten

Das Ablaufschema der Programmknoten stellt sicher, dass die Berechnungs-Callbacks (OnFrame bzw. OnComputeValue) erst aufgerufen werden, wenn alle Rechenknoten der eingehenden Datenfelder ihre Berechnungen durchgeführt haben.

Die Sensorknoten werden bei jedem Frame als erstes evaluiert, da sie die Datenquellen des Programms bilden. Dadurch werden neue Datentoken auf ihren Ausgabefeldern erzeugt und an die verbundenen Felder weiter propagiert.

Einzelne Rechenknoten und Aktorknoten werden zur Evaluierung angemeldet, sobald eines ihrer Eingabefelder einen neuen Wert erhält. Hier muss sichergestellt werden, dass Knoten erst dann evaluiert werden, wenn alle seine Eingabeknoten schon abgearbeitet sind. Falls mehrere Rechenknoten gleichzeitig zur Evaluation anstehen, werden diese nach einer Prioritätenliste abgearbeitet. Dieses verhindert das mehrmalige Abarbeiten eines Knotens wenn mehrere seiner Eingabefelder neue Werte erhalten. Dafür werden die Prioritäten für die Evaluierung wie folgt vergeben:

- Alle Sensorknoten erhalten die höchste Priorität.
- Hat ein Knoten nur ein verbundenes Eingabefeld, erhält dieser Knoten die gleiche Priorität wie der Eingabeknoten.
- Hat ein Knoten mehrere eingehende Verbindungen, erhält dieser eine niedrigere Priorität als die niedrigste Priorität seiner Eingabeknoten.

Wenn Rechenknoten während der Evaluierung Daten auf ihren Ausgabestrom geschrieben haben, werden diese Ergebnisse am Ende der Berechnung auf die Ausgabefelder gesetzt und durch die Feldverbindungen an die Eingabefelder der verbundenen Knoten propagiert, welche dann zur Evaluierung angemeldet werden.

An dieser Stelle kann die Evaluation mehrerer Programmknoten parallelisiert werden. Eine einfache Vorgehensweise ist die parallele Berechnung der zur Evaluation angemeldeten Programmknoten, welche die gleiche Priorität haben. Wenn zusätzlich sichergestellt wird, dass die Berechnung der Programmknoten erst nach Abschluss der Berechnungen der Knoten mit einer höheren Priorität anfängt, ist durch die oben angegebene Vergabe der Prioritäten garantiert, dass das Ablaufschema der Knoten eingehalten wird. Eine Analyse der Datenströme – die als Vorverarbeitung bei der Veränderung der Graphstruktur geschehen kann – erlaubt eine weitere Parallelisierung der Berechnung von unabhängigen Datensträngen. Da die Aufteilung in mehrere Prozessstränge potentiell die Performanz der Berechnungen steigert, aber gleichzeitig durch die Verwaltung der Threads und den zusätzlichen Speicherplatzbedarf ein Mehraufwand entsteht (vergleiche Abschnitt 2.2.2), kann je nach den Vorraussetzungen, den die aktuell eingesetzte Hardware bietet (Multiprozessoren, Hyperthreading, Speicherplatz, etc.) die Aufteilung in zu viele kleine Rechenstränge unterbunden werden.

Die *Aktorknoten* werden genau so wie die Rechenknoten evaluiert. Da sie aber keine Ausgabefelder und somit auch keine von ihnen abhängige Knoten besitzen, muss die Fertigstellung ihrer Bearbeitung bei der Parallelisierung der Berechnungen nicht berücksichtigt werden.

Falls das vorhandene Ablaufschema des eingesetzten VR-Tools mächtig genug ist, das vorgestellte Schema abzubilden, kann es direkt in der Ablauflogik des Datenflusskonzeptes der Knoten implementiert werden. Dieses hat den Vorteil für die VR-Umgebung nur ein Ablaufschema einsetzen zu müssen. Dieses Vorgehen wurde z.B. im Fall der Einbindung in die Programmierumgebung von AVANGO gewählt.

Ist das Ablaufschema der Zielplattform nicht mächtig genug, oder handelt es sich dabei um ein Szenengraphtool ohne eine komplette Anbindung von Datenflusskonzepten, können flexibel Teile des Ablaufschemas von des *Field-Event-Layers* übernommen werden. Die Implementierung des Field-Event-Layers enthält ein eigenes Feldkonzept mit Feldverbindungen und Programmknoten. Eine eigene Shedule-Engine der Knoten sorgt für die Aufrufe der entsprechenden Evaluate-Callbacks. Auch alle anderen benötigten Callbacks für die internen Berechnungen, wie sie im folgenden Kapitel vorgestellt, werden von dem Layer bereitgestellt.

4.4.7 Berechnungs-Callbacks der Rechenknoten

Die einzelnen Rechenknoten stellen verschiedene Callbacks für mögliche Berechungen bzw. Initialisierungen zur Verfügung. Diese Callbacks unterscheiden sich vor allem in der Zeitlichkeit ihrer Aufrufe.

Der **‚OnInit'**-Callback wird genau einmal bei der Instanziierung eines Rechenknotens aufgerufen. Hier können Variablen initialisiert, oder z.B. für die Visualisierungsknoten Szenengraphteile aufgebaut werden.

Der **‚OnFieldChanged'**-Callback wird direkt bei jeder Änderung eines Wertes an den Eingabefeldern aufgerufen und bezieht sich auf ein angegebenes Eingabefeld. In diesem Programmteil können Berechnungen angestellt werden, die direkt mit der Änderung von dem Wert dieses Feldes gekoppelt sind. Zum Beispiel können Felder für Richtungsvektoren bei Änderung der Feldwerte die Normierung dieser Vektoren anstoßen.

Falls sich Feldwerte geändert haben, werden nach der Änderung aller Eingabedatenfelder durch das oben beschriebene Ablaufschema die Rechenknoten zur Evaluation angemeldet. Während dieser Evaluation der Knoten wird einmal der **‚OnFrame'**-Callback aufgerufen. Hier werden Berechnungen angestoßen, die trotz mehrer Datentoken an den Eingabefeldern nur einmal pro Frame durchgeführt werden sollen. Ein Beispiel sind hier Veränderungen am Szenengraph, welche nur zum Zeitpunkt des Renderings Auswirkungen haben, und daher nicht mehrmals pro Frame berechnet werden sollten. Dieser Callback kann je nach Anforderung am Anfang des Frames – d.h. vor der Berechnung der aktuellen Feldwerte – oder am Ende des Frames ausgeführt werden.

Für jeden Berechnungszeitpunkt der Werte, wie sie in den Datenkanälen durch die Synchronisationsfunktion vorgegeben werden (siehe Abschnitt 4.4.3), wird einmal der **‚OnComputeValue'**-Callback aufgerufen. Innerhalb dieses Callbacks werden die aktuellen Datenwerte des Datenstromes durch vordefinierte Variablen entsprechend der Wertabfragefunktion $V_{Value}^{InterpolationMode}$ des Datenkanals und dem zugeordneten Interpolationsmodus zur Verfügung gestellt. Dieser Callback ist auch für die Berechnung und Erzeugung der neuen Werte auf den Ausgabekanälen des Rechenknotens verantwortlich. Dafür wird für jeden Ausgabefeld eine entsprechende ‚set'-Methode definiert, welche dann in diesem Callback aufgerufen wird und ein neues Datentoken mit dem Zeitstempel der aktuellen Berechnung aus den Ausgabekanal schreibt.

4.5 Programmknotentypen

Die Knoten des visuellen Programms sind nach Definition 4.6, je nachdem, ob ihnen Eingabefelder und/oder Ausgabefelder zugeordnet sind, formal in drei Gruppen unterteilt:

- *Sensorknoten* stellen Daten zur Verfügung, die direkt von der Sensorhardware aufgenommen, oder von externen Programmen vor verarbeitet wurden.
- *Rechenknoten* integrieren unterschiedliche Datenströme und stellen die gewünschten Berechnungen an.
- *Aktorknoten* bilden die Datensenken. Sie können z.B. Operationen im Szenengraph ausführen, die Daten visualisieren oder für eine Weiterverwendung speichern.

4.5.1 Sensorknoten

Die Sensorknoten bilden die Datenquellen für die virtuellen Programme. Sie sind die Schnittstelle zu allen Eingangsdaten, die für die Abarbeitung des Programms gebraucht werden, falls sie nicht selber durch ein visuelles Programm erzeugt werden. Ein Grossteil der Eingangsdaten wird durch Sensorhardware erzeugt, welche in der

Regel Informationen über den Benutzer aufnimmt. Aber Sensoren können auch Informationen bereitstellen, die in anderen Prozessen der Virtuellen Umgebung – wie z.B. die Kollisionserkennung – erzeugt wurden. Schließlich können Sensoren selber auch Zufallsdaten erzeugen. Daten, die durch Sensorknoten bereitgestellt werden, können beliebige zeitliche Auflösungen haben.

Eingabesensoren

Eingabesensoren können sowohl Daten in einem festen zeitlichen Raster bereitstellen, wie sie z.B. von einem Trackingsystem erzeugt werden. Auch sporadisch verteilte Daten, die durch die Anbindung eines Eingabegerätes (z.B. ein Trackball oder ein einfacher Taster), das nur bei der Betätigung durch den Benutzer Daten sendet, können durch Sensoren bereitgestellt werden. Im Sensorknoten werden die externen Daten in Attributsequenzen gespeichert und in Datenkanälen verpackt, welche dann an den Ausgabefeldern zur Verfügung stehen. Da die Interpolation von diskreten Daten – z.B. ob ein Knopf gedrückt ist oder nicht – nicht sinnvoll ist, bestimmen Sensorknoten über die Konfiguration des Datenkanals auch, ob die bereitgestellten Daten interpoliert werden dürfen und setzen auch die Priorität als Masterkanal. Die Masterpriorität der Daten ergibt sich durch die Qualität des zeitlichen Rasters der Daten. Hier sind Datenkanäle mit einer möglichst hohen und konstanten Datenrate, wie sie z.B. ein optisches Trackingsystem liefert, zu bevorzugen.

Neben den Sensorknoten für Daten aus der Virtuellen Umgebung (6DOF-Tracking, Datenhandschuhen, Stylus, etc.) werden auch Sensoren für die Eingabegeräte einer Desktopumgebung (Tastatur, Maus) bereitgestellt. Das ermöglicht die Interaktion am normalen Bildschirm und auch ein eingeschränktes Testen der erstellten Programme. So wird durch ein einfaches Umschalten der Datenströme von dem Stylus-Sensor der Virtuellen Umgebung auf den Maus-Sensor die gleiche Interaktion am Desktop, wie in der Virtuellen Umgebung ermöglicht[8].

Alle durch externe Sensorhardware aufgenommenen Daten werden durch Sensorknoten bereitgestellt, welche zusammen mit Zeitstempeln des Aufnahmezeitpunktes der Daten an die erzeugte Datensequenz angehängt werden. Falls diese Zeitpunkte nicht durch die Hardware selber bereitgestellt werden, müssen die Sensorknoten eine möglichst genaue Abschätzung der Aufnahmezeitpunkte durch die ermittelte Latenzzeit zwischen Aufnahme und Eingang der Daten vornehmen. Falls nötig können die Sensorknoten auch schon eine Normierung der Rohdaten vornehmen.

Multicast-Netzwerksensoren

Die Netzwerksensoren sind spezielle Eingabesensoren, welche Ereignisse und Daten von externen Softwarekomponenten über die Schnittstelle der Sensorknoten bereitstellen. Diese Sensorknoten empfangen ihre Daten über einen Multicast-Netzwerkmechanismus. Das ermöglicht auch die Auslagerung von Programmteilen auf andere Rechner im Netzwerk. Hier können z.B. basierend auf den Daten der Tracker schon Vorberechnungen übernommen werden und erst die vorverarbeiteten Daten an die Hauptapplikation übermittelt werden. Der Multicast-Mechanismus sorgt dafür, dass

[8] Natürlich mit der Einschränkung auf eine zweidimensionale Bewegung bei den Daten des Maus-Sensors. Benutzt wird dieses z.B. bei der Bearbeitung der visuellen Programme in der Desktop-Umgebung.

die Daten parallel für alle Programme, die auf Rechnern im Netzwerk laufen, mit möglicht geringem Netzwerkverkehr zur Verfügung stehen.

Da die externen Programme aber keine Informationen über die Renderframes der VR-Applikation haben, müssen sie die Datentoken direkt einzeln verarbeiten und an das Hauptprogramm weiter senden. Die Integration von Daten mit unterschiedlicher Taktung sollte erst im Hauptprogramm der VR-Applikation geschehen, da erst hier die genaue zeitliche Steuerung der Datenintegration vorgenommen werden kann.

GUI-Sensoren

Neben der freien sprachlich-gestischen Eingabe des Benutzers können auch fest vorgegebene graphische Eingabefunktionen sinnvoll sein. Diese *Graphical User Interfaces (GUI)* können in einer Virtuellen Umgebung z.B. als Berührungsschalter oder Schieberegler *(Slider)* auftauchen. 3D-GUI-Sensoren bieten in der Virtuellen Umgebung platzierbare Schalter und Slider, welche den Ausgabewert auf den entsprechenden Ausgabefeldern bereitstellen. 2D-GUI-Sensoren erlauben Eingaben über übliche zweidimensionale Schalter, Slider und Tastatureingabefelder. Sie sind für den Testeinsatz in der Desktopumgebung gedacht und können in einem eigenen Steuerungsfenster platziert werden. Die GUI-Sensoren können Abläufe in der Virtuellen Umgebung, aber auch direkt dem Datenfluss der visuellen Programme steuern.

Datenabspielsensoren

Neben den Sensoren, welche aktuelle Daten erzeugen bzw. empfangen und bereitstellen, gibt es auch Sensoren, die Daten aus einer vorher aufgenommenen Datei auslesen. Die Dateien speichern in einem XML-Format rohe oder vorverarbeitete Trackingdaten, welche z.B. von den Datenrekordern (siehe Abschnitt 4.5.3) aufgenommen wurden.

Diese Datenabspielsensoren verfügen über eine zeitliche Steuerung, welche den zeitlichen Abgleich der eingelesene Daten aus verschiedenen Dateien oder die Integration mit aktuell aufgenommenen Daten ermöglicht.

4.5.2 Rechenknoten

Ein Rechenknoten bearbeitet den Datenstrom der Datenkanäle seiner Eingabefelder und produzieren Ergebnisse als Datentoken auf den Ausgabefeldern. Das Ablaufschema (siehe Abschnitt 2.2.1) legt fest, wann welche Rechencallbacks der Knoten aufgerufen werden. Diese Callbacks können durch bereitgestellte Funktionen Datentoken in die Kanäle der Ausgabefelder schreiben.

Arithmetik

Die Rechenknoten stellen in der Regel arithmetische Operationen für typische Datentypen der VR-Umgebung zur Verfügung. Diese Operationen reichen von einfachen Grundrechenarten der verschiedenen Datentypen bis hin zu für das konkrete Programm spezialisierten komplexen Berechnungen. Da das XML-Format der Programmknoten eine einfache und schnelle Erweiterung der bereitgestellten Rechenknoten erlaubt, ist das Erstellen von spezialisierten Knoten, deren Berechnungen sich besser in der externen imperativen Sprache beschreiben lassen, ohne großen Aufwand möglich.

Datenflusskontrolle

Entsprechend dem Ablaufschema der klassischen Datenflussprogrammierung können Rechenknoten auch den Datenfluss zwischen den Knoten im visuellen Programm steuern. Die Knoten können z.B. von mehreren Eingabedatenkanälen, je nach dem aktuellen Wert an der Steuerleitung, einen auswählen und auf das Ausgabefeld des Knotens legen. Der Wert an der Steuerleitung ergibt sich in der Regel aus der Berechnung eines anderen Programmknotens, kann aber auch durch eine Auswahl der Benutzers im GUI mittels eines GUI-Sensors bestimmt werden. Dieses ermöglicht dann die interaktive Kontrolle des Datenflusses durch einen Benutzer in der Virtuellen Umgebung.

Resampling

Resamplingknoten können das zeitliche Raster der Eingabedaten verändern, so dass das neue Raster als Masterkanal für die folgenden Rechenknoten übernommen wird. Dieses Resampling kann nach Vorgabe einer festen Datenrate geschehen, oder das vorhandene Zeitraster ausdünnen. Während im ersten Fall die Daten im Eingabekanal interpoliert werden müssen und sich durch das Resampling eine erhöhte Latenzzeit ergibt (siehe Abschnitt 4.7) werden im zweiten Fall nur bestimmte Datensamples an den Ausgabekanal weitergegeben. Dieses kann bei konstanter Datenrate der Eingabedaten z.B. jeder zweite oder dritte Datensample sein, oder bei weniger konstanten Abständen der Datensamples das nächste Datensample nach einer vorgegebenen Zeit.

Szenengraphabfragen

Da die Programmknoten neben den Daten ihrer Eingabefelder auch Zugriff auf die Szenengraphstruktur haben, können Rechenknoten auch Informationen bereitstellen, welche aus dem Szenengraph abgeleitet werden können.

Ein Beispiel hierfür ist ein Knoten, der zu einem Liniensegment die Szenengraphknoten eines vorgegebenen Typs findet, welche diesem Segment am nächsten liegen. Dieser Knoten wird z.B. bei der Auswahl von Objekten durch eine Zeigegeste des Benutzers eingesetzt.

4.5.3 Aktorknoten

Visualisierungsknoten

Die Visualisierungsknoten erhalten ihre Daten über die Eingabefelder und können diese auf geeignete Weise in der Virtuellen Umgebung darstellen. Der Einsatzbereich dieser Visualisierungen ist vielfältig. So können Visualisierungsknoten bei dem Verständnis und der Fehlersuche in einem visuellen Programm von entscheidender Bedeutung sein. Sie ermöglichen neben der direkten Visualisierung der Feldwerte eines aktuell ausgewählten Feldes die konstante Visualisierung von Zwischenergebnissen des Programms. Sie bilden damit eine „What You See Is What You Test“ (WYSIWYT) Umgebung, welche sich als vorteilhaft für visuelle Programme herausgestellt hat (siehe Abschnitt 2.1).

Ein anderer Einsatzbereich der Visualisierungsknoten ergibt sich durch den Aufbau von Datenvisualisierungen zeitlich und räumlich komplexer Datensätze. Hierbei

können die visuellen Programme zusammen mit den Visualisierungen ähnlich der in Abschnitt 2.4.3 beschriebenen Visualisierungstools ganze Visualisierungsumgebungen in der Virtuellen Umgebung aufbauen.

Datencontainer

Eine grundlegende Eigenschaft der Datenflusssprachen ist, dass auf die Historie der Daten nicht zugegriffen werden kann, da immer nur die aktuellen Daten im DF-Graphen vorhanden sind. Damit nicht nur aktuelle Daten abgefragt werden können, sondern auch der zeitliche Verlauf der Daten für bestimmte Zeiträume bereitgestellt werden, können die Daten aus einem Datenkanal in einen entsprechenden Container gespeichert werden. Da die Datencontainer die Ergebnisse der Berechnungen über einen definierten Zeitraum vorhalten, dienen sie als Schnittstelle zu anderen Komponenten der Applikation.

Ein Beispiel für die Übergabe von Daten an externe Programme ist die Übergabe der Ergebnisse der Gestenerkennung an die Komponente zur multimodalen Integration (siehe Abschnitt 6.5.2), welches die erkannten Gesten mit einem sprachlichen Kanal für die multimodale Interaktion integriert.

Datencontainer entsprechen Ringspeichern, welche die mit Zeitstempeln versehenen Daten für eine festgelegte Zeit bereitstellen. Sie bieten ein komfortables Interface zum Zugriff auf die gespeicherten Daten zu einem bestimmten Zeitpunkt oder einem Zeitintervall. Hierbei kann für den Zugriff auf einen Zeitpunkt festgelegt werden, ob ein interpolierter Wert oder einfach der Wert mit dem am besten passenden Zeitstempel zurückgegeben wird. Bei der Abfrage eines Zeitintervalls wird der durchschnittliche Wert des Intervalls zurückgegeben. Im Falle von Datencontainern mit booleschen Werten werden die Werte ‚Wahr' und ‚Falsch' auf 0 bzw. 1 abgebildete und dann interpoliert. So kann einfach abgefragt werden, ob in einem Zeitraum ein ‚Wahr'-Wert aufgetaucht ist (Intervallabfrage > 0) oder der Wert den ganzen Zeitraum über ‚Wahr' war (Intervallabfrage $= 1$) und ähnliches. Bei einem Container für Vektoren muss festgelegt werden, ob sich z.B. um Positionsvektoren oder um Richtungsvektoren handelt, um hier die Interpolation richtig berechnen zu können. Auch die Interpolation von Matrizen kann je nach Interpretation der Matrix in der Applikation unterschiedlich berechnet werden.

Datenrekorder

Ähnlich der Datencontainer gibt es entsprechendes Datenrekorder, welche die Daten aus dem eingehenden Datenstrom in einem XML-basierten Format abspeichern können. Diese können dann z.B. für weitere Testläufe des Programms oder für eine spätere ‚offline'-Analyse oder Visualisierung der Daten wieder eingelesen werden. Für eine mögliche Einbindung von neuronalen Netzen als Komponenten in die visuelle Programmierung können hiermit auch Trainingsdaten für das Trainieren der Netze bereitgestellt werden.

Scripting-Knoten

Um komplexere Aktionen in der Virtuellen Umgebung auszulösen, werden Scripting-Knoten eingesetzt, wie sie z.B. auch in dem X3D-Format vorkommen (siehe Abschnitt 2.5.1). Über ihre Eingabefelder kann die jeweilige Aktion angestoßen werden. Eine Script-Funktion kann diesem Knoten als Callback zugewiesen werden. Dadurch

sind diese Knoten sehr flexibel in der visuellen Programmierung einsetzbar. Da diese Funktionen aber auch beliebige Felder von anderen Knoten verändern können, muss der Programmierer sich bewusst sein, dass diese Veränderung von Feldwerten unter Umgehung des Ablaufschemas der Datenflussprogrammierung stattfindet und somit Seiteneffekte auf andere Programmteile haben kann. Der Haupteinsatzzweck dieser Knoten sind komplexe Veränderungen im Szenengraphen, und z.B. auch die Erstellung neuer Feldverbindungen bei der Interaktion mit den visuellen Programmen.

4.6 Level-Of-Detail der Berechnungen

Zusätzlich zu der Steuerung über die Berechnungsmodi können aber auch ganze Programmteile mit einer flexibel den Anforderungen angepassten Datenrate betrieben werden, und nicht benötigte Programmteile ganz von der Evaluation ausgenommen werden. Dieses ermöglicht eine einfache *Level-Of-Detail* (LOD) Anpassung einzelner Programmteile unter Berücksichtigung der benötigten Präzision der Daten und der Ablaufgeschwindigkeit der Datenflussprogramme.

Da alle Rechenknoten maximal mit dem Takt des Masterkanals arbeiten, können für verschiedene Programmstränge die Datenraten sehr übersichtlich auf die Anforderungen der aktuellen Berechnung angepasst werden. Ist – z.B. für die Berechnung statischer Fingerstellungen – eine geringere Datenrate nötig als sie von der Sensorhardware geliefert wird, reicht ein Resample-Knoten am Anfang der entsprechenden Programmkette, um den Berechnungstakt für die ganze Kette herabzusetzen. Da mitunter viele folgende Rechenknoten betroffen sind, wird dadurch wertvolle Rechenzeit eingespart.

Durch Switch-Steuerknoten können dynamisch ganze Programmteile durch die Unterbrechung des Datenstromes stillgelegt werden, deren Berechnungen in der aktuellen Situation nicht benötigt werden. So kann z.B. eine rechenaufwendige Bewegungssegmentierung und -erkennung dynamisch stillgelegt werden, falls der Benutzer sich gerade kaum bewegt.

Ähnlich den LOD-Verfahren in aktuellen Szenengraphtools, bei denen je nach Rechenlast des Systems und der aktuellen Sichtbarkeit (Entfernung vom Betrachter) unterschiedlich komplexe Geometrien zur Darstellung ausgewählt werden, können auch dynamisch – z.B. abhängig von der zur Verfügung stehenden Rechenkapazität und Auslastung – andere Taktraten oder sogar unterschiedliche Rechenstränge für die Berechnung benutzt werden. So lassen sich Kompromisse zwischen der aktuellen Genauigkeit der berechneten Daten und der maximal möglichen Auslastung der Rechenkapazität erreichen.

4.7 Latenzzeitberechnungen

Die Zeit zwischen der Aufnahme der Daten, z.B. von den Positions-Trackern des Benutzers, und der Darstellung dieser abgegriffenen Daten in der Virtuellen Umgebung ist entscheidend für den Eindruck der Immersion und der Ausführung von präzisen Interaktionen. In (Bryson, 1996) und (Kreylos et al., 2001) wird eine Zeit von maximal 100 Millisekunden festgelegt, innerhalb der die Rückmeldung eines VR-Systems erfolgen muss. Auch die Framerate darf nach (Kreylos et al., 2001) ein Limit von 30 Bildern pro Sekunde nicht unterschreiten. Daher ist das visuelle Programmier-

system, das für den Einsatz in VR-Systemen gedacht ist, auch für Berechnungen mit möglichst geringen Latenzzeiten ausgelegt.

4.7.1 Latenzzeiten in den Sensorendaten

Bei der ersten Abschätzung der Latenzzeiten der Eingangsdaten sind zwei Faktoren von Bedeutung: Die Latenz, die schon bei der Aufnahme, Vorverarbeitung und Übertragung der Daten bis zum Eingabestrom entstanden ist (Ld), und die Datenrate (Frequenz) mit der die Daten aufgenommen werden (Fd).

Damit ergeben sich für den Zugriff auf die möglicht aktuellsten Daten zu einem beliebigen Zeitpunkt folgende Latenzzeiten:

Die minimale Latenzzeit L_{min} (bei Zugriff genau zu dem Zeitpunkt eines Datensamples):

$$L_{min} = Ld$$

Die maximale Latenzzeit L_{max} (bei Zugriff genau am Ende eines Datenframes):

$$L_{max} = Ld + \frac{1}{Fd}$$

Und die durchschnittlich zu erwartende Latenzzeit arithmetisch gemittelt:

$$L_{avg} = \frac{L_{min} + L_{max}}{2} = Ld + \frac{1}{2Fd}$$

4.7.2 Latenzzeiten bei der Verrechnung

Bei der Verrechnung von zwei Eingangsdatenkanälen mit unterschiedlicher Taktung ist es von Bedeutung, in welchem Modus die Daten verrechnet werden. Bei dem Precise-Modus ergibt sich eine größere Latenz als bei den beiden anderen Modi, da hier auf die aktuellen Datensamples aus beiden Datenkanälen gewartet werden muss.

Es ergibt sich für den Precise-Modus:

$$L_{avg}^{pc} = \max(L_{p1}, L_{p2}) + \frac{1}{2F_{d1}} + \frac{1}{2F_{d2}}$$

In diesem Fall müssen die erwarteten Latenzzeiten, die durch die Framezeiten der beiden Datenkanäle entstehen, addiert werden, da bei Ende des Frames des einen Kanals im ungünstigsten Fall ein ganzer Frame des anderen Kanals gewartet werden muss, bis bei beiden ein aktuelles Datum anliegt. Die Latenzzeiten aus der Vorverarbeitung der Daten addieren sich hingegen nicht, sondern eine der beiden – in der Regel der Masterkanal – bestimmt die gesamte Latenz.

Für den Sequence- und Immediate-Modus ergibt sich folgendes:

$$L_{avg}^{sq} = L_{avg}^{im} = \max(L_{p1}, L_{p2}) + \frac{1}{2F_{Master}}$$

Hier bestimmt nur die Latenzzeit des Masterkanals die gesamte Latenz, da mit einem Wert aus dem Master-Datenkanal schon ein neuer Wert berechnet werden kann und die Werte in den anderen Kanälen ohne Rücksicht auf die Aktualität der Daten einfach interpoliert bzw. extrapoliert werden.

Beim Precise-Modus erhöht sich also bei jeder Zusammenführung von zwei Datenkanälen, welche unterschiedlich getaktet sind, die zu erwartende Latenzzeit. Beim Sequence-Modus hingegen kann, falls über den gesamten Programmstrang ein Mastertakt beibehalten wird, auch bei der Zusammenführung von mehren, unterschiedlich getakteten Datenkanälen die Latenzzeit konstant gehalten werden.

Bei der normalen Berechnung von nur einem Datenkanal oder mehreren Kanälen mit gleicher Zeitabfolge der Daten bleibt die Latenzzeit des Datenstromes konstant. Es ist allerdings darauf zu achten, dass viele Berechnungen im gesamten Programm die Framerate verringert und somit die Latenzzeit für sämtliche Berechnungen entsprechend erhöht.

Jedes Resampling der Daten im Programmablauf erhöht die zu erwartende Latenz um die Hälfte des Kehrwertes der neuen Samplingfrequenz. Daher ist ein Resampling vor allem bei Daten, die zur direkten Darstellung in der Virtuellen Umgebung dienen und damit auf möglicht niedrige Latenzzeiten angewiesen sind, möglichst zu vermeiden.

Bei der Zusammenführung von zwei oder mehreren Datenkanälen zu einem Datenstrom wird ein Masterkanal festgelegt. Hierbei sollte entweder der Kanal mit den grundlegenderen Daten[9] oder der mit der für den Berechnungsstrang besser passenden Datenrate ausgewählt werden. Beim Immediate-Modus ist das allerdings nicht von Bedeutung, da hier kein eigentliches Resampling der Daten stattfindet, sondern immer die jeweils aktuellsten Daten pro Renderframe genommen werden.

Ingesamt muss für die Latenzzeit des Gesamtsystems noch die Zeit für die Berechnung aller Daten und die Zeit für das Rendering addiert werden. Hierzu gehören auch die Latenzzeiten, welche durch die Hardware der Ausgabegeräte noch dazukommen (z.B. eventuelle Netzlaufzeiten im Rendercluster und die Framerate bzw. Schaltzeiten der darstellenden Beamer).

Bei der direkten Verarbeitung der Daten aus nur einen Datenkanal ergibt sich damit eine durchschnittliche Gesamtlatenz von:

$$L_{avg}^{ges} = L_{preD1} + \frac{1}{2F_{d1}} + \frac{1}{F_{render}} + L_{postHW}$$

Zum Beispiel ergibt sich im System mit einem Datenkanal 60 Hertz und einer Framerate von 50 Hz die folgende durchschnittliche Gesamtlatenz:

$$\begin{aligned} L_{avg}^{ges} &= 0.015 + 1/60 + 1/50 + 0.01 \\ &= 0.015 + 0.0166 + 0.02 + 0.01 \\ &= 0.0666 \end{aligned}$$

[9] Z.B. sind bei der Berechnung einer Fingerposition die Trackerdaten von größerer Bedeutung als die Daten des Handschuhs, da die Position und Orientierung der Hand mehr Einfluss auf die Fingerposition haben als die Gelenkwinkeldaten aus dem Datenhandschuh.

Bei der Berechnung im Precise-Modus bei zwei Datenkanälen mit 60 Hz (Master) bzw. 80 Hz ergibt sich:

$$L_{avg}^{ges} = 0.015 + 1/60 + 1/80 + 1/50 + 0.01$$
$$= 0.015 + 0.0166 + 0.0125 + 0.02 + 0.01$$
$$= 0.0791$$

Wie man sieht, erhöht sich die durchschnittlich zu erwartende Latenz im Precise-Modus bei der Zusammenführung von zwei Kanälen, obwohl der zweite Kanal eine höhere Framerate hat, als der ursprüngliche Masterkanal.

Wird dieselbe Berechnung im Immediate-Modus gemacht, resultiert das in der folgenden Latenzzeit:

$$L_{avg}^{ges} = 0.015 + \max(1/60, 1/80) + 1/50 + 0.01$$
$$= 0.015 + 0.0166 + 0.02 + 0.01$$
$$= 0.0666$$

Das Latenzverhalten bei der Integration von zwei Kanälen im Immediate-Modus entspricht dem des Einzelkanals. Hierbei ist aber zu beachten, dass ein Wert dabei in der Regel extrapoliert werden muss und daher nicht so exakte Werte bei der Berechnung geliefert werden können. Zudem ist das Ergebnis durch die unterschiedlichen Zeitpunkte der Extrapolation abhängig von dem Berechnungstakt, wobei im Precise-Modus das Ergebnis unabhängig von den Berechnungszeitpunkten ist.

4.7.3 Beispiele zur Verrechnung von Datenkanälen

In diesem Absatz werden Beispiele angegeben, welche die Unterschiede in den Ergebnissen und den Latenzzeiten der verschiedenen Berechnungsmodi aufzeigen. Diese Unterschiede sind vor allem abhängig von der Datenrate der beiden Kanäle und bei den Sequence- und Immediate-Modus auch von dem Berechnungstakt der Programmknoten.

Integration von Datenkanälen mit extrem unterschiedlicher Datenrate

Dieses Beispiel zeigt eine Berechnung mit zwei (für Testzwecke erzeugten) Datenkanälen mit sehr unterschiedlicher Datenrate, da man hier gut die prinzipiellen Unterschiede der Berechnungsmodi zeigen kann. Beide Kanäle enthalten Zufallszahlen im Fließkomma-Format; der erste mit einer Datenrate von 1 Hertz und im Zahlenbereich von 2 bei 4 und der zweite mit einer Rate 60 Hertz im Bereich von 0 bis 0,3. Zur Verrechnung der beiden Kanäle wurde in diesem Beispiel ein einfacher Additionsknoten verwendet. Als Masterkanal wurde der zweite Datenkanal festgelegt, da dieser die zeitlich feiner aufgelösten Daten enthält. Somit ergibt sich für den Ausgabekanal im Precise- und Immediate-Modus auch eine Datenrate von 60 Hertz. Im Fall des Immediate-Modus entspricht die Datenrate des Ausgabekanals dem des externen Berechnungstaktes, welcher für dieses Beispiel auf ca. 20 Hertz festgelegt wurde[10].

[10] In einer realen Anwendung ergibt sich der Berechnungstakt aus der Updaterate der gesamten

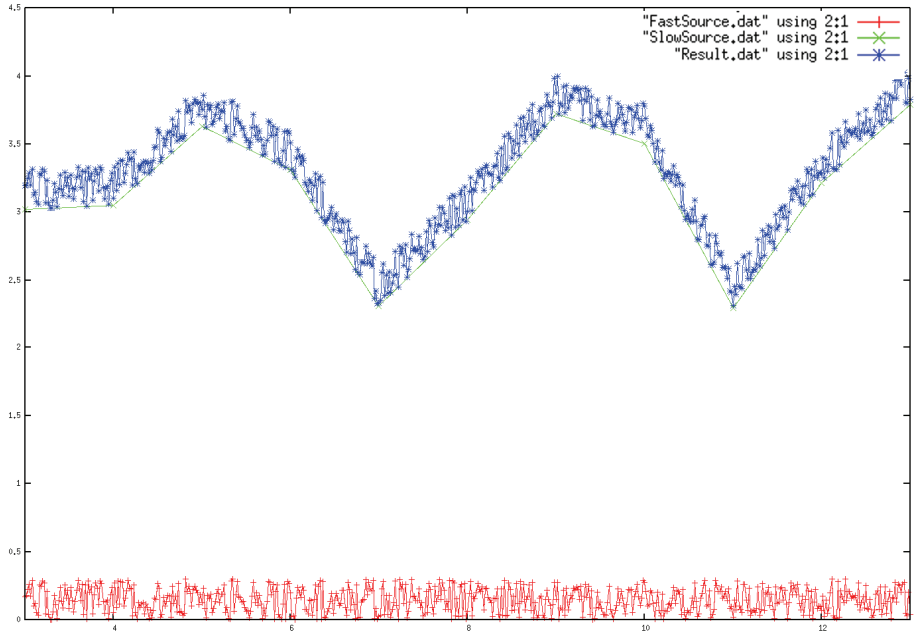


Abbildung 35: Plot der Datentoken bei der Berechnung im Precise-Modus

Abbildung 35 zeigt die Daten der beiden Eingabekanäle und das Ergebnis im Ausgabekanal des Additionsknotens für den Fall einer Berechnung im Precise-Modus. Zusätzlich ist als weiterer Plot die real gemessene Latenzzeit der Berechnung der einzelnen Datentoken aufgezeichnet. Die Latenzzeit bezieht sich dabei nur auf den Zeitversatz, der durch die Berechnung, bzw. durch die Vorgabe des Rechenmodus entsteht und wird gemessen als Differenz zwischen den Zeitstempeln der Datentoken und dem Zeitpunkt zu dem die Ergebnisdaten vorliegen. Sowohl die initiale Latenzzeit durch die Sensorhardware – welche bei der Erzeugung der Daten durch Zufallszahlen in diesem Fall sowieso nicht vorhanden ist – als auch der nachfolgende Versatz durch das Rendersystem werden hier nicht beachtet, da hier der Fokus auf die Zeiten bei der Berechnung der Daten gelegt werden soll.

In Abbildung 35 sieht man in dem Plot der Daten die korrekte Addition der Datentoken mittels Interpolation der Daten im langsameren Datenkanal. Dargestellt sind die beiden Ausgangssequenzen, zum einem die mit der hohen Datenrate in dem Plot unten und zum anderen die Datensequenz mit der niedrigen Datenrate als durchgehende, interpolierte Linie in der Mitte. Die resultierende Datensequenz der Addition ist oberhalb der langsamen Sequenz abgebildet.

Simulationsschleife der Anwendung und ist daher in der Regel nicht so konstant wie für dieses Beispiel erzeugt, sondern kann je nach aktueller Last des Systems schwanken.

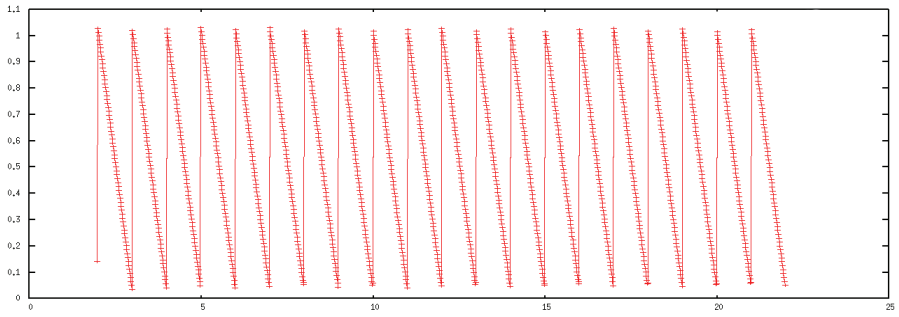


Abbildung 36: Latenzzeiten der Daten im Precise-Modus

In Abbildung 36 ist ein Plot der sägezahnartig schwankenden Latenzzeiten der Ausgabedaten in diesem Modus zu sehen. Da im Precise-Modus die Daten immer nur bis zu dem Zeitpunkt berechnet werden können, zu dem in allen Kanälen ein aktuelles Datentoken vorliegt, muss die Berechnung in diesem Beispiel immer bis zu einer Sekunde auf das Datentoken im langsameren ersten Datenkanal warten. Erst dann können alle Datentoken zwischen dem letzten und dem aktuellen Token berechnet werden. Die Wartezeit der einzelnen Datentoken bis zur ihrer Berechnung schwankt daher ungefähr zwischen 0 und 1 Sekunde. Es ergibt nach der Formel aus Abschnitt 4.7.2 bei der Berechnung – ohne die Betrachtung der Zeit für die Abarbeitung selber – eine durchschnittliche Latenz von:

$$
\begin{aligned}
L_{avg}^{pc} &= \max(L_{p1}, L_{p2}) + \frac{1}{2F_{d1}} + \frac{1}{2F_{d2}} \\
&= 0 + \frac{1}{2 \cdot 1} + \frac{1}{2 \cdot 60} \\
&= 0.508\overline{3}
\end{aligned}
$$

Die real gemessene mittlere Latenzzeit für die Beispieldaten ist 0.5125, was dem theoretischem Wert sehr nahe kommt.

Wird die Berechnung bei derselben Taktung der Datenquellen im Sequence-Modus gemacht, kann die Latenzzeit hier drastisch verringert werden. Andererseits macht der Datenverlauf im Plot der Datentoken in Abbildung 37 auch die Probleme durch die Extrapolation der Daten sehr deutlich.

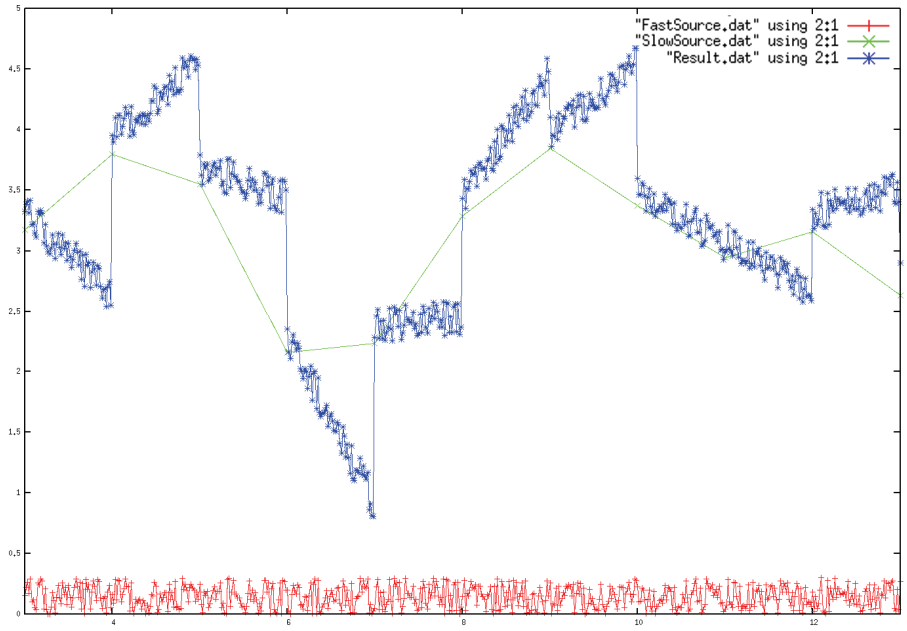


Abbildung 37: Plot der Datentoken bei der Berechnung im Sequence-Modus

Gerade bei Daten, welche sich durch einen stark schwankenden Verlauf für eine lineare Extrapolation nicht eignen (wie in diesem Beispiel aufgrund der benutzten Zufallszahlen der Fall), kann das Ergebnis stark von der präzisen Berechnung abweichen. Die Wahl des Berechnungsmodus kann daher nur anhand von weiteren Informationen der Daten (wie z.B. Interpolierbarkeit, Datenrate, Datentyp etc.) und der letztendlichen Verwendung getroffen werden, wobei die Unterschiede aufgrund einer im Realfall geringeren Differenz in den Datenraten der Kanäle nicht so extrem ausfallen, wie in diesem künstlichen Beispiel.

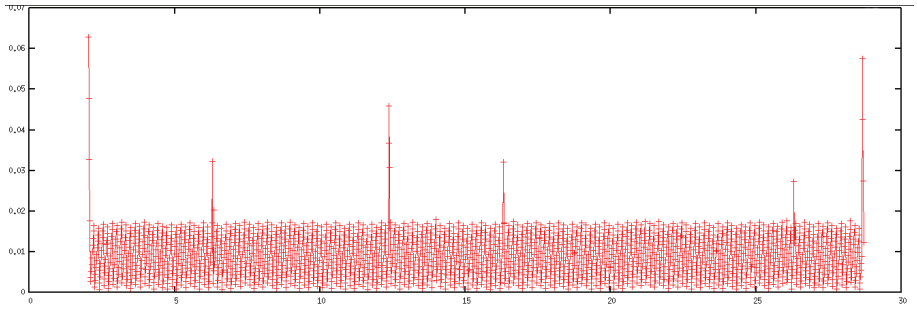


Abbildung 38: Latenzzeiten der Daten im Sequence-Modus

Wie in Abbildung 38 zu sehen, bewegt sich die Latenzzeit bei der Berechnung im Sequence-Modus nur noch ungefähr zwischen 0 und 0,02 Sekunden, mit wenigen Ausreißern aufgrund einer nicht absolut konstanten Framerate der Applikation. Entsprechend der theoretischen Vorhersage durch die Formel aus Abschnitt 4.7.2:

$$
\begin{aligned}
L_{avg}^{sq} &= \max(L_{p1}, L_{p2}) + \frac{1}{2F_{Master}} \\
&= 0 + \frac{1}{2 \cdot 60} \\
&= 0.008\overline{3}
\end{aligned}
$$

Auch hier ist die gemessene, durchschnittliche Latenzzeit von 0.009246 nahe am theoretischen Wert, wenn man beachtet, dass die Ausreißer in der Framerate die durchschnittliche Latenzzeit erhöhen.

Bei der Berechnung im Immediate Modus ergibt sich ein ähnlicher Plot wie im Beispiel vorher im Sequence-Modus (siehe Abbildung 39) mit den gleichen Problemen durch die lineare Extrapolation. Der Hauptunterschied ist hierbei die geringere Datenrate in der resultierenden Datensequenz. Da in diesem Modus nur das aktuellste Datum für den jeweiligen Render-Frame berechnet wird, passt sich die Datenrate der Updaterate der Applikation an. Für dieses Beispiel wurde die Updaterate der Applikation auf 20 Hertz beschränkt.

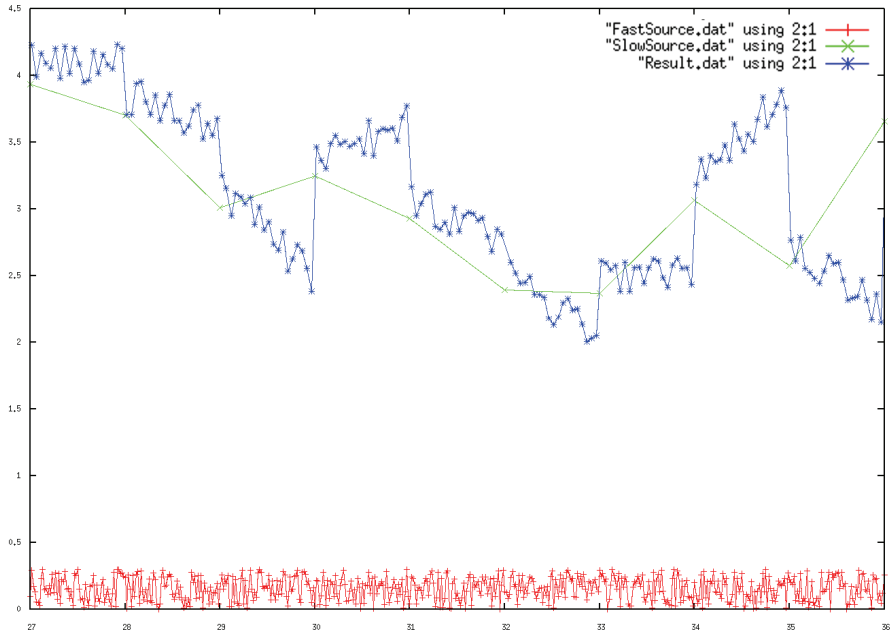


Abbildung 39: Plot der Datentoken bei der Berechnung im Immediate-Modus

4.8 Visualisierung der Programme in der Virtuellen Umgebung

Die Visualisierung der Programmknoten in der Virtuellen Umgebung folgt weitgehend der Graphvisualisierung bei der formalen Beschreibung der Syntax der Programme (siehe Abschnitt 4.3). Um den Aufwand für das Rendering der Knoten möglichst gering zu halten, werden Programmknoten und Felder als einfache Boxen dargestellt, welche mit möglichst wenigen Polygonen auskommen. Wie bei der vereinfachten Visualisierung als Graph werden die Eingabe- und Ausgabefelder auf dem Rand der Programmknoten dargestellt. Anders als bei der formalen Graphvisualisierung werden die Parameterfelder nicht nur an der Seite der Knoten dargestellt. Die bei der Definition der Felder für die Visuelle Programmierumgebung vorgesehen Parameterfelder, welche nach den formalen Definitionen immer bidirektionale Felder sind, können noch in Eingabe-, Ausgabe- und bidirektionale Parameterfelder unterschieden werden. Da diese Unterscheidung für die formale Definition und das Ablaufverhalten der Programme keinen Unterschied macht, ist diese in den formalen Betrachtungen nicht getroffen worden. Die Unterscheidung äußert sich in der Darstellung der Parameterfelder in der Virtuellen Umgebung, um die Parameterfelder danach zu kenn-

zeichnen, ob deren Werte hauptsächlich als Eingabeparameter oder Ausgabeparameter verwendet werden. Hier werden die bidirektionalen Felder wie in den formalen Betrachtungen am Rand der Programmknoten dargestellt. Die Parameterfelder, welche als Eingabeparameter definiert sind, werden wie die formalen Eingabefelder im oberen Bereich der Knoten dargestellt, aber im Gegensatz zu denen nicht auf dem Rand, sondern im inneren Bereich der Knoten. Ausgabe-Parameterfelder werden dementsprechend im unteren Bereich angezeigt. Abbildung 40 zeigt einen Programmknoten in der formalen Graphdarstellung auf der linken Seite und in der Darstellung in der virtuellen Umgebung. Die 5 Eingabefelder und 3 Ausgabefelder werden auch in der VR-Darstellung am oberen, bzw. unteren Rand des Knotens angezeigt. Die Parameterfelder, von denen zwei als Eingabeparameter, eines als Ausgabeparameter und eines als bidirektionaler Parameter definiert sind, werden in der Graphdarstellung alle auf dem rechten Rand des Programmknotens gruppiert. In der Darstellung in der VE unterteilen sie sich in die drei Gruppen: Für Eingabe- und Ausgabeparameter im oberen, bzw. unteren Bereich des Knotens, das bidirektionale Parameterfeld bleibt an der rechten Seite.

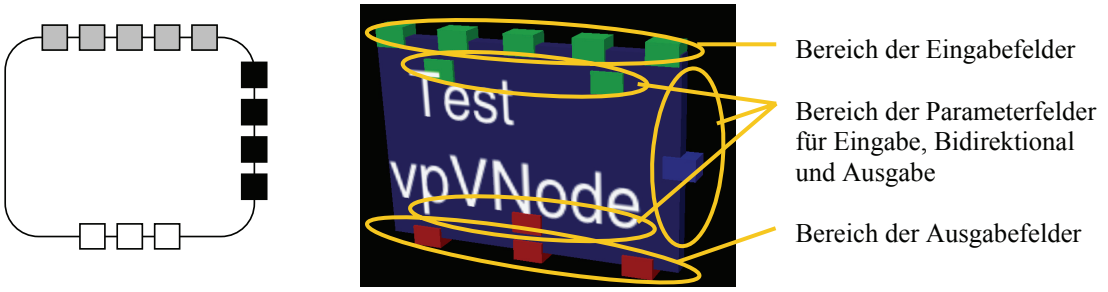


Abbildung 40: Formale Graphdarstellung eines Knotens und entsprechende Visualisierung in der VE

Zusätzlich werden der Name und der Typ als Text auf den Programmknoten und wahlweise auch den Feldern dargestellt. Verbindungen zwischen den Feldern visualisieren sich in Form von Pfeilen in der VE.

4.8.1 Interaktiver Aufbau der visuellen Programme

Die Verschaltung von Rechenknoten der visuellen Programme erfolgt entweder durch eine textuelle Beschreibung über ein XML-File, oder durch die Bearbeitung der visuellen Repräsentation in der Virtuellen Umgebung oder alternativ am Desktop.

Für die Grundoperationen an vorhandenen Programmknoten eignet sich am besten eine einfache und präzise Bearbeitung durch ein Eingabegerät, das über einen Zeigestrahl in der Virtuellen Umgebung Aktionen auslösen kann (auch „*Pointing Device*“ genannt). Ein Pointing Device wird in der Virtuellen Umgebung durch einen *Stylus* realisiert. Der Stylus ist ein Gerät, dessen Position und Orientierung durch ein Trackingsystem aufgenommen und zur Steuerung eines Zeigestrahls verwendet wird. An dieser Stelle wird noch keine auf natürliche Modalitäten basierende Interaktion mit den visuellen Programmen angestrebt, was den Einsatz von speziellen Eingabegeräten ja überflüssig machen könnte. Dies hat mehrere Gründe: Zum einem ist z.B. für die Selektion der Felder von Programmknoten eine präzise Steuerung des Auswahlgerätes nötig, was durch einen Stylus exakter realisiert werden kann als z.B. durch eine Zeigegeste. Zum anderen ist durch die relativ einfache Abbildung der Interaktion mit dem Stylus in der Virtuellen Umgebung zu einer Interaktion mit einer 2D-Maus eine

- natürlich etwas eingeschränkte - Bearbeitung der Programme auch in der Desktop-Umgebung möglich. Dadurch können einfache Testläufe und Anpassungen, welche keine Daten von Interaktionen aus der Virtuellen Umgebung erfordern, in Desktop-Umgebungen – welche in der Regel in einer höheren Anzahl von Arbeitsplätzen zur Verfügung stehen – durchgeführt werden. Außerdem soll die Gestenerkennung ja erst durch die Programme selber realisiert werden.

Um aber auch eventuell komfortablere Interaktionen bei der visuellen Programmierung realisieren zu können, ist es möglich, neben der vorhandenen Interaktionsmöglichkeit durch einen Stylus, auch eine Verschaltung von Programmen durch das Interaktions-Interface der Programmierung auszuführen. Damit kann das ganze System durch eine Art Bootstrapping-Verfahren durch sich selbst weiter entwickelt werden. Hierbei sollte die Interaktion durch den Stylus immer möglich gehalten werden, um einen Anker zu einer möglichen Interaktion zu haben, für den Fall, dass das neue Interaktions-Programm nicht (mehr) funktioniert.

Die Grundaktionen der visuellen Programmierung in der virtuellen Welt sind folgende:

- Erzeugen/Löschen von Instanzen aus einem Satz von Recheneinheiten
- Verschaltung und Trennung von Feldverbindungen der Recheneinheiten
- Setzen der konstanten Parameter der Recheneinheiten

Da die sprachliche Bezeichnung der unterschiedlichen Typen von Recheneinheiten problematisch ist und für die heutigen Spracherkenner ein fast unlösbares Problem darstellt, wird für die Auswahl der verschiedenen Recheneinheiten ein intuitiv zu bedienendes Auswahl-Tool zur Verfügung gestellt. Es besteht aus einem hierarchisch aufgebauten Rondell, welches mit Hilfe der Trackball-Funktion des eingesetzten Stylus gesteuert werden kann. Jede Hierarchieebene besteht aus einem Ring von unterschiedlichen Typen der Recheneinheiten. Die erste Ebene enthält eine grobe Unterteilung in die Unterklassen: Arithmetik, Speicher, Szenengraph, Constraint-Mediatoren und Visualisierung. Wird einer dieser Grundtypen durch die entsprechende Drehung mit Hilfe des Trackballs ausgewählt, erscheint die nächste höhere Auswahlebene. Im Falle der Arithmetik ist diese gegliedert in die von der entsprechenden Recheneinheit bearbeiteten Datentypen: Bool, Integer, Floating-point(Double), Vector, Quaternion, Segment und Matrix. Der ausgewählte Programmknoten wird darauf hin in einer speziellen Ablageleiste platziert. Durch das Etablieren einer Feldverbindung wird der neue Knoten dann schließlich in den Programmgraphen eingefügt und dort – nach einer weiter unten im Abschnitt angegebenen Heuristik – an eine passende Stelle gesetzt.

Für die Auswahl von vorhandenen Programmeinheiten zur Bearbeitung wird mit dem Stylus ein Zeigestrahl assoziiert, mit dem man die anvisierten Teile selektieren kann. Ein konstantes Liniensegment definiert die Zeigerichtung, welche durch die aktuelle Matrix des Stylus transformiert wird. Dieses Liniensegment dient als Eingabe eines speziellen Rechenknotens, der Objekte im Szenengraphen ermittelt, die möglichst dicht an dem gegebenen Segment liegen. Abbildung 41 zeigt das visuelle Programm zur Auswahl eines Programmknotens. Durch den „point-seg“-Knoten wird das konstante Liniensegment durch die Matrix des Stylus – bzw. in diesem Beispiel einer Desktop-Interaktion die Matrix eines Maus-Trackers – transformiert. Der point-map

Knoten nimmt dieses Segment als Eingabe und sucht über einen Traversal auf den Szenengraphen die Objekte innerhalb eines vorgegebenen Winkels in Richtung des Segments. Die Ausgabefelder des Knotens sind zum einem ein Feld mit einem eigenem Datentyp, der eine Liste von Knoten mit ihren Bewertungen enthält, zum anderen ein Feld, das direkt das beste Objekt bereitstellt. Dieses Objekt wird durch eine Feldverbindung an den Selektorknoten weitergegeben.

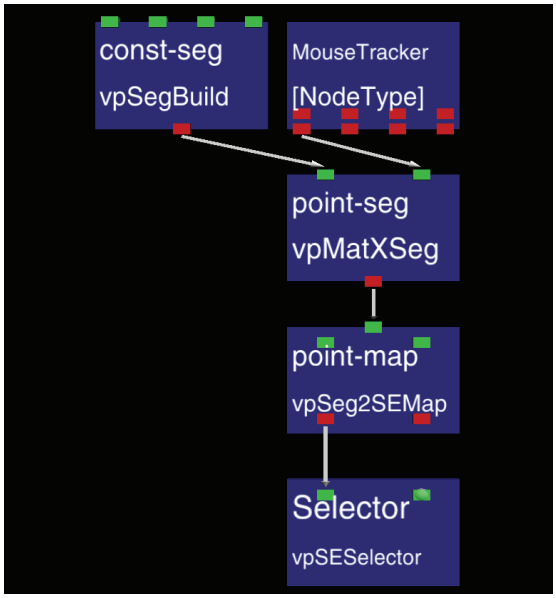


Abbildung 41: Visuelles Programm zur Auswahl von Knoten und Feldern

Gefunden werden hierbei spezielle Informations-Knoten („Semantic-Entities“, siehe Abschnitt 6.1) im Szenengraphen, welche die einzelnen Knoten und deren Felder in der Visualisierung der Rechenknoten repräsentieren. Da diese als Schnittstelle zu dem semantischen Wissen über die repräsentierten Einheiten dienen, kann über sie auf die Information für eine gewünschte Aktion zugegriffen werden. Da der Selektionsknoten auch den aktuell besten Knoten als Feld bereitstellt, kann dieser direkt an weitere Knoten zur Verarbeitung weitergegeben werden. Als erstes können Visualisierungsknoten den aktuell selektierten Knoten, bzw. das selektierte Feld darstellen, um die Interaktion für die präzise Auswahl zu erleichtern. Hier stehen mehrere Möglichkeiten zur Verfügung: Ein einfacher Visualisierungsknoten positioniert eine Kugelgeometrie im Zentrum des aktuell selektierten Knotens und zeigt damit dessen Position an. Die Art, Größe und Farbe der Geometrie des Selektors kann frei angepasst werden, um für die jeweilige Umgebung den besten visuellen Eindruck zu erzielen. Da bei komplexeren Programmen die Einzelheiten der Programmknoten aufgrund der begrenzten Auflösung der Darstellung nur schwer zu erkennen sind, kann ein weiterer Visualisierungsknoten den aktuellen Programmknoten in einer vergrößerten Darstellung zeigen.

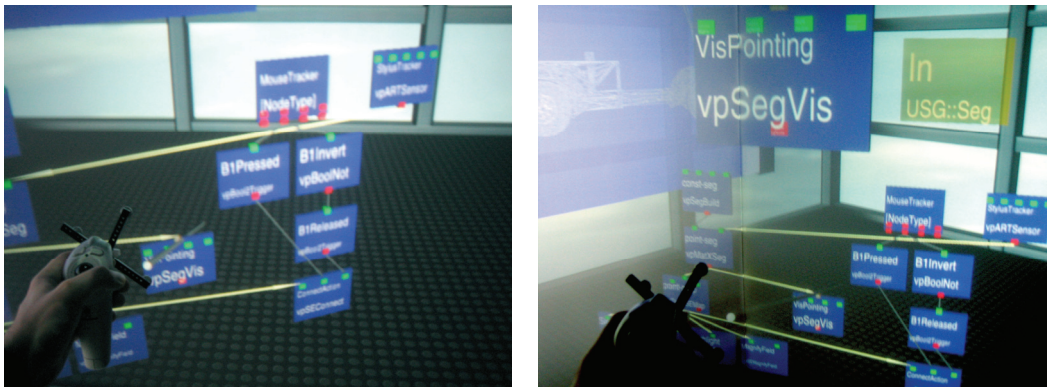


Abbildung 42: Selektion eines Feldes durch einen Stylus in der VE

Da für das Verbinden von Felder die Bezeichnungen und Typen gut lesbar sein müssen, werden selektierte Felder von einem anderen Knoten in einer noch größeren Darstellung gezeigt, wie in Abbildung 42 rechts zu sehen ist. Selektierte Felder können zusätzlich auch eingefärbt werden, um die aktuelle Auswahl anzuzeigen. Welche der Anzeigemöglichkeiten für die Interaktion genommen werden soll, kann einfach durch die Verbindung des Auswahlknotens zu den jeweiligen Visualisierungsknoten individuell konfiguriert werden.

Auch die Namen der Felder des Rechenknotens werden erst angezeigt, wenn der Knoten selektiert wurde. Dieses ist von Vorteil, da die vielen Textgeometrien bei komplexen Programmen für die Beschriftung der einzelnen Felder sehr viele Graphikprimitive erzeugen. Außerdem wird eine große Anzahl von Texten in der Virtuellen Umgebung auch schnell unübersichtlich.

Durch verschiedene Ereignisse können Programmknoten getriggert werden, welche die jeweilige Aktion z.B. als Scheme-Funktion oder auch direkt in der imperativen Programmiersprache implementieren. Aufgrund der langsameren Abarbeitung von Scripten im Vergleich zu kompilierten C++-Code sollten Scripting-Knoten nur für einzelne Events, wie z.B. einer Neukonfiguration im Programm oder Szenengraphen benutzt werden, und nicht für stetige Berechnungen im Datenfluss. Für diese Berechnungen können komfortabel über die XML-Deklaration der Rechenknoten hoch performante Rechenknoten erstellt werden (siehe Abschnitt 4.11). Da diese kompiliert und in das laufende Programm eingebunden werden müssen, können Scripting-Knoten aber in machen Fällen flexibler eingesetzt werden.

Das visuelle Programm in Abbildung 43 zeigt in der untersten Ebene einen Programmknoten, der die Verbindung von zwei Feldern erstellen kann. Dieser Knoten hat drei Eingangsfelder: Zum einen ein Eingangsfeld, das mit dem Auswahlknoten des Programms aus Abbildung 42 verbunden ist und an dem somit das aktuell selektierte Feld anliegt.

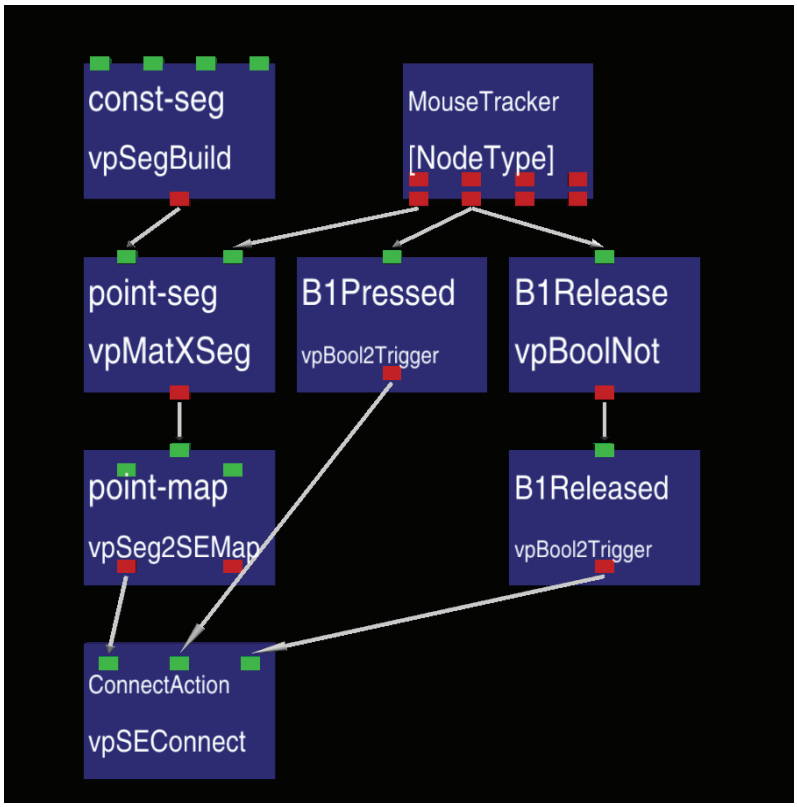


Abbildung 43: Visuelles Programm zur Verbindung von Feldern

Die beiden Triggerfelder dienen zum Auslösen der Aktion. Bei Aktivierung des ersten Feldes merkt sich der Verbindungsknoten das aktuell selektierte Feld als Startfeld der Verbindung. Dementsprechend wird das selektierte Feld zum Zeitpunkt der Aktivierung des zweiten Feldes als Zielfeld gespeichert. Gleichzeitig wird nach Überprüfung der Eigenschaften der Felder auf ihre formalen Anforderungen für eine Verbindung – kompatible Typen, Eingabe-/Ausgabefelder, bereits verbundene Felder (siehe auch Abschnitt 4.4.1) – die Feldverbindung etabliert. In dem Programm aus Abbildung 43 geschieht die Selektion des ersten Feldes beim Drücken einer Taste der Maus, bzw. in der Virtuellen Umgebung einer Taste des Stylus. Die Selektion des zweiten Feldes wird dann zusammen mit der Verbindungs-Aktion beim Loslassen der Taste ausgelöst.

Die Anordnung in der Visualisierung der Recheneinheiten in einem möglichst übersichtlichen Graphen zur Vermeidung von unübersichtlichen sich überlagernden Verbindungen (in (Ibrahim & Yoshizumi, 1999) auch „Visuelles Spaghetti“ genannt) erfolgt weitgehend automatisch.

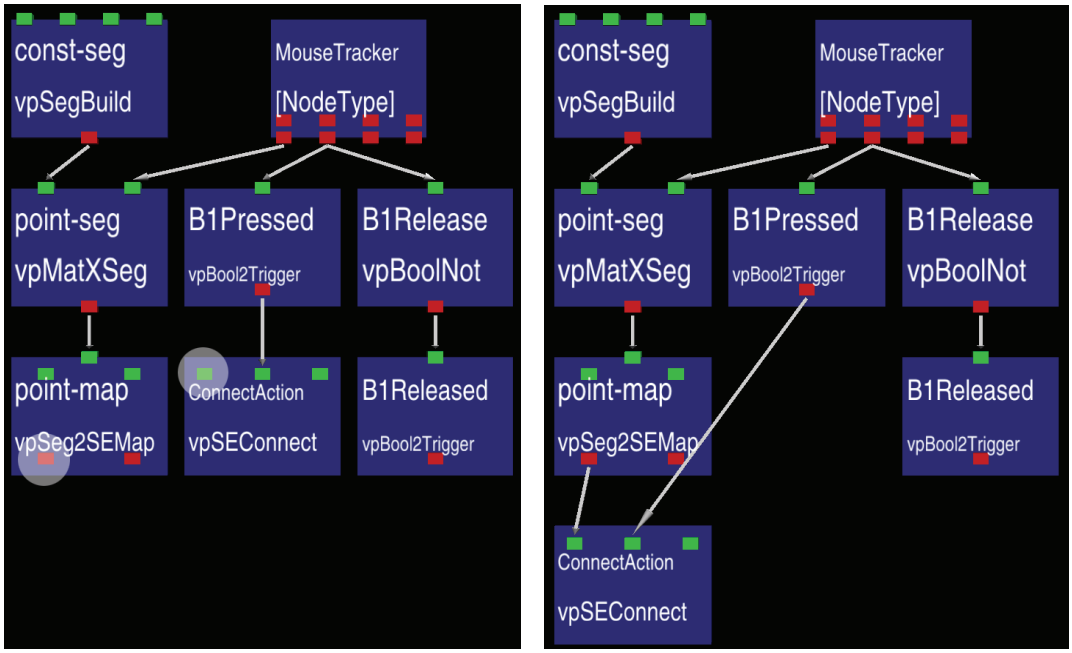


Abbildung 44: Automatische Anordnung der Programmknoten nach einer Feldverbindung

Damit möglichst keine rückwärtsgerichteten Verbindungen dargestellt werden müssen, ordnet sich ein verbundener Programmknoten immer unterhalb des Elternknotens an, der sich im Graphen in der untersten Ebene befindet. Abbildung 44 zeigt ein visuelles Programm vor und nach der Verbindung der beiden – auf der linken Seite der Abbildung hell unterlegten – Felder. Da die zwei zu verbindenden Knoten sich in derselben Ebene befinden, wird der Zielknoten der Verbindung neu unterhalb des Quellknoten eingehängt.

Indem die einzelnen Knoten im Graphen die maximale Breite – d.h. die maximale Anzahl der Kinder innerhalb einer Ebene – abschätzt und sich dem entsprechend viel Platz innerhalb seiner Ebene reserviert, werden die Knoten auch innerhalb einer Ebene möglichst übersichtlich angeordnet. Jede neue Verbindung von Feldern im Programm bewirkt eine Neuanordnung des Graphen im Bereich der Umgebung der beteiligten Programmknoten. Diese Art der Anordnung im Graphen ist aber auf eine azyklische Struktur angewiesen. Daher wird die Anordnung der Programmknoten in erster Linie anhand der Datenverbindungen berechnet. Die Steuerverbindungen, welche auch zyklische Strukturen erlauben, werden nur zur Anordnung der Knoten im Graphen benutzt, wenn sonst keine Datenverbindungen zu diesem Programmknoten bestehen.

Der Benutzer kann aber in dem visuellen Programm zu einzelnen Abschnitten navigieren und sich genauer zu betrachtende Abschnitte vergrößern. Insbesondere können in einen *Knotencontainer* (siehe folgender Abschnitt) gekapselte Abschnitte des visuellen Programms auf Wunsch dargestellt werden.

Dieser gekapselte Graph wird dann in einer weiter im Vordergrund liegenden visuellen Ebene angezeigt. Er schwebt damit vor der eigentlichen Programmierebene, welche durch die dreidimensionale Darstellung in der Virtuellen Umgebung gut visuell getrennt ist. Eventuell störende Verdeckungen können durch die Veränderung

des Blickwinkels – d.h. in der visuellen Umgebung einfach durch das Bewegen des Kopfes – beseitigt werden.

4.9 Knotencontainer

Da die Darstellung des gesamte Programm mit allen Programmknoten aber schnell unübersichtlich werden kann (siehe auch die Diskussion in Abschnitt 2.1.2), gibt es zusätzlich die Möglichkeit, Programmteile in eigenen Containern zusammenzufassen. Diese Container sind selber Programmknoten und ihre zugeordneten Felder dienen als Interface, mit denen Verbindungen von externen Feldern zu Feldern der internen Knoten oder Programmeinheiten erzeugt werden können. Die Container kapseln somit ganze Programmteile, welche als Einheit in bestehende Programme eingefügt und wiederverwertet werden können.

Definition 4.14 (**Knotencontainer**): Ein Programmknoten gehört zu der Menge der Knotencontainer (*NC*), wenn ihm durch spezielle Kanten: E^{NC}_{Assign} andere Programmknoten zugeordnet sind.

Hierbei sind:

- $E^{NC}_{Assign} \subseteq (P \times NC)$

  Kanten, die Programmknoten dem Knotencontainer zuordnen

Die Zuordnungsfunktion:

- $P_{intern} : NC \rightarrow P$

  $P_{intern}(nc) = \{p \in P \mid (p \times nc) \in E^{nc}_{Assign}\}$

  liefert die Menge aller einem Knotencontainer nc enthaltenen Programmknoten

Die zu den Knotencontainern zugehörigen Programmknoten werden durch spezielle Kanten im Programmgraphen zugeordnet. Um eine Kapselung der internen Programmknoten des Containers zu erreichen, müssen Feldverbindungen zwischen internen Feldern, die Programmknoten eines Containers zugeordnet sind, und Feldern, die externen Programmknoten zugeordnet sind, unterbunden werden. Dafür werden folgende Einschränkungen für Feldverbindungen von Feldern, welche zu den internen Knoten eines Containers gehören, definiert:

Definition 4.15 (**Containerfelder**):

Die Felder eines Knotencontainers nc werden anhand der folgenden Funktionen in interne und externe Felder des Containers unterschieden.

- $F_{intern} : NC \rightarrow F$

  $F_{intern}(nc) = \{f \in F \mid \exists p \in P_{intern}(nc) \quad (f,p) \in E \vee (p,f) \in E\}$

- $F_{extern} : NC \rightarrow F$

  $F_{extern}(nc) = \{f \in F \mid (f,nc) \in E \vee (nc,f) \in E\}$

Dann gilt für die Feldverbindungen:

$\forall f, f' \in F \quad (f,f') \in E \wedge f \in F_{intern}(nc) \Rightarrow f' \in F_{intern}(nc)$

Feldverbindungen von internen Feldern eines Knotencontainers sind nur zu internen Feldern *dieses* Containers erlaubt.

Zusätzlich gelte:

$\forall nc \in NC, f \in F_{intern}(nc), \; f' \in F_{extern}(nc) \quad (f',f) \in E \wedge (f,f') \notin E \Rightarrow (f \in F_{In} \wedge f' \in F_{In})$

$\forall nc \in NC, f \in F_{intern}(nc), \; f' \in F_{extern}(nc)$
$(f',f) \notin E \wedge (f,f') \in E \Rightarrow (f \in F_{Out} \wedge f' \in F_{Out})$

$\forall nc \in NC, f \in F_{intern}(nc), \; f' \in F_{extern}(nc) \quad (f',f) \in E \wedge (f,f') \in E \Rightarrow (f \in F_{IO} \wedge f' \in F_{IO})$

Während Feldverbindungen im visuellen Programm in der Regel von Ausgabefeldern zu Eingabefeldern oder zwischen bidirektionalen Feldern etabliert werden, bestimmt die letzte Einschränkung von Definition 4.15, dass externe Eingabefelder mit internen Eingabefeldern und interne Ausgabefeldern mit externen Ausgabefeldern des Containers verbunden werden. Dadurch bilden die externen Felder des Containers das Interface zu den Feldern der internen Programmknoten, wobei die Feldtypen der internen Felder durch die gleichen Feldtypen der externen Felder repräsentiert werden. Dieses ermöglicht die Einbindung der Knotencontainer im visuellen Programm auf dieselbe Art wie die normalen Programmknoten

Abbildung 45 zeigt die Graphdarstellung eines Knotencontainers mit seinen zugeordneten Programmknoten und Feldverbindungen.

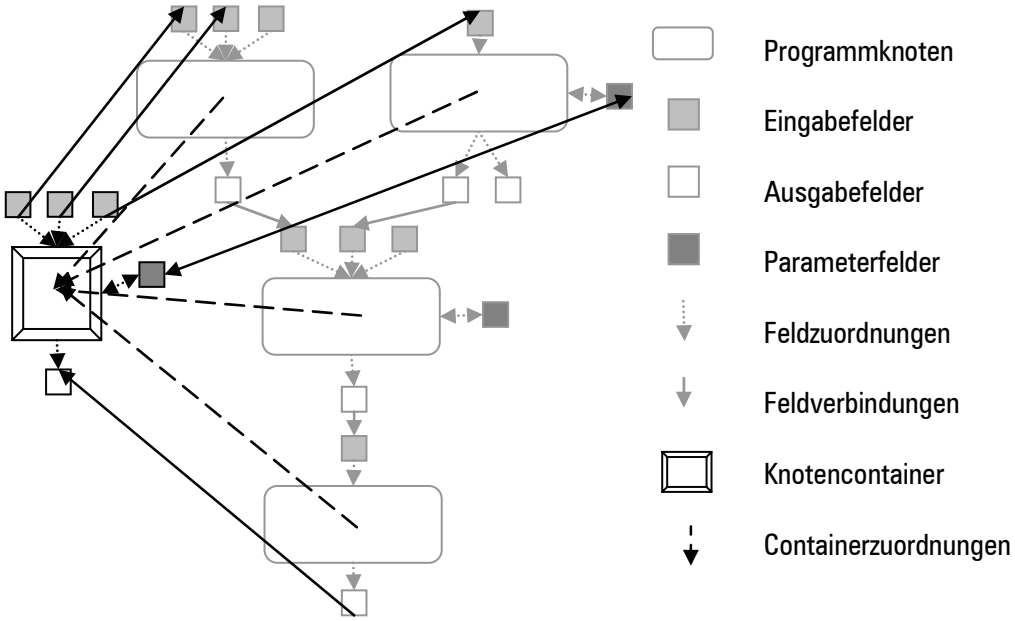


Abbildung 45: Graphdarstellung eines Knotencontainers

Die Darstellung der Knotencontainer als kompletter Graph ist offensichtlich nicht vorteilhaft für die Übersicht der visuellen Programme. Aber durch eine einfache Operation, einen Programmknoten, welcher einem Knotencontainer zugeordnet ist zusammen mit seinem Feldern im Inneren des Knotencontainers darzustellen, wobei die Zuordnungspfeile dann nicht mehr dargestellt werden, kann eine sehr übersichtliche Darstellung erzeugt werden. Abbildung 46 zeigt eine visuelle Grammatik, durch die diese Operation ausgeführt wird.

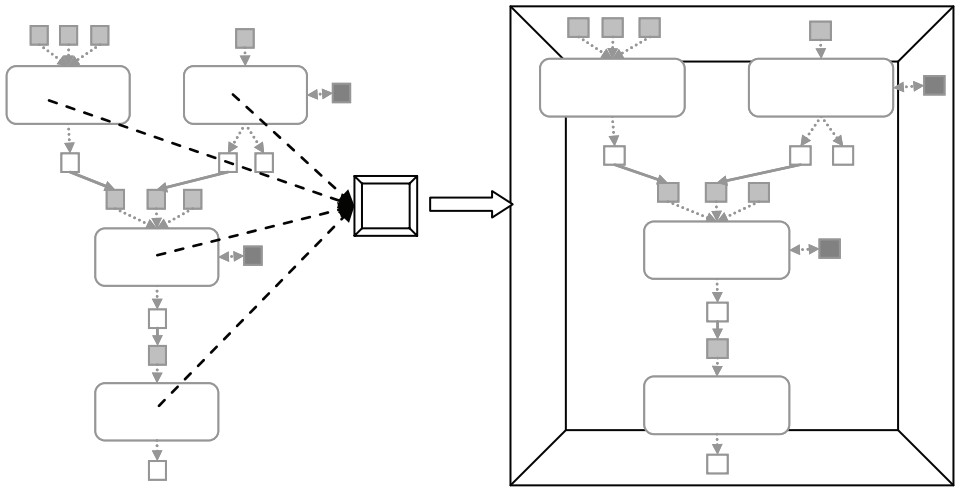


Abbildung 46: Visuelle Grammatik zur Vereinfachung von Knotencontainern

Zusammen mit den bisherigen Vereinfachungsregeln für Programmknoten (siehe Abbildung 32), welche entsprechend auf Knotencontainer angewendet werden können, ergibt sich für den Knotencontainer aus dem oben gezeigten Beispiel die in Abbildung 47 gezeigte, übersichtliche Darstellung:

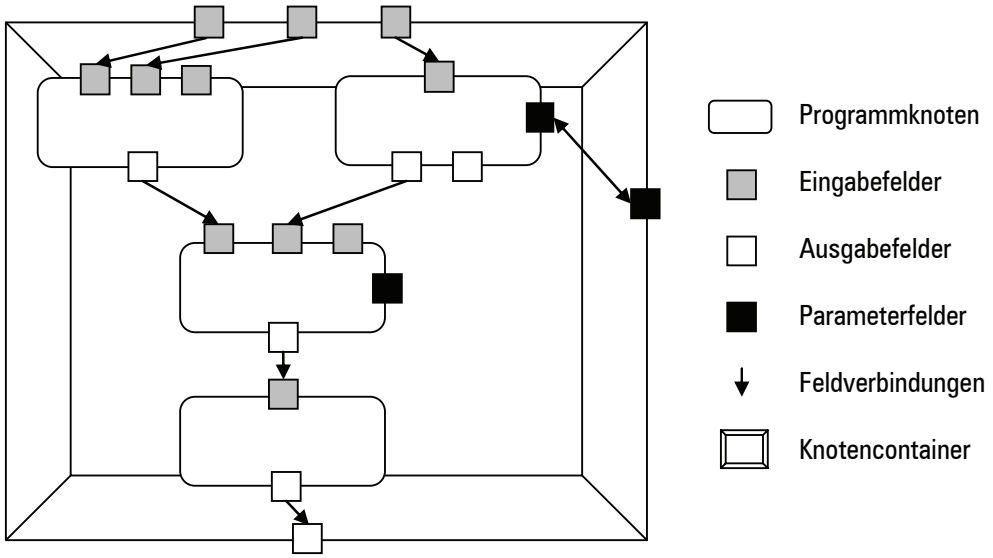


Abbildung 47: Vereinfachte Darstellung des Knotencontainers aus Abbildung 45

Bei der Verschaltung von Knotencontainer mit dem restlichen Programm ist die Sicht auf die internen Knoten nicht nötig und der Container kann in einer kondensierten Form ohne die internen Komponenten dargestellt werden.

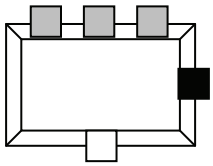


Abbildung 48: Kondensierte Darstellung des Knotencontainers aus Abbildung 47

In dieser kondensierten Ansicht (siehe Abbildung 48) ist ein Knotencontainer von einem normalen Programmknoten in der Interaktion nicht zu unterscheiden und kann auch genauso wie ein Programmknoten gehandhabt werden. Ein Unterschied ist, dass der Knotencontainer auf Wunsch wieder in seiner expandierten Form gezeigt werden kann, um z.B. die internen Komponenten zu editieren.

Es ergeben sich die folgenden Aktionen zur Behandlung von Programmcontainern:

- Erzeugen/Löschen eines Programmcontainers
- Erzeugen von Ein- und Ausgabefeldern eines Containers
- Einfügen einer Programmeinheit in einen Container
- Verschaltung und Trennung von Feldverbindungen von Containerfeldern

Die Knoten-Container leiten sich wie die atomaren Rechenknoten von der Basisklasse der Knoten ab. Da sich diese Container im Gegensatz zu den atomaren Rechenknoten dynamisch während der interaktiven Programmierung verändern und beliebig neue Ein- und Ausgabefelder definieren können, werden die für die Feldvisualisierungen benötigten Szenengraphknoten in dynamischen Listen verwaltet. Dieses ermöglicht das Hinzufügen von Feldvisualisierungen eines Programm-Knoten während der Interaktion.

Knotencontainer stellen die oben genannten Operationen zur Verfügung. Die Container selber sind auf die gleiche Weise wie die atomaren Knoten visualisiert und fügen sich daher direkt in die bestehende Interaktion für die visuelle Programmierung ein. Als zusätzliches Feature bieten die Container die Möglichkeit, ihren gekapselten Subgraphen anzuzeigen und zu bearbeiten.

4.10 Visualisierung der Datenströme

Vielfältige Visualisierungsmöglichkeiten erlauben dem Benutzer die genaue Kontrolle über den Datenstrom und der einzelnen Verarbeitungsschritte durch das Programm. Die Visualisierung des Datenflussprogramms in der Virtuellen Umgebung ermöglicht hierbei das gleichzeitige Testen (Erzeugung der Eingabedaten) und Überprüfen der Verarbeitung. Ein einfaches Interface ermöglicht zudem die Korrektur bzw. den Ausbau vorhandener Programme direkt in den VEs.

Eine erste Visualisierungsmöglichkeit zeigt ob, bzw. wie viele Datensamples pro Sekunde über eine Datenleitung fließen. Hierbei werden die Verbindungselemente entsprechend ihrem Datendurchsatz eingefärbt. Ein visuelles Programm zeigt seine aktiven Feldverbindungen als rot über gelb bis weiß glühend an. Je länger keine Daten über diese Verbindung fließen und je geringer die aktuelle Datenrate ist, desto mehr verändert sich die Farbe des Verbindungselements zu einer dunkleren Färbung. Wird ein Datenstrom z.B. durch die Verdeckung eines optischen Trackers unterbrochen, ist das für den Benutzer direkt anhand der scheinbar abkühlenden Datenleitungen sichtbar. In Abbildung 49 ist die Visualisierung im Falle eines unterbrochenen Datenstromes eines Tracker-Sensors dargestellt. Die zuerst hell weiß glühenden Datenverbindungen (Abbildung 49 links) werden nach kurzer Zeit orange (Mitte) und dann nur noch schwach rot glühend (im Bild rechts zu sehen). Ist der Datenstrom lange Zeit nicht aktiv, wie in dem Beispiel der Programmstrang rechts im Bild, werden die Verbindungen dunkelgrau dargestellt.

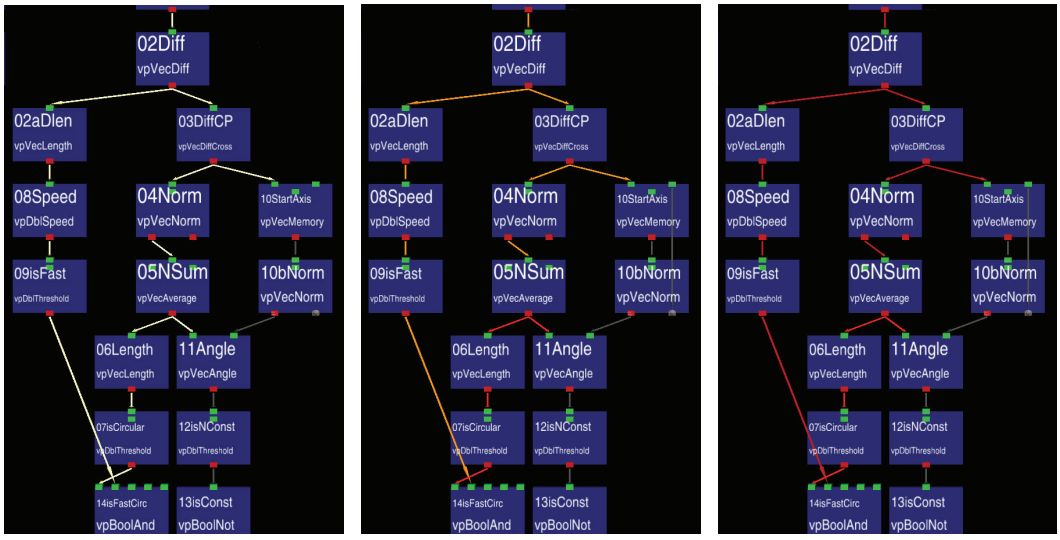


Abbildung 49: Visualisierung des Datendurchsatzes im Programm

Diese Visualisierung erlaubt dem Benutzer auf einem Blick, die genaue Stelle einer Unterbrechung des Datenstromes zu erkennen. Unterbrechungen des Datenstromes sind in der Datenflussprogrammierung eine häufige Fehlerursache und ohne visuelle

Unterstützung oft nur mit viel Aufwand zu finden. Neben der Unterbrechung von Datenströmen können mit Hilfe dieser Visualisierung auch Programmteile entdeckt werden, welche einen sehr hohen Datendurchsatz haben, der je nach Einsatz der berechneten Daten nicht in dieser Genauigkeit vorliegen muss. Dementsprechend können diese mit einer niedrigeren Datenrate betrieben werden, um die Rechenzeit des Programms zu optimieren.

Die zweite Stufe der Visualisierungsmöglichkeiten zeigt eine grobe Darstellung der Werte, welche aktuell an den Feldern der Programmeinheiten anliegen. Diese wird angezeigt, wenn das zu überprüfende Feld, während der Interaktion (siehe Abschnitt 4.8) selektiert wird. Abbildung 50 zeigt neben den bisherigen Visualisierungen des aktuell selektierten Knoten und dem aktuell selektierten Feld (im Bild oben links, hell unterlegt), eine Visualisierung des aktuellen Feldwertes, in diesem Fall ein Richtungsvektor in Form eines Pfeils.

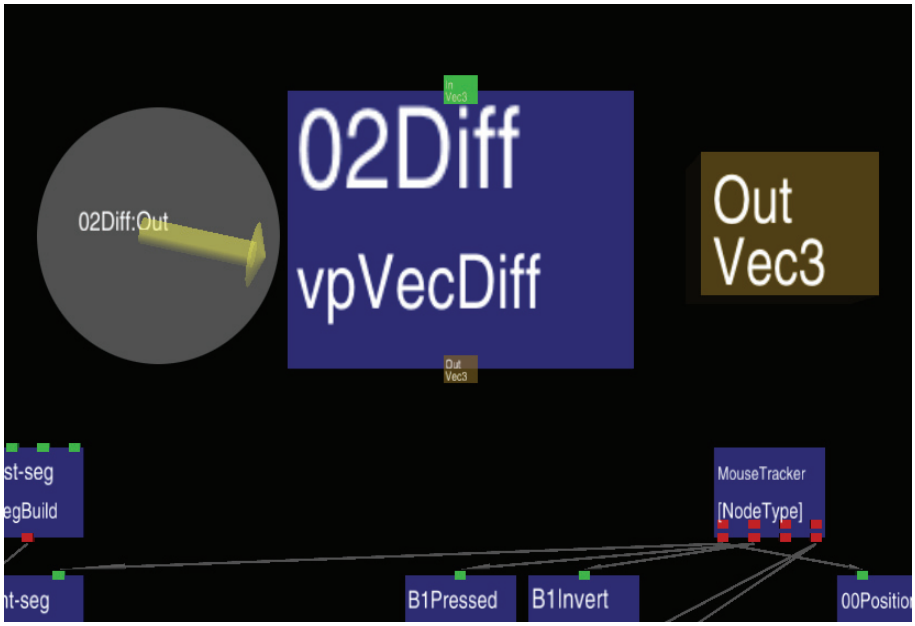


Abbildung 50: Visualisierung einfacher Feldwerte

Hier werden nur grobe Wertebereiche visualisiert und nur einfache Werttypen angezeigt. Zur Darstellung auf dieser Ebene der Visualisierung eignen sich z.B. boolesche Werte und skalare Werte in einem definierten Wertebereich. Durch die dreidimensionale Visualisierung in der Virtuellen Umgebung können aber z.B. auch Richtungsvektoren angezeigt werden.

Für komplexe Datentypen oder genauere Verläufe der Werte können einzelne Felder mit speziellen Visualisierungseinheiten verbunden werden. Für skalare Werte stehen hier Visualisierungen als Balkengraphik oder für die genauere Überprüfung des zeitlichen Verlaufes Graphplotter zur Verfügung.

Feldwerte mit Positionsangaben können dem Benutzer direkt als Objekt an der Position in der virtuellen Welt visualisiert werden. Segmente im Raum, wie z.B. der Zeigestrahl vom Zeigefinger eines Benutzers können auch direkt während der Interaktion mit dem System zusätzlich eingeblendet werden, um die Genauigkeit der getrackten bzw. berechneten Daten überprüfen zu können. Auch die Position und

Orientierung von Koordinatensystemen, welche im Programmfluss als Typ 4x4-Matrix auftauchen kann damit direkt in der virtuellen Welt dargestellt werden.

Die Darstellung von Richtungsvektoren, Positionen und Koordinatensystemen auf einem zweidimensionalen Ausgabegerät ohne die Verortung im virtuellen Raum enthält nur einen erheblich geringeren Informationsgehalt, bzw. ist wesentlich schwerer zu interpretieren, als die direkte Darstellung in der virtuellen Welt des Benutzers.

Zudem können die entsprechenden Visualisierungen direkt in der Testumgebung zugeschaltet werden, während der Benutzer mit dem System interagiert. Die Interaktion, Fehlersuche und Fehlerbehebung sind somit fast gleichzeitig und direkt in der Virtuellen Umgebung möglich und beschleunigen die Programmierung von robusten Programmen für den Interaktionsablauf innerhalb dieser Umgebung um ein Vielfaches. Reaktive Programmierumgebungen mit direkter Manipulation und Visualisierung wurden auch als vorteilhaft bei der Fehlersuche bestätigt (siehe Abschnitt 2.1).

4.11 Definition und Erweiterbarkeit der Programmknoten

Eine Voraussetzung bei der Programmierung mit visuellen Systemen ist eine umfangreiche Menge von Programmierbausteinen, welche zu visuellen Programmen verschaltet werden. Daher gilt für visuelle Datenfluss-Programmiersprachen (DFVPL) folgende Feststellung:

> Eine DFVPL ist nur so gut, wie der Satz der vorhandenen Programmbausteine.

Da aber kein vorgefertigter Satz von Programmbausteinen für alle möglichen Programmieraufgaben passend sein kann, sollte daher eher gelten:

> Eine DFVPL ist nur so gut, wie der Satz der vorhandenen Programmbausteine und einer komfortablen Möglichkeit, diesen Satz von Bausteinen beliebig zu erweitern.

Gerade bei grobkörnigen DFVPLs ist eine die Erweiterung des Satzes von Rechenknoten entscheidend, da die einzelnen Knoten oftmals schon spezielle Berechnungen implementieren. Wenn ein neuer Programmknoten dann nur aufwändig zu erstellen oder zu erweitern ist, ist es schwierig, die Programmiersprache auf verschiedene Einsatzgebiete anzupassen. Außerdem ist ein Satz von Rechenknoten, der durch verschiedene Programmiergruppen ständig verbessert und erweitert wird, ein Vorteil für alle Anwender der visuellen Programmierumgebung, wobei der Einsatz nicht selbst programmierter Rechenknoten durch die Kapselung und eindeutige Schnittstelle im visuellen Programmiersystem besonders einfach möglich ist.

Das innerhalb dieser Arbeit entwickelte XML-Fileformat für die Definition von Programmknoten („Visual Interactive Programming Markup Language“, VIPML) bietet dem Programmierer eine komfortable Möglichkeit, neue Knoten zusammen mit ihren Eingabe- und Ausgabefeldern zu definieren. Der sequentielle Code für die Berechnungsvorschriften der Programmknoten wird dabei direkt in C++ angegeben. Für die Schnittstelle zwischen der Berechnungsvorschrift und den definierten Feldern des Knotens werden vordefinierte Variablen und Funktionen zum Wertezugriff und dem Erzeugen von Datentoken auf den Ausgabefeldern bereitgestellt. Operationen auf den komplexen Datentypen und Veränderungen in der Szenengraphstruktur, werden durch den USG-Layer (siehe Abschnitt 4.12) gekapselt. Daher ist der C-Code für den

Einsatz in verschiedenen VR-Toolkits kompatibel. Somit können auch die Programmknoten in unterschiedlichen Systemen eingesetzt werden.

Die Eingabekanäle der Eingabefelder werden automatisch in Datenströmen (siehe Abschnitt 4.4.4) zusammengefasst. Diese Datenströme legen anhand der Datentoken und Prioritätsinformationen in den Datenkanälen die Zeitpunkte für die Neuberechnungen der Daten fest. Die Berechnungsvorschrift der einzelnen Knoten gibt nur die Verrechnung eines Datensatzes zu einem Zeitpunkt an. Im Falle von Datenkanälen mit dicht gefüllten Datensequenzen wird bei dem entstandenen Programmknoten automatisch dafür gesorgt, dass diese Berechnung für jeden vom Datenstrom vorgegebenen Zeitpunkt ausgeführt wird.

4.11.1 XML-Format der Programmknoten

Das XML-Format für die Definition der Programmknoten (*VIPML*) wurde entwickelt, um eine abstrakte, deklarative Beschreibungsform für Berechnungsknoten der visuellen Datenflussprogrammierung zur Verfügung zu stellen. Sie abstrahiert die konkrete Implementation der Felder, der Callback-Aufrufe und der Variablendeklarationen.

Tag	Sub-Tags	Attribute [Wert]	Body
VPNode	Fields	name [String]	-
	OnComputeValue	base-class [C++ class name]	
	OnInit	visual [“true”|“false”]	
	OnFrame	sensor [“true”|“false”]	
	OnFieldChanged		
	AddInclude		
	AddVariable		
	AddDeclaration		
	AddCode		

Das Tag „VPNode“ ist die Hauptkomponente einer Programmknotendefinition. Es enthält alle Subtags für die Definition der Felder, Rechencallbacks, Variablen und zusätzlichen C++-Code für die neue Klassendefinition des Knotens. Der Name des Knotentyps wird im „name“-Attribut angegeben. Da dadurch gleichzeitig der Name der neuen C++-Klasse angegeben wird, muss er immer vorhanden und eindeutig sein. Soll der Knoten von einem vorhandenen Knotentyp abgeleitet werden, wodurch er dessen Felder, Variablen und Callbacks übernimmt, kann der Name der Basisklasse im „base-class“-Attribut angegeben werden. Da manche Programmknoten nicht direkt in den visuellen Programmen verbaut werden, brauchen diese keine Geometrien und Informationsknoten für den Szenegraph anzulegen. Diese Eigenschaft wird dadurch

definiert, dass das „visual“-Atrribut auf „false“ gesetzt wird. Ist der Programmknoten ein Sensor, wird das „sensor“-Attribut auf „true“ gesetzt. Das hat zur Folge, dass er in jedem Berechnungsframe evaluiert wird, auch ohne dass ein Feldwert sich ändert.

Tag	Sub-Tags	Attribute	Body
Fields	Input Output Bidirect	-	-

Im „Field“-Tag werden die Felder des Knotens deklariert. Sie werden nach Eingabefeldern, Ausgabefeldern und Feldern unterschieden, die sowohl Daten lesen, als auch Datentoken erzeugen können.

Tag	Sub-Tags	Attribute	Body
Input|Output	Stream Value Trigger	-	-

Die Eingabe- und Ausgabefelder werden in drei Typen unterschieden: Mit dem „Stream“-Tag werden Eingabe und Ausgabefelder mit Datensequenztypen deklariert, während das „Value“-Tag für die normalen Feldtypen benutzt wird. Dadurch entstehen Parameterfelder, welche nur genau einen Wert pro Berechnungsschritt enthalten. Außerdem können noch Triggerfelder über das „Trigger“-Tag definiert werden. An dieser Stelle weichen die Deklarationsmöglichkeiten von der formalen Definition, wie sie in Abschnitt 4.4.1 gegeben wurde ab. Nach der formalen Definition können einfache Felder mit einfachen Werten ohne Datensequenzen nur bidirektionale Felder sein. Die zusätzliche Unterscheidung dieser Felder in die drei Klassen dient hier nur der besseren Dokumentation der Feldeigenschaften und der übersichtlicheren Darstellung später im visuellen Programm (siehe Abschnitt 4.8). Für das Ablaufschema der Programmknoten gelten alle „Value“-Felder als bidirektional.

Tag	Sub-Tags	Attribute	Body
Bidirect	Value Trigger	-	-

Da nach der formalen Definition alle Felder mit Datensequenzen eindeutig als Eingabefeld bzw. Ausgabefeld definiert sein müssen, tauchen bei der Deklaration der bidirektionalen Felder nur „Value“ und „Trigger“-Felder auf.

Tag	Sub-Tags	Attribute	Body
Stream	-	**name** [String] **type** [Datentyp] **init** [Wert vom Datentyp]	-

Das "Stream"-Tag deklariert ein Feld mit einer Datensequenz als Typ. Name und Typ des Feldes bzw. der Datensequenz können in den entsprechenden Attributen angegeben werden. Als Wert des Feldes wird eine Datensequenz im Const-Modus (siehe Abschnitt 4.4.5) erzeugt. Der initiale Wert dieser konstanten Datensequenz muss durch das „init"-Attribut angegeben werden.

Tag	Sub-Tags	Attribute	Body
Value	-	**name** [String] **type** [Datentyp] **init** [Wert vom Datentyp] mode [**"direct"**|"link"|"multi"]	-

Die Deklaration im "Value"-Tag ist ähnlich zu der Deklaration der Datensequenzfelder. Erzeugt wird in diesem Fall ein Feld mit einem Feldwert vom angegebenen Datentyp. Für die Erzeugung von Feldern mit Zeigerstrukturen – für die Übergabe von Objekten mit größeren Speicherbedarf als Referenz – und Multifeldern, welche mehrere Objekte vom angegebenen Datentyp in Listen speichern können – kann zusätzlich das „mode"-Attribut auf „link" bzw. „multi" gesetzt werden. Um die Speicherverwaltung bei den über Zeiger referenzierten Objekten zu automatisieren, werden als Zeiger *smart-pointer* (Edelson & Pohl, 1991) eingesetzt, welche das referenzierte Objekt automatisch freigeben, wenn es keine Referenz auf dieses Objekt mehr gibt.

Tag	Sub-Tags	Attribute	Body
Trigger	-	**name** [String]	-

Da Triggerfelder keine Werte enthalten entfallen hier alle Attribute außer dem Namen des Feldes.

Tag	Sub-Tags	Attribute	Body
OnComputeValue	-	-	**C++-Code** zur Berechnung eines Datensatzes

Der „OnComputeValue"-Tag enthält den auszuführenden C++-Code zur Berechnung eines Datentoken im Body. Da der Code während eines Berechnungsframes mehrmals aufgerufen werden kann und die Werte der Datensequenzfelder immer die – abhängig von dem Berechnungsmodus eventuell interpolierten – Werte der Daten-

sequenz zu dem aktuellen Berechnungszeitpunkt liefern sollen, werden die aktuellen Werte für die Berechnung in vordefinierten Variablen zur Verfügung gestellt. Diese Art des Wertzugriffs auf die Felder abstrahiert auch davon, ob es sich um ein Datensequenzfeld, oder ein Feld mit einfachen Datentypen handelt. Zusätzlich wird der Zeitpunkt angegeben, für den die aktuelle Berechnung durchgeführt wird. Eine weitere Variable enthält die Information, ob sich der Feldwert während der letzten Berechnung verändert hat. Letzteres kann z.B. dafür benutzt werden, zu überprüfen, ob ein spezieller Programmteil überhaupt neu berechnet werden muss. Die folgende Tabelle 3 zeigt die vordefinierten Variablen für den Berechnungs-Callback.

[Feldname]_Value	Aktuelles Datentoken aus dem Datenstrom eines Datensequenzfeldes bzw. aktueller Feldwert eines einfachen Feldes
[Feldname]_Changed	boolscher Wert, ob das entsprechende Feld neue Daten enthält
ComputeTime	Zeitpunkt, für den die neuen Datentoken berechnet werden

Tabelle 3: Vordefinierte Variablen für den Berechnungs-Callback

Auch das Setzen der Feldwerte bei den unterschiedlichen Feldarten hat eine davon abhängige Semantik. Während bei den einfachen Feldern der Wert direkt als Feldwert übernommen werden kann, wird bei den Datensequenzfeldern, die ja alle Daten eines Frames enthalten, ein zusätzliches Datentoken in die enthaltene Datensequenz geschrieben. Um dem Programmierer die unterschiedliche Behandlung der Ausgabefelder abzunehmen, wird diese Funktionalität durch eine vordefinierte Funktion gekapselt. Tabelle 4 zeigt die vordefinierte Funktion mit der das berechnete Datentoken an ein Ausgabefeld gesendet werden kann.

Set_[Feldname] ([Wert])	hängt das im Argument angegebene Datentoken an die Datensequenz des Ausgabefeldes, bzw. setzt den Feldwert eines einfachen Feldes

Tabelle 4: Vordefinierte Ausgabefunktion für den Berechnungs-Callback

Tag	Sub-Tags	Attribute	Body
OnInit	-	-	**C++-Code** für Initialisierung des Containers

Für die Initialisierung von Variablen, oder für das Aufbauen von Szenengraphabschnitten für eine Visualisierung steht der „OnInit“-Callback zur Verfügung.

Tag	Sub-Tags	Attribute	Body
OnFrame	-	at [“start”|**”end”**]	**C++-Code** zur Ausführung einmal pro Render-Frame (vor bzw. nach den Berechnungsschritten)

Der „OnFrame“-Callback enthält C++-Code, der in jedem Frame einmal, unabhängig von der Berechnung der Datentoken ausgeführt werden soll. Bei normalen Re-

chenknoten setzt das voraus, dass sich überhaupt Feldwerte des Knotens geändert haben. Sind Knoten durch das „sensor“-Attribut als Sensorknoten deklariert, wird dieser Callback genau einmal pro Berechnungsframe aufgerufen. Werden zusätzlich noch Werte im „OnComputeValue“-Callback berechnet, kann durch das „at“-attibute angegeben werden, ob der Code vor oder nach der Berechnung der Feldwerte ausgeführt wird.

Tag	Sub-Tags	Attribute	Body
OnFieldChanged	-	field [Feldname|**“any“**]	**C++-Code** zur Ausführung *direkt* bei einer Feldwertänderung

Manchmal sollen Operationen direkt bei der Änderung eines Feldwertes ausgeführt werden. Im Body des „OnFieldChanged“ Callback kann der C++-Code für diese Operationen angeben werden. Das Attribut „field“ gibt dabei das Feld an, dessen Änderung die Ausführung des Codes auslösen soll. An dieser Stelle dürfen im C++-Code des Bodys keine der vordefinierten Variablen oder Methoden für die Feldwerte benutzt werden, da die Eingabewerte evtl. noch nicht definiert sind, und das Setzen der Ausgabefelder zu undefinierten Verhalten und Endlosschleifen führen kann, da der Code hier direkt, unter Umgehung des Ablaufschemas der Rechenknoten evaluiert wird.

Tag	Sub-Tags	Attribute	Body
AddInclude	-	file search [**“local”**|“system”]	C++-Code für Includes

Durch das “AddInclude”-Tag können direkt externe C++-Header-Files eingebunden werden. Die Includes können wahlweise auch direkt als C++-Code im Body angegeben werden. Dieser Code wird dann direkt oben im Header-File der Klasse des neuen Knoten platziert.

Tag	Sub-Tags	Attribute	Body
AddDeclaration	-	sight [**“private”**,“public”, “protected”]	Code zur Definition von Klassen-Variablen und Methoden

Das „AddDeclaration”-Tag erlaubt die Deklaration von Klassenvariablen und Methoden für den neuen Knoten und platziert den C++-Code aus dem Body in der Klassendeklaration. Das „sight“-Attribut bestimmt hierbei die Sichtbarkeit der lokalen Definitionen.

Tag	Sub-Tags	Attribute	Body
AddCode	-	-	Zusätzlicher C++-Code für Hilfs-Funktionen, Methoden etc.

Zusätzlicher Code z.B. für Hilfsfunktionen oder Klassenmethoden wird im „AddCode“-Tag angegeben.

Tag	Sub-Tags	Attribute	Body
AddVariable	-	name [Variablenname] type [Typ] init [initialer Wert] sight [“private”,“public”, “protected”] mode [“direct”, “link”]	-

Das Tag „AddVariable“ dient nur zu der Vereinfachung und kürzeren Schreibweise bei der Programmierung. Die Variable kann auch mit ihrem Namen und Typ aus den jeweiligen Attribut im „AddDeclaration“-Tag definiert und im Code des „OnInit“-Tags mit dem Wert des „init“-Attributes initialisiert werden.

Beispiel

Als Beispiel die Definition eines einfachen Rechenknotens zur Berechnung einer Addition von zwei Integer-Werten.

```
< VPNode name=„vpIntAdd“ >
  < Fields >
    < Input >
      < Channel name=„In1“ type=„int“ init=„0“ />
      < Channel name=„In2“ type=„int“ init=„0“ />
    </ Input >
    < Output >
      < Channel name=„Out“ type=„int“ init=„0“ />
    </ Output >
  </ Fields >
  < OnComputeValue >
     int result = In1_Value + In2_Value;
     set_Out (result);
  </ OnComputeValue >
</ VPNode >
```

Dieser Rechenknoten enthält zwei Eingabekanäle, welche bei der Übersetzung der XML-Beschreibung in einem Eingabestrom zusammengefasst werden. Dieser bestimmt aus den Attributen der beiden Kanäle die gewünschte Taktung mit der die Daten in dem Knoten berechnet werden sollen. Falls ein oder mehrere Zeitsamples innerhalb des aktuell zu berechnenden Frames liegen, wird für jeden Sample einmal die ‚**OnComputeValue**’-Routine aufgerufen. Vor dem Aufruf werden die Datenvariablen, hier ‚**In1_Value**’ und ‚**In2_Value**’ auf die Werte gesetzt, die zu dem zu berechenden Zeitpunkt in den jeweiligen Datenkanälen gesetzt sind. In diesem Beispiel wird der Datenkanal an dem Feld ‚In1’ als Master angenommen. Daher werden die Werte und Zeiten direkt für die Berechnung übernommen. Die Werte für den zweiten Kanal werden hingegen je nach Datentyp im Datenkanal an den vorge-

gebenen Zeitpunkten interpoliert, bzw. im Sequence- und Immediate-Modus bis zum Ende des Frames linear extrapoliert oder als konstant angenommen.

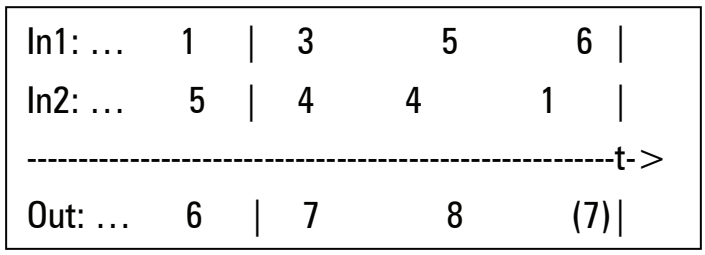


Abbildung 51: Beispielverlauf von Datentoken im Datenstrom

Abbildung 51 zeigt einen möglichen Verlauf der Datensamples im Ein- und Ausgabedatenstrom der Recheneinheit. In diesem Beispiel wurden die Zahlenwerte im Slave-Kanal (In2) interpoliert und das Ergebnis danach wieder auf ganze Zahlen gerundet. Der letzte Wert des Ausgabestromes würde im Precise-Modus erst im nächsten Frame über die Interpolation mit dem nächsten Datentoken im zweiten Eingabekanal berechnet. Im Immediate-Modus hingegen ist das der einzige Wert, der in diesem Frame berechnet würde, da er das aktuellste berechenbare Ergebnis darstellt. Die nicht benötigten Zwischenergebnisse werden in diesen Fall gar nicht erst berechnet.

4.12 Der universelle Szenengraph-Abstraktions-Layer

Der Code im OnCompute-Callback ist in C++ geschrieben und sollte sich weitgehend an den Programmierstandard von C++ und generischen Erweiterungen wie z.B. die Standard-Template-Library, STL (Musser & Saini, 1996) halten. In dem Bereich der Programmierung von Szenengraphen ist das leider (noch) nicht möglich. Hier muss sowohl bei Rechenoperationen der – für Szenengraphen typische – Datentypen, als auch bei der Manipulation der Szenengraphen selber, auf eine proprietäre Lösung zurückgegriffen werden. Leider hat sich bei der Fülle von Szenengraphtools für das Programmierinterface bisher kein Standard durchsetzen können. Zusätzlich scheinen regelmäßig neue Szenengraphbibliotheken entworfen zu werden, während bisher scheinbar etablierte Tools nicht mehr problemlos eingesetzt werden können. Die Gründe für das Wegfallen vorhandener Szenengraph-Tools sind vielfältig. Zum einen kann es sein, dass kommerzielle Produkte nicht mehr weiter supported werden, da der Hersteller nicht mehr existiert oder die Unterstützung des Produktes sich wirtschaftlich nicht mehr lohnt. Auch bei Open-Source-Projekten kann die Weiterentwicklung ausbleiben, wenn z.B. die Kernentwickler das Interesse an der Software verlieren, oder ein neueres Projekt die Fähigkeiten des aktuellen Tools abdeckt und die Entwicklungsarbeit dann in das neue Produkt fließt. In allen Fällen wo die Weiterentwicklung eines Produktes ausbleibt, kann durch die Weiterentwicklung von notwendigen Programmpaketen oder Umstellung von Compilern oder Entwicklungspaketen der weitere Einsatz der Software schnell unmöglich werden bzw. im Falle von Open-Source-Entwicklungen viel Eigenleistung erfordern. Da das visuelle Programmiersystem für den Einsatz in Virtuellen Umgebungen zwingend auf ein Szenengraphtool angewiesen ist, aber nur einen Basissatz von Rechenoperationen und Szenengraph-Manipulationen braucht, wurde ein abstrakter C++-Layer entwickelt, der diese Grundfunktionalitäten kapselt.

Hierbei ist darauf hinzuweisen, dass der Layer keine eigene generalisierende Implementation eines Szenengraphs für verschiedene Render-Engines ist, wie sie z.B. in (Döllner & Hinrichs, 2000) realisiert wurde. Stattdessen soll durch ein komfortables Programmierinterface die Basisfunktionalität von verschiedenen, vorhandenen Szenengraph-APIs generalisiert werden. Alle Operationen des generalisierenden Layers werden auf die Funktionalität des darunter liegenden, aktuell eingesetzten Szenengraph-Tools umgesetzt.

Um die in Abschnitt 4.5 beschriebenen Rechenknoten allgemein für unterschiedliche SG-Tools einsetzen zu können, muss der Code zur Verrechnung der Eingangsdaten mit einer abstrakten Schicht – sowohl für die Datentypen, als auch für die auftretenden Operationen – versehen werden. Da die verschiedenen Tools für VR-Umgebungen (Performer, OpenSG, …) mit unterschiedlichen Datentypen rechnen, muss das Programmiersystem die Möglichkeit besitzen, diese verschiedenen Datenformate zu kapseln.

Für die Anbindung heterogener Applikationsteile in VR-Programmierumgebungen ist es sinnvoll ein Framework zu haben, das die jeweiligen Datenformate ineinander konvertieren kann (vgl. (Heumer *et al.*, 2005)). Bei dem Einsatz einer Programmierumgebung, welche sehr eng mit dem entsprechenden Szenengraphen-Tool verbunden ist, und die Berechnungen der (Benutzer-) Eingabedaten zur direkten Auswertung und Visualisierung in der Virtuellen Umgebung mit hoher Performanz berechnet werden müssen, ist dieser Weg nicht vorzuziehen. Im Gegensatz zu den extern angebundenen Applikationsteilen müsste in diesem Fall über die Schnittstelle eine hohe Menge an Daten ausgetauscht und konvertiert werden. Die externe Anbindung der Programmierknoten mit eigenen Datentypen würde außerdem eine eher künstliche Trennung von den Programmierknoten und den internen Knoten des eingesetzten VR-Tools schaffen.

Um die Konvertierung der Daten, die je nach Programm an vielen Stellen im Programmablauf zwischen der Programmberechnung und dem Szenengraphen ausgetauscht werden müssen, komplett zu vermeiden, wird ein übergeordneter Abstraktions-Layer eingeführt, der die üblichen Grundoperationen der verschiedenen VR-Tools abbilden kann. In diesem Layer kann auch auf eventuelle Unterschiede der einzelnen Tools eingegangen werden. So benutzen z.B. einige Szenengraphentools Bounding-Spheres als Clusterinformationen für die Ausdehnung der Geometrie der einzelnen Knoten im Szenengraphen, während andere hingegen standardmäßig Bounding-Boxen benutzen. Der Zugriff auf diese Bounding-Volumes im Datenflussprogramm kann dann durch die abstrakte Schicht möglichst einheitlich gestaltet werden, auch wenn die beiden Modelle unterschiedlich mächtig sind. So ist der ungefähre Mittelpunkt des betrachteten umgebenen Volumens in beiden Fällen meist sehr ähnlich und die ungefähre Größe der Geometrie kann auch in beiden Fällen grob aufeinander angeglichen werden.

Zusätzlich zu den Operationen im Szenengraph wird auch die Arithmetik für Szenengraph-typische Datentypen, wie z.B. 4x4-Matrixen, drei- bzw. vier-komponentige Vektoren und Quaterinonen, durch die Abstraktionsschicht gekapselt.

Da die Anbindung der Programmteile an die Daten des Szenengraphs auf ein Feldkonzept aufbaut, sind auch die Zugriffsfunktionen auf die Felder durch eine Schicht gekapselt. Viele SG-Tools bieten von sich aus schon ein Feldkonzept für den Daten-

zugriff. Ansonsten muss der Field-Event-Layer (siehe Abschnitt 4.13) dieses noch selbst als Interface zu den Szenengraphdaten implementieren.

4.12.1 USG-Komponenten

Die USG-Abstraktionsschicht abstrahiert die Datentypen des Szenengraphs und einen Teil der grundlegenden Szenengraphknoten:

Datentypen für Szenengraphen

Da es keinen Standard für komplexere Datentypen gibt, welche in Szenengraphen zur Beschreibung von Transformationen, Positionen und Richtungen verwendet werden, werden diese durch die USG-Schicht gekapselt.

Transformationen werden in drei Typen unterschieden: Rotationen, Translationen und Skalierungen. Für jeden dieser Typen wird eine eigene Klasse bereitgestellt. Normalerweise werden Rotationen als Quaternionen und Translationen sowie Skalierungen als Vektor mit drei Komponenten (*Vec3*) dargestellt. Die eindeutige Zuordnung zu eigenen Klassen ermöglicht die einfache Konversion der Transformationstypen in 4x4-Matritzen und umgekehrt. So kann z.B. im Konstruktor der Matrixklasse einfach direkt die gewünschte Transformation angegeben werden (Abschnitt 4.12.2 zeigt Beispiele, in denen diese Konversionen eingesetzt werden). Dieses erhöht die Lesbarkeit des Programmcodes gegenüber den üblichen Programmkonstrukten der Szenegraphtools. Auch für die Repräsentation von Positionen und Richtungen, welche in machen SG-Tools nur als einfacher Vektor beschrieben werden, sind eigene Klassen definiert. Tabelle 5 zeigt eine Übersicht der Klassen, wie durch den USG-Layer bereitgestellt werden.

Klasse	Basisklasse	Beschreibung
Trf	4x4-Matrix	Beliebige kombinierte Transformation als 4x4-Matrix repräsentiert
Trl	Vec3	Klasse für die Repräsentation von Translationen um den entsprechenden Vektor.
Scl	Vec3	Klasse für die Repräsentation von (inhomogenen) Skalierungen mit einem Skalierungswert pro Raumachse
Rot	Quaternion	Klasse für die Repräsentation von Rotationen in der eindeutigen Form als Quaternion
Pos	Vec3	Repräsentation einer Position im Raum
Dir	Vec3	Repräsentation einer Richtung in Form eines Richtungsvektors. Die Länge des Vektors wird automatisch auf die Länge 1 normiert.
Color	Vec4	Repräsentation von Farben in Form eines vierkomponentigen Vektors mit jeweils einer Komponente für Rot, Grün, Blau und der Transparenz des Objektes.

Tabelle 5: Die Transformations- und Vektorklassen des USG-Layers

Knotentypen des Szenengraphs

Die Basisknotentypen, wie sie in der Regel in jedem SG-Tool vorhanden sind werden auch vom USG-Layer in eigenen Klassen zur Verfügung gestellt. Diese Basisknoten beschränken sich auf Knoten zur Transformation, Erstellen primitiver Geometrien, Laden von Geometriedatenformaten und einfacher Strukturierungen des Szenengraphs.

Obwohl die Methoden der Knoten sich auf die grundlegenden Operationen im Szenengraph beschränken, kapseln sie zum Teil komplexe Operationen des zugrunde liegenden Szenengraphtools. So kann z.B. die häufig gebrauchte Abfrage der globalen Transformation, wenn sie nicht schon durch das SG-Tool selber implementiert ist, die Abfrage und Kombination mehrerer lokaler Transformationen im Szenengraph bedeuten. Vor allem die einfach erscheinende Operation des Einfärbens von Knoten oder die Darstellung von Text, kann in der konkreten Umsetzung im SG-Tool sehr komplex sein.

Klasse	Beschreibung	Methoden
sgNode	(Abstrakte) Basisklasse aller Szenengraphknoten des USG-Layers	Abfrage des (globalen) Bounding-Volumes Abfrage der globalen Transformation Einfärben des Knotens (und aller eventuellen Kinderknoten)
sgGrp	Klasse für alle Knoten mit Kinderknoten	Verwaltung der Kinderknoten
sgTrf	Knoten mit Transformationsmatrix	Abfrage/Setzen der Transformationsmatrix
sgSwitch	Knoten der die Darstellung seiner Kinderknoten gezielt deaktivieren kann	Setzen der Auswahl der Kinderknoten
sgPrimitiveGeom	Knoten zur Erzeugung einfacher Geometrieformen	Setzen der Geometrieart (Quader, Kugel, Zylinder, etc.) Setzen der Abmessungen
sgText	Knoten zur Erzeugung von gerenderten Text	Setzen des Textstrings Setzen von Größe und Ausrichtung
sgLoadGeom	Knoten zum Laden von externen Geometriedatenformaten	Setzen der Datenfiles
sgLight	Lichtknoten zur Beleuchtung der vorhandenen Geometrie	Setzen der Lichtparameter

Tabelle 6: Die Szenengraphknoten des USG-Layers

Andere Operationen im Szenengraph, welche häufig der Modellierung z.B. von Geometrie, Materialeigenschaften oder Texturierung dienen, werden durch den USG-Layer zunächst nicht implementiert, da der Einsatzzweck des Layers auf die Interaktion mit vorhandener Geometrie ausgelegt ist. Ein Großteil der statischen Geometrieeigenschaften kann durch das Laden externer Geometriedatenfiles abgedeckt werden. Tabelle 6 zeigt eine Übersicht der Knoten des USG-Layers und der von ihnen bereitgestellten Methoden.

4.12.2 Beispiele für Rechenknoten mit USG-Code

Beispiel 1: Die Definition eines Rechenknoten zum Färben der Geometrie unterhalb eines Knotens im Szenengraph.

```
<VPNode name=„vpColorNode“>
 <Fields>
  <Input>
   <Value name=„Node“ type=„USG::sgNode“ mode=„link”
init=„NULL“ />
   <Value name=„Color“ type=„USG::Vec4“
              init=„USG::Color( 0.5, 0.5, 0.5, 1.0 )“ />
   <Trigger „Execute” />
  </Input>
 </Fields>
 <OnFrame>
  if ( Execute_Changed && Node_Value != NULL ) {
     Node_Value -> colorNode ( Color_Value );
  }
 </OnFieldChanged>
</VPNode>
```

Das erste Feld mit dem Namen „Node” enthält einen Zeiger auf den zu färbenden Knoten des Szenegraphen. Das zweite enthält die Angabe der neuen Farbe für diesen Knoten. Die Farbe ist als vier-komponentiger Vektor angegeben, wobei die ersten drei Komponenten die Werte für den roten, grünen bzw. blauen Anteil der Farbe enthalten. Der vierte Wert gibt die Transparenz des Knotens an. Ein zusätzliches Triggerfeld, das die Aktion auslöst, ist notwendig, da sonst das Ergebnis der Färbung abhängig von der Reihenfolge des Setzens der beiden Wert-Felder ist. In diesem Beispiel löst der „Execute“-Trigger bei Veränderung des Feldes im „OnFrame“-Callback die Aktion aus. Das Einfärben des Knotens wird vom USG-Layer für alle Knoten des Szenengraphs implementiert und ist in der Regel ein reichlich komplexer Vorgang in dem konkreten Interface des Szenengraphentools.

Beispiel 2: Ein Programmknoten zur Visualisierung eines Zahlenwertes durch die Größe einer Kugel.

```
<VPNode name=„vpDblVisSphere“>
 <Fields>
  <Input>
   <Channel name=„In“ type=„double“ init=„0“ />
  </Input>
  <Output>
   <Value name=„VisNode“ type=„USG::sgTrfNode“ mode=„link“
          init=„USG::sgTrfNode ()“ />
  </Output>
 </Fields>
 <AddVariable name="ScaleNode" type="USG::sgTrfNode"
          mode=„link” init="USG::sgTrfNode ()"/>
 <AddVariable name="SphereGeom"
          type="USG::sgSimpleGeomNode"
          init="USG::sgSimpleGeomNode (USG::SPHERE,1.0)"
          mode=„link” />
 <OnInit>
  VisNode_Value -> addChild(ScaleNode);
  ScaleNode -> addChild(SphereGeom);
  SphereGeom -> colorNode ( Vec4 ( 1.0, 0.5, 0.5, 1.0 ) );
 </OnInit>
 <OnFrame at="end">
  ScaleNode -> setMatrix ( Scl ( In_Value, In_Value, In_Value ) );
 </OnFrame>
</VPNode>
```

Das einzige Eingabefeld „In“ enthält den zu visualisierenden Zahlenwert. Obwohl das Feld eine Datensequenz enthält, wird bei den Visualisierungsknoten immer nur der aktuellste Wert benutzt, da die berechnete Geometrie nur einmal pro Renderframe gezeichnet werden kann.

Das Ausgabefeld enthält den Visualisierungsknoten, der später bei der Instanziierung an eine geeignete Stelle im Szenengraph gehängt werden kann. Die zwei lokalen Klassenvariablen werden direkt mit einem Transformationsknoten für die Skalierung bzw. einer Kugelgeometrie mit dem Durchmesser von einer Einheit initialisiert. Bei der Initialisierung des Knotens werden der Skalierungsknoten und der Geometrieknoten unter den Visualisierungsknoten gehängt und die Kugelgeometrie in einem hellen Rot eingefärbt.

Um zu erreichen, dass die Visualisierung nur für den aktuellsten Wert berechnet wird, geschieht dieses im „OnFrame“-Callback nach der Berechnung der Datentoken. In diesem Fall enthält die Variable „In_Value“ den aktuellsten Wert aus der Datensequenz. Aus diesem Wert wird eine Skalierungsmatrix erstellt und als aktuelle Transformation des Transformationsknotens gesetzt. Der Durchmesser der Kugel entspricht somit immer dem Zahlenwert an dem Eingabefeld und kann eine schnell erfassbare Darstellung von Wertveränderungen und dem ungefähren, aktuellen Zahlenwert zur Verfügung stellen.

4.12.3 Konkrete USG-Layer Implementationen

Im Bereich der Implementation des visuellen Programmiersystems wurden für zwei VR-Szenengraph-Tools USG-Layer implementiert. Hierbei musste von zwei unterschiedlichen Systemen abstrahiert werden, die sich grundlegend im Aufbau der Szenengraphknoten unterscheiden: Zum einem OpenGL-Performer von SGI und zum anderen die Open-Source Entwicklung OpenSG (siehe Abschnitt 2.5.1).

OpenGL-Performer setzt eine eher übliche Szenengraphstruktur in Form eines gerichteten Graphen mit mehreren Elternknoten ein. Dadurch sind Mehrfach-Instanziierungen von Objekten über mehrere Pfade, die zu dem Objekt führen, möglich. OpenSG erlaubt hingegen nur Baumstrukturen als Szenengraph und ermöglicht die Wiederverwendung z.B. von Geometrien durch die von mehreren Knoten gemeinsam benutzbare Programmkerne (*Node Cores*). Auch in der Art der Instanziierung der SG-Knoten und sogar in der Interpretation der Multiplikationsreihenfolge der homogenen 4x4-Matrizen für die Transformation unterscheiden sich die VR-Tools. Die Umstellung größerer VR-Applikationen von einem der beiden Szenengraphtools zum anderen ist ohne ein abstraktes Programmier-Interface, wie der USG-Layer es bildet, meistens nur mit einer kompletten Re-Implementierung oder überhaupt nicht realisierbar.

In beiden Tools ist die Erstellung von primitiven Geometrien – z.B. Quadern, Zylindern, Kugeln und Kegel – möglich. Die jeweiligen Ausrichtungen, Größen und initialen Translationen der einzelnen Geometrien variieren zwischen den beiden SG-Tools und sind oft mühsam abzugleichen. Die abstrakte Beschreibung des USG-Layers für bestimmte Geometrien bringt die unterschiedlichen Darstellungen durch entsprechende integrierte Transformationen der Geometrien auf eine einheitliche Form.

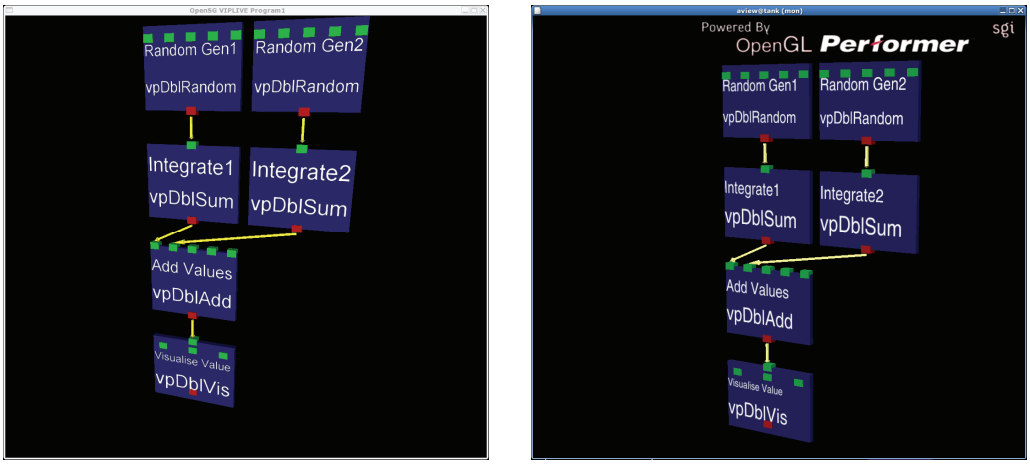


Abbildung 52: VIPLIVE-Programm in OpenSG und AVANGO

Abbildung 52 zeigt zweimal das gleiche visuelle Programm in VIPLIVE. Auf der linken Seite wurde als VR-Tool OpenSG über den USG-Layer angebunden; die rechte Seite zeigt das Programm mit der USG-Anbindung von AVANGO/Performer. Der einzige visuelle Unterschied der beiden USG-Layer Implementierungen zeigt sich in der leicht unterschiedlichen 3D-Schrift da für die beiden Tools nicht der gleiche Schriftsatz zur Verfügung steht. Auch die genaue Anpassung der Größe der

Schrift auf die Breite der Programmknoten ist momentan in OpenSG exakter realisiert.

4.13 Der Feld-Event-Layer

Da die meisten Szenengraphtools kein eigenes, für die Datenflussprogrammierung brauchbares Feld und Eventsystem bereitstellen, wurde ein generelles Programmierinterface für die Benutzung von Feldern, Feldverbindungen und durch das Eventsystem der Feldwertänderungen aufgerufenen Callbacks entwickelt.

Im Falle des VR-Tools Avango bildete dieses zunächst die Implementation des von Avango bereitgestellten Systems der Felder und Callbacks ab. Damit wurde innerhalb des Avango Toolkits auch eine Version der visuellen Programmierumgebung realisiert.

Bei der Re-Implementation der Funktionalität für das OpenSG-Tool, das von sich aus kein geeignetes Eventsystem mitbringt, zeigte sich aber, dass ein abstrakterer Ansatz, als das von Avango bereitgestellte Interface, einfacher für die Realisierung eines allgemeinen Datenflussablaufschemas zu handhaben ist.

Die Hautunterschiede des Feld-Event-Layers im Vergleich zu dem in Abschnitt 2.5.1 beschriebenen System von Avango sind:

- Die Unabhängigkeit von dem Typsystem der verwendeten Datentypen, das für Avango von OpenGL-Performer übernommenen wurde.

  Da der Layer für unterschiedliche VR-Tools eingesetzt werden soll ist die Abhängigkeit von einem speziellen Toolkit nicht wünschenswert und kann durch den Field-Event-Layer zusammen mit dem USG-Layer vermieden werden.

- Ein zusätzlicher Callback, der bei der Abfrage von Feldwerten aufgerufen wird.

  Während bei Avango alle eventuell abgefragten Feldwerte in jedem Frame berechnet werden müssen, damit der aktuelle Wert immer zur Verfügung steht, kann durch diesen Callback die Berechnung auf die Fälle eingeschränkt werden, in denen der Wert auch wirklich benutzt wird.

- Eine klarere Strukturierung des Scheduling-Verfahrens für die Anmeldung und Abarbeitung der Programmknoten für den Berechnungs-Callback.

  Bei Avango melden sich alle Knoten, die in jedem Frame evaluiert werden sollen (wie z.B. alle Sensorknoten) selber zur Evaluierung an und stehen dann eventuell mehrfach in der Evaluationsqueue. In dem neuen Sheduling-Konzept können Knoten direkt einmal zur Evaluation in jedem Frame angemeldet werden. Sie werden dann in einer eigenen Liste gespeichert und am Anfang jedes Frames in die aktuelle Evaluationsqueue eingetragen. Auch verlegt der Prozess von Avango in der aktuellen Implementierung die Evaluierung von Programmknoten mit geringerer Priorität als der aktuell evaluierte Knoten auf den nächsten Berechnungsframe, auch wenn sie noch im aktuellen Frame abgearbeitet werden sollten. Eine weitere besondere Anforderung stellt die mehrfache Evaluierung von Programmknoten, die z.B. im Falle von Knoten für die Constraintüberwachung (siehe Kapitel 1) nicht generell unterbunden werden darf.

Weitgehend identisch geblieben sind das Prinzip der Felder und Feldcontainer, die Aufrufe der Callbacks bei der Änderung von Feldwerten und die Propagierung der Werte durch die Feldverbindungen. Tabelle 7 zeigt die top-level Klassen des Feld-Event-Layers ohne die darunter liegenden Basisklassen.

Klasse	Beschreibung	Methoden / Callbacks(CB)
Field	Template-Klasse für Felder mit beliebigen Werttypen	Abfragen/Setzen des Feldwertes Etablieren/Lösen von Feldverbindungen CB bei Änderung des Feldwertes CB bei Abfrage des Feldwertes
FieldContainer	Klasse für Rechenknoten mit Feldern	Methoden zur Verwaltung der eigenen Felder CB bei Änderung eines Feldes des Containers CB bei Abfrage eines Feldes des Containers CB für die Evaluation
Evaluater	Klasse für die Instanz eines statischen Objektes, das die Evaluierung der Fieldcontainer organisiert.	Methode zur Anmeldung des Containers zur Evaluation (einmal oder für jeden Frame) CB zur Abarbeitung der Evaluationsqueue einmal pro Berechnungsframe

Tabelle 7: Oberste Klassenstruktur des Feld-Event-Layers

Aufgrund der besseren Handhabbarkeit für die Datenflussprogrammierung des neu implementierten Layers für Felder und Events gegenüber dem von Avango, aber auch wegen der einheitlicheren Implementation der Rechenknoten bei den unterschiedlichen SG-Tools, wurden die aktuelle Version der Rechenknoten mit der neuen Implementation realisiert. Für die Anbindung der Felder des benutzten VR-Tools kommen eigene Konverterknoten zum Einsatz, die sowohl Felder von Avango bzw. OpenSG als auch des Feld-Event-Layers besitzen können.

5 Constraint-Mediatoren

Mit Hilfe der visuellen Datenflussprogrammierung ist es möglich, komfortabel Werte für die Interaktion mit Teilen in der Virtuellen Umgebung zu errechnen und an Parameter im Szenengraph zu propagieren. Um bestimmte Abläufe in der Virtuellen Umgebung festzulegen und z.B. relative Transformationen von Objekten einschränkten zu können ist es notwendig, Feldwerte nicht nur zu setzen, sondern deren Werte auch zu überwachen und einzuschränken. Die Einschränkungen im Eventfluss der Datenflussprogrammierung, welche wichtig für ein konsistentes Ablaufmodell der DFVPL sind, verhindern aber die Realisierung dieser Überwachungsknoten. Das Problem entsteht, da sowohl in dem Ablaufmodell der klassischen Datenflussprogrammierung, als auch der Realisierung in den Feldverbindungen der meisten mit Datenflusskonzepten angereicherten VR-Tools, die Möglichkeit eines gleichzeitig lesenden als auch schreibenden Zugriffs eines Knotens auf ein Feld ausgeschlossen wird. Das Hauptproblem ist hierbei die grundlegende Forderung nach azyklischen Strukturen im Datenflussgraphen und damit auch in dem Aufbau der Feldverbindungen.

Abbildung 53 zeigt zwei Realisierungen von überwachenden Knoten, welche aber zwangsläufig zyklische Strukturen bauen, um die Werte des überwachten Feldes auslesen und dann eventuell eingeschränkte Werte wieder setzen zu können. Sowohl die Implementation als bidirektionales Feld als auch der Umweg über getrennte Eingabe- und Ausgabefelder sind in der Regel in den datenflussorientierten VR-Tools nicht erlaubt. Zyklische Verbindungen führen bei der Propagierung durch das Eventsystem zu Endlosschleifen oder die Weiterleitung der Events wird bei zyklischen Verbindungen von der Applikation erkannt und unterbunden.

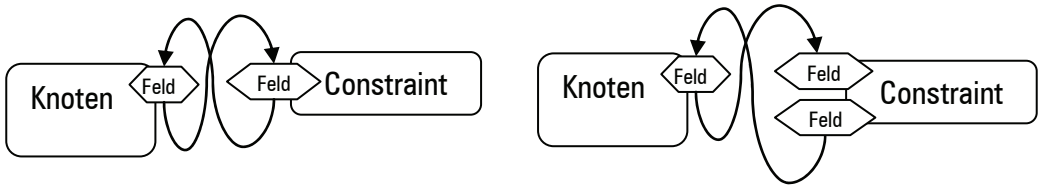


Abbildung 53: Verbotene zyklische Verbindungen zur Realisierung von überwachten Feldern

Aber auch der oft geforderte Verzicht von Joint-Links bzw. eindeutigen eingehenden Feldverbindungen verhindert eine Modellierung von überwachten Feldern, wenn diese durch eine weitere Feldverbindung gesetzt werden sollen. Da die propagierende Verbindung das Feld kontrolliert und weitere Feldverbindungen bei dem Verzicht von Joint-Links ausgeschlossen sind, kann die zusätzliche Verbindung vom Watcher nicht etabliert werden.

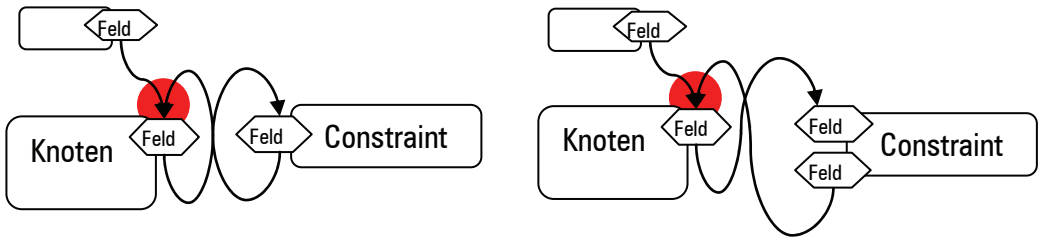


Abbildung 54: Zusätzliche Probleme durch das Verbot von Joint-Links

Abbildung 54 zeigt die problematischen Joint-links (in der Abbildung dunkel unterlegt), die bei der Überwachung von Feldern, welche schon durch Feldverbindungen gekoppelt sind, entstehen.

Die Möglichkeit Feldwerte zu überwachen kann in datenflussorientierten Szenengraph-Tools aber die Möglichkeit erschaffen, lokale Constraints direkt auf Felder anzuwenden um damit eine komfortable Möglichkeit der Anwendungssteuerung zu erreichen.

Die Implementierung von lokalen Constraints erfolgt in dem visuellen Programmiersystem durch *Constraint-Mediatoren (CM)*. Diese sind Rechenknoten, welche ähnlich aufgebaut sind wie die Rechenknoten in der Visuellen Programmierung VIPLIVE. Constraint-Mediatoren erfordern nur geringe Erweiterungen in der Definition der ursprünglichen Programmknoten und können mit diesen kombiniert werden.

Die Constraint-Mediatoren verfolgen eine lokale Lösungsstrategie des Constraint-Problems, deren Abarbeitung aber lokal durch ein System von Propagierungsevents oder durch eine globale Instanz gesteuert werden kann. Bei Konflikten können einzelne CM auf die Erfüllung des durch sie modellierten Constraints je nach Priorität auch ganz oder teilweise verzichten. Daher modellieren sie keine strikten Constraints sondern können die Constraintlösung anhand von Prioritäten der einzelnen lokalen Constraints untereinander vermitteln. In der Regel passen Constraint-Mediatoren bei einer Veränderung der Werte von außen ihre Parameter innerhalb der durch sie definierten Freiheitsgrade möglichst weit an und propagieren eventuell nicht erfüllbare Teilkonfigurationen an übergeordnete Instanzen. Diese lokale Strategie mit einer globalen Kontrollinstanz erlaubt in Teilbereichen eine Lösung bei zyklischen Strukturen in dem implizit durch die CM aufgebauten Constraint-Graphen (siehe Abschnitt 3.1.3).

Wird das vom Constraint überwachte Feld von außen so verändert, dass der Feldwert nicht mehr den Bedingungen genügt, wird er nicht zurück auf den Ausgangswert, sondern auf einen möglicht nahen Wert noch innerhalb der Grenzen gesetzt. Der Abstand zwischen dem Sollwert und dem gesetzten Wert wird über eine zugeordnete Metrik bestimmt. Die Differenz zwischen dem Sollwert und dem eingenommenen Wert kann weiter an ein anderes Feld propagiert werden, welches seinerseits eventuell auch wieder durch ein Constraint eingeschränkt wird. Einfache, regelbasierte Geometric-Contraint-Solver verfügen zwar über die Möglichkeit Constraints nach Prioritäten zu ordnen (siehe Abschnitt 3.1.3), können aber nicht mehrere Constraints bei der Veränderung von nur einem Wert abgleichen, wobei der Wert durch die einzelnen Constraints schon teilweise angenähert werden kann.

So können z.B. durch das einfache Setzen einer Matrix eines Transformationsknotens im Szenengraphen mehrere Constraints ihre Parameter verändern, um die gewünschte Transformation möglichst exakt zu erreichen. Ein Anwendungsbeispiel ist in Abschnitt 6.3.5 zu finden.

Ein direkter Vorteil bei der Definition von Constraint-Mediatoren ist, dass die CM in dem gleichen XML-Format wie die Rechenknoten erstellt werden können. Der Hauptunterschied ist die Eingabe- bzw. Ausgabesemantik der Felder und das Ablaufschema der CM. Während die Rechenknoten neben den Parameterfeldern eindeutige, unidirektionale Eingabe- bzw. Ausgabefelder haben und im Sinne der Ablaufsemantik der Datenflussprogrammierung keine Feldwerte außer den eigenen Ausgabefeldern

ändern dürfen, müssen die CM die komplette Kontrolle über externe Felder übernehmen können. Außerdem folgen CM nicht dem strikten Ablaufschema der Datenflussprogramme.

5.1 Erweiterung der Programmknotendefinition

Die Feldtypen der Rechenknoten werden um einen weiteren Typ erweitert, um die Referenzierung von knotenfremden Feldern mit Lese- und Schreibzugriff zu ermöglichen. Außerdem muss der Ablauf der Constraint-Mediatoren, der nicht dem strikten Ablaufschema der Datenflussprogrammierung folgt, von außen gesteuert werden können. Die Berechnung eines CMs wird dabei entweder durch die Propagierung einer lokalen Wertänderung oder durch eine globale Kontrollinstanz angestoßen. Dafür muss es die Möglichkeit geben die Rechenknoten der CM unabhängig vom Ablaufschema evaluieren zu lassen.

5.1.1 Feld-Referenzen

Das in 4.11.1 beschriebene XML-Format wird für die Constraint-Mediatoren um die folgenden XML-Tags erweitert.

Zunächst ergibt sich ein neues Subtag („Reference“, in der Tabelle durch Unterstreichung hervorgehoben) für das „Fields“-Tag. Da Feldreferenzen durch ihren direkten Zugriff nicht den Eingabe- oder Ausgabefeldern zugeordnet werden können, ergibt sich hier die Unterscheidung.

Tag	Sub-Tags	Attribute	Body
Fields	Input Output Bidirect Reference	-	-

Dieses „Reference“-Tag hat die üblichen Sub-Tags zur Unterscheidung zwischen Feldern mit Datensequenzen und denen mit einfachen Feldwerten. Da Triggerfelder keine Werte haben, die man referenzieren könnte, tauchen diese hier nicht auf.

Tag	Sub-Tags	Attribute	Body
Reference	Stream Value	-	-

Die beiden Sub-Tags des „Reference“-Tags sind wie bei den üblichen Eingabe- und Ausgabefeldern aufgebaut. In diesem Fall bezeichnet das „type“-Attribut den Datentyp des referenzierten Feldes und das „init“-Attribut kann direkt auf das Referenzfeld verweisen, wird aber in der Regel nicht benutzt, da die Feldreferenzen normalerweise erst nach der Instanziierung der Knoten gesetzt werden.

Diese Felder ermöglichen die Referenzierung externer Felder (Channel und einfache Feldwerte) für den freien Zugriff auf die Feldwerte. Hierdurch kann die strikte,

gerichtete Feldwertpropagierung der Datenflussprogrammierung aufgebrochen werden. Das ist notwendig, da die eingesetzten Constraints in der Regel ungerichtete, bidirektionale Abhängigkeiten beschreiben, und daher die Werte je nach Eventfluss in mehrere Richtungen propagieren können (siehe auch (Biermann & Wachsmuth, 2004)).

Für den Zugriff auf die Werte des referenzierten Feldes in dem C++-Code sind wie bei den anderen Feldern folgende Variablen und Funktionen vordefiniert:

[Feldname]_Value	enthält den Wert des referenzierten Feldes zur aktuellen Berechnungszeit
[Feldname]_Changed	boolscher Wert, ob das referenzierte Feld neue Daten enthält
Set_[Feldname] ([Wert])	setzt das im Argument angegebene Datentoken in der Datensequenz des referenzierten Feldes, bzw. setzt den Feldwert eines einfachen, referenzierten Feldes

Im Falle eines Datensequenz-Kanals schreibt die „set_...“-Funktion kein zusätzliches Datentoken auf einen Ausgabekanal, sondern ändert direkt das Datentoken im Kanal des referenzierten Feldes[11].

5.1.2 Evaluations-Trigger

Damit die Berechnung CM unabhängig von dem Evaluierungsprozess der Rechenknoten, wie er im Ablaufschema vorgegeben ist, angestoßen werden kann, werden zusätzliche Triggerfelder eingeführt, welche bei Aktivierung über die Veränderung eines verbundenen Feldes direkt die Abarbeitung der internen Rechenprozesse auslösen. Diese *Evaluations-Trigger* führen bei jeder Aktivierung direkt eine Evaluierung des zugehörigen Knotens durch. Da Feldreferenzen nicht über Feldverbindungen gesetzt werden, wird durch die Veränderung eines überwachten Feldwertes der CM-Knoten nicht zur Evaluierung angemeldet. Damit CM-Knoten nicht ständig ihre Feldwerte überprüfen müssen, auch wenn diese sich nicht verändert haben, wird die Evaluierung durch diese Trigger angestoßen. Realisiert werden Evaluations-Trigger dadurch, dass in dem OnFieldChanged-Callback für diesen Trigger einmal die Methode zur Evaluierung des Knotens aufgerufen wird. Im XML-Tag zur Definition der Triggerfelder wird daher ein zusätzliches Attribut zur direkten Evaluation eingeführt.

Tag	Sub-Tags	Attribute	Body
Trigger	-	**name** [String] direct-eval = [“true”|**“false”**]	-

[11] Da der Abgleich der Taktung von zwei referenzierten Feldern mit Datensequenzen – aufgrund der Interferenz mit den Berechnungen im normalen Datenfluss – problematisch ist, sollte ein Constraint-Mediator immer nur maximal ein Feld mit Datensequenzen überwachen, falls diese unterschiedliche Taktungen haben.

Eine direkte Evaluierung unter Umgehung der Evaluationsqueue ist notwendig, um z.B. geschachtelte Aufrufe von CM-Knoten zu ermöglichen. Zum Einsatz kommt dieses bei der Constraint Propagierung, welche im folgenden Abschnitt beschrieben wird. Hierbei können externe CM gerade veränderte Feldwerte überprüfen und gegebenenfalls einschränken, bevor die interne Abarbeitung fortgeführt wird.

5.2 Constraint Propagierung

Die Art der Propagierung von Werten durch Constraint-Mediatoren muss an zwei Stellen festgelegt werden. Zum einem muss bei bidirektionalen CM das Schema für die Propagierung der internen Werte festegelegt sein, zum anderen muss eine globale Kontrolle die Abarbeitung der CM überwachen falls die lokale Strategie der Evaluierung nicht ausreicht und eine zusätzlich komplexes Verhalten der Anwendung bei dem Umgang mit Konflikten der lokalen Constraints erforderlich ist.

5.2.1 Interne Propagierung und Konfliktbehandlung

Da bidirektionale Constraint-Mediatoren Werte in zwei Richtungen propagieren können, wird ein internes Propagierungsschema in Abhängigkeit der Änderung der überwachten Werte festgelegt.

Dabei wird zunächst grob unterschieden, ob sich einer der beiden Werte oder beide sich geändert haben. Für diese Festlegung wird darauf zurückgegriffen, dass die Constraints in ein VR-System mit definierten Verarbeitungsschritten (Frames) eingebettet sind. Daher wird bei Betrachtung, ob die Werte eines Constraints sich verändert haben, nur der aktuelle Frame ausgewertet. Sobald ein CM evaluiert wird, wird überprüft, welcher der beiden Werte verändert wurde.

Ist nur einer der überwachten Werte verändert, wird dieser als Master festgelegt. Der Master hat in der folgenden Berechnung die Priorität über den anderen überwachten Wert. Zunächst kann der Masterwert aber selber durch interne Constraints des Mediators eingeschränkt werden. Danach erfolgt die Berechnung des untergeordneten Wertes, welcher als Feldwert gesetzt wird. An dieser Stelle kann durch die Veränderung des Feldwertes über einen externen CM erfolgen, der durch einen Evaluations-Trigger angestoßen wird (siehe Abschnitt 5.2.2). Mittels dieses – eventuell eingeschränkten Wertes – wird ein neuer Wert des ursprünglichen Masterwertes berechnet und gesetzt. Durch diese Strategie kann der CM innerhalb einer Evaluation den untergeordneten Wert auf den Wert des Masters setzen und Einschränkungen, welche diesen Wert betreffen, direkt wieder zurück an den Master propagieren.

Hat sich im letzen Frame keiner der überwachten Werte verändert, wird der CM aufgrund des Ablaufschemas der Rechenknoten nur evaluiert, falls sich ein anderer Parameterwert verändert hat. Da die Veränderung des Parameters ein anderes Ergebnis bei der Einschränkung der Werte oder bei der Berechnung der Kopplungsfunktion zur Folge haben kann, wird auch hier der gleiche Ablauf wie im vorherigen Fall abgearbeitet. Der Masterwert bleibt unverändert und wird daher durch die vorherige Veränderung der überwachten Werte bestimmt.

Falls die beiden überwachten Werte sich innerhalb eines Frames gleichzeitig geändert haben, liegt ein potentieller Konflikt vor. Dies kann eintreten, wenn an mehreren Stellen in der überwachten Struktur gleichzeitig durch externe Ereignisse Werte

verändert wurden, oder die Struktur des implizit über die lokalen Constraints aufgebauten Constraintgraphen Schleifen beinhaltet. Der erste Fall tritt in der Praxis nur selten auf, da Interaktionen in der Virtuellen Umgebung in der Regel nur ein Bauteil oder Aggregat betreffen. Ansonsten kann eine interne Regel des CM bestimmen, welche Propagierungsrichtung in einem solchen Fall Vorrang hat, oder ob zwischen den Werten vermittelt werden soll und z.B. das arithmetische Mittel der beiden überwachten Werte als Ergebnis bereitgestellt wird.

Für den Fall von Schleifen im Constraintgraphen wird zunächst überprüft, ob die Veränderungen der beiden Werte mit dem Constraint, der durch den aktuellen CM implementiert wird, kompatibel sind. Falls dem so ist, werden die beiden Werte unverändert gelassen und damit die weitere Propagierung der Werte an dieser Stelle abgebrochen, da ein kohärenter Zustand erreicht ist. Der Fall, dass die beiden überwachten Werte sich gleichzeitig, aber im Rahmen der Einschränkungen des überwachenden Constraints verändern, kann z.B. bei kompatiblen Mehrfachverbindungen von Bauteilen auftreten. Hierbei versuchen beide Verbindungsconstraints dieselbe Transformation an das verbundene Bauteil zu propagieren.

Eine Priorität der einzelnen Constraints kann auch bestimmen, welche CM im Falle von Konflikten ihre Constraints durchsetzen können. So sollte z.B. die Bewegung eines Lenkrades nur soweit der Bewegung der greifenden Hand folgen, wie es seine Constraints durch die verbundenen Gelenke erlauben, vorausgesetzt dass es auch fest mit diesen Teilen verbunden ist. Hier sollten also auf jeden Fall die Bewegungsconstraints, welche die Bauteile definieren, Vorrang vor den Interaktions-Constraints der freien Interaktion durch den Benutzer haben.

In machen Fällen reicht eine interne Vermittlung der Werte oder die Vergabe von Prioritäten nicht aus. Werden bei der Interaktion Constraints z.B. absichtlich verletzt, muss eine globale Kontrolle ein adäquates Verhalten der Anwendung anstoßen, wie es im folgenden Abschnitt beschrieben wird.

Da die Constraint-Mediatoren beim Lösen des implizit aufgebauten Constraint-Graphen eine lokale Strategie – welche die Behandlung von Schleifen im Graphen ermöglichen – bei der Lösung der Constraints verfolgen, muss die Reihenfolge der Abarbeitung der CM-Knoten kontrolliert werden.

5.2.2 Ablaufschema der Constraint-Mediatoren

Für die Abarbeitung der einzelnen CM-Knoten arbeiten die Mediatoren mit einer zweifachen Strategie: Zum einem wird ein Eventsystem der Feldwertpropagierung eingesetzt um zu ermitteln, welche Mediatoren für diesen Frame ihren Zustand überprüfen müssen. Dieses erlaubt eine höchst effiziente Auswertung der CM, da nur die Mediatoren berechnet werden müssen, welche direkt an die im aktuellen Render-Frame veränderten Felder gekoppelt sind. Da sich in der Regel durch Benutzer-Interaktion oder andere Ereignisse nur ein Bruchteil der Felder – z.B. die Transformationsmatrizen – im Szenengraph verändert, brauchen auch nur wenige Mediatoren ihren Zustand zu überprüfen. Die Evaluierung über diese Art von Events erfolgt nach den Regeln der Abarbeitung von den Programmknoten über die Evaluationsqueues in der visuellen Programmierung. Das heißt ein CM wird zur Evaluierung angemeldet sobald ein (überwachter) Feldwert sich ändert. Die dann folgende Abarbeitung des

CM kann dann durch weitere Feldwertänderungen die Anmeldung von weiteren Knoten zur Evaluierung zur Folge haben.

Für den Fall, dass Mediatoren die Werte des veränderten Feldes verändern oder Werte an andere Felder weiter propagieren, können die Mediatoren über die Evaluations-Trigger Events an bestimmte Felder senden, mit denen sich Mediatoren rekursiv evaluieren lassen. Dafür können von den Feldern eines Mediators Verbindungen zu Evaluations-Triggerfeldern aufgebaut werden, die eine direkte Evaluation ohne den Weg über die Evaluationsqueues auslösen.

Behandlung von Schleifen

Um mehrfache Ausführung während eines Frames zu vermeiden, ist jedem CM-Knoten die aktuelle Frame-Nummer bekannt und er kann anhand dieser feststellen, ob und wie oft er evaluiert wurde. Je nach Typ und Einsatz des CMs sind eventuell mehrere Updates während eines Frames nötig um einen kohärenten Zustand zu erreichen. Wird durch zyklische Strukturen im implizit aufgebauten Constraintgraphen die Anzahl der erlaubten Evaluierungen pro Frame überschritten, können die beteiligten CM wie im Falle von widersprüchlichen Werten entweder interne Lösungen über die Vermittlung der Werte anbieten, oder eine externe Problembehandlung anstoßen.

Schleifen, die durch rekursive Aufrufe über Evaluations-Trigger verursacht werden, werden direkt unterbunden, da sie sich unkontrolliert außerhalb der normalen Abarbeitung der Programmknoten ereignen. Das wird dadurch erreicht, dass während der Abarbeitung der ‚Evaluate'-Methode eines Knotens alle Evaluations-Trigger für diesen Knoten deaktiviert werden.

5.2.3 Externe Konfliktbehandlung

Die Kopplung der Mediatoren an die semantischen Informationen der Bauteile über Semantic-Entities (siehe Abschnitt 6.1) ermöglichen es, zusätzliche Statusinformationen über die Abarbeitung der Constraints an weitere Komponenten weiterzugeben. So kann z.B. die Information, dass, bzw. wann, ein Bauteil oder Aggregat unter den gegebenen Constraints einen kohärenten Zustand erreicht hat – bzw. aufgrund von Konflikten oder Schleifen nicht erreichen kann – im semantischen Netz gespeichert werden. Diese Information kann von weiteren externen Mediatoren benutzt werden um z.B. die Propagierung der Bewegungen möglicht effizient abzuarbeiten. Dies ermöglicht auch ein komplexes Verhalten der Applikation anzusteuern, welches einen vorliegenden Konflikt durch die Konfiguration oder die gezielte Deaktivierung von Constraints auflösen kann.

Werden – z.B. während der Interaktion – zwei verbundene Bauteile in einer entgegengesetzten Richtung auseinander gezogen, kann eine geeignete Interaktion die beteiligten Verbindungsconstraints eventuell an geeigneter Stelle abbauen. Im vorliegenden Beispiel könnte der Constraint, welcher die Verbindungsrelation an den Verbindungsstellen der beiden Bauteile überwacht, komplett deaktiviert werden und damit die Verbindung der Bauteile lösen. Da dieses Verhalten der Anwendung eventuell von weiteren Vorgaben abhängig ist und außerdem eine Veränderung in der semantischen Repräsentation – hier der Verbindungsrelationen der Bauteile – nach

sich zieht, muss diese Aktion durch die globale Anwendungslogik der Applikation gesteuert werden.

5.2.4 Propagierung Beispiel: Bauteil-Verbindungen

Ein Beispiel für mehrfach zu evaluierende Mediatoren sind die CMs, welche die Verbindungen zwischen zwei Bauteilen überwachen (siehe Abschnitt 5.3.3). Diese propagieren beim ersten Durchlauf erfolgte Bewegungen eines Bauteiles an das verbundene Teil. Im zweiten Durchlauf werden die Bewegungen, die nicht den Bewegungsconstraints des verbundenen Bauteiles genügen, wieder zurück an das erste Bauteil propagiert. Damit können eingeschränkte Bewegungen auch für Teile wirksam werden, deren Bewegung nur durch verbundene Bauteile indirekt eingeschränkt ist (siehe das Beispiel in Abschnitt 6.3.5).

5.3 Constraint-Mediator-Typen

Für den Einsatz in der Virtuellen Konstruktion werden unterschiedliche Arten von Constraints benötigt, für die verschiedene CM bereitgestellt werden. In erste Linie betrifft das geometrische Constraints von Bauteilen und ihrer Verbindungen. Die mehrstufige Verschaltung von CM ermöglicht mit einer überschaubaren Anzahl verschiedener Constraint-Mediator-Typen ein breites Spektrum von realisierbaren Constraints.

5.3.1 Matrix-Constraint-Mediatoren

Für die unter Abschnitt 3.2 beschriebenen Transformations-Constraints wird jeweils ein Matrix-Constraint-Mediator eingesetzt. Sie halten jeweils ein Referenzfeld für die zu überwachende Matrix des entsprechenden Transfomationsknotens im Szenengraph und – je nach Anzahl der konstanten Parameter und Freiheitsgrade – entsprechend viele Wertfelder. Für die verschiedenen Transformationsgruppen: Translation, Skalierung und Rotation ist jeweils ein Matrix-CM definiert, der nur die jeweilige Transformationsart für die überwachte Matrix zulässt. Die hierarchische Anordnung mehrerer von CM überwachten Transformationen erlaubt dabei beliebige Constraint-Arten durch die Kombinationen der Constraints. Für häufig benutzte Kombinationen von verschiedenen Constraint Arten, z.B. Rotation um eine Achse plus Translation in der Ebene, können auch spezielle Matrix-CM vordefiniert werden.

Der Zugriff auf die Matrixwerte erfolgt direkt über die Feldreferenz. Falls die Matrix im überwachten Feld oder eines der Parameterfelder sich ändert, wird der OnComputeValue-Callback des Mediators aufgerufen. Innerhalb dieses Callbacks wird dann geprüft, ob der Wert innerhalb der definierten Vorgaben liegt und kann ansonsten verändert werden, um den vorgegebenen Constraint teilweise oder komplett zu erfüllen.

Bei einer Änderung der Matrix wird diese in ihre jeweiligen Komponenten der Translation, Rotation bzw. Skalierung aufgeteilt. Durch diese Komponenten wird die überwachte Matrix auf die den Parametern entsprechende Transformation eingeschränkt. Dafür werden die Parameterfelder der Freiheitsgrade gesetzt, welche direkt durch weitere Constraint-Mediatoren an diesen Feldern (siehe folgender Abschnitt) eingeschränkt werden können, um schließlich daraus die neue Transformationsmatrix

zu erstellen. Generell können CM ihre definierten Constraints auch nur zu einem Teil erfüllen, indem sie die vorgegebenen Werte möglicht gut annähern und die Differenz zur Vorgabe an weitere übergeordnete Transformationsknoten und deren Mediatoren weitergeben.

Die Weitergabe erfolgt über die Referenz auf die Matrix des entsprechenden Transformationsknotens. Der propagierte Wert an diese Matrix ergibt sich aus der Differenz zwischen der aktuellen Transformation $M_{N_{CM}}$ und der Transformation nach Anwendung des Constraints: $M^{cnstr}_{N_{CM}}$. Hierbei ist zu beachten, dass diese Transformation für die Weitergabe von dem eigenen in das Koordinatensystem des Transformationsknotens umzurechnen ist. Nur so kann gewährleistet werden, dass die globale Transformation des gesamten Szenengraph-Astes sich nicht verändert.

Sei:

N_{CM} der durch den CM überwachte Knoten

$M_{N_{CM}}$ seine lokale Matrix vor der Anwendung des Constraints

$M^{cnstr}_{N_{CM}}$ die Matrix, die den Constraint Vorgaben entspricht

N_{adj} der Knoten für die Propagierung der Differenz, der sich in der SG-Hierarchie oberhalb von N_{CM} befindet

$M_{N_{adj}}$ seine lokale Matrix;

$M_{N_{adj}}'$ seine Matrix nach der Propagierung

$T_{\rightarrow N}$ die globale Transformation des Knotens N

$p(N)$ der Szenengraph-Elternknoten von N

$T_{N_1 \rightarrow N_2}$ die Transformation von dem Koordinatensystem des Knotens N_1 in das von N_2 (inklusive der jeweils eigenen Transformation der Knoten)

Dann gilt:

$$T_{N_{adj} \rightarrow p(N_{CM})} = T_{\rightarrow N_{adj}}^{-1} \cdot T_{\rightarrow p(N_{CM})}$$

$$T_{\rightarrow N_{CM}} = T_{\rightarrow p(N_{CM})} \cdot M_{N_{CM}}$$

$$T_{\rightarrow N_{CM}} = T_{\rightarrow N_{adj}} \cdot T_{N_{adj} \rightarrow p(N_{CM})} \cdot M_{N_{CM}}$$

$$T_{\rightarrow N_{CM}} = T_{\rightarrow p(N_{adj})} \cdot M_{N_{adj}} \cdot T_{N_{adj} \rightarrow p(N_{CM})} \cdot M_{N_{CM}}$$

Da die globale Transformation nach Einschränkung und Propagierung identisch sein soll, gilt:

$$T_{\rightarrow N_{CM}} = T_{\rightarrow p(N_{adj})} \cdot M_{N_{adj}}' \cdot T_{N_{adj} \rightarrow p(N_{CM})} \cdot M^{cnstr}_{N_{CM}}$$

$$T_{\rightarrow p(N_{adj})} \cdot M_{N_{adj}} \cdot T_{N_{adj} \rightarrow p(N_{CM})} \cdot M_{N_{CM}} = T_{\rightarrow p(N_{adj})} \cdot M_{N_{adj}}' \cdot T_{N_{adj} \rightarrow p(N_{CM})} \cdot M^{cnstr}_{N_{CM}}$$

$$M_{N_{adj}} \cdot T_{N_{adj} \to p(N_{CM})} \cdot M_{N_{CM}} = M_{N_{adj}}' \cdot T_{N_{adj} \to p(N_{CM})} \cdot M_{N_{CM}}^{cnstr}$$

$$M_{N_{adj}}' = M_{N_{adj}} \cdot T_{N_{adj} \to p(N_{CM})} \cdot M_{N_{CM}} \cdot M_{N_{CM}}^{cnstr\,-1} \cdot T_{N_{adj} \to p(N_{CM})}^{-1}$$

Um die globale Transformation $T_{Glob}(N_{CM})$ konstant zu halten, muss also die Matrix des zu adjustierenden Knotens $M_{N_{adj}}$ gesetzt werden auf:

$$M_{N_{adj}} \cdot T_{N_{adj} \to p(N_{CM})} \cdot M_{N_{CM}} \cdot M_{N_{CM}}^{cnstr\,-1} \cdot T_{N_{adj} \to p(N_{CM})}^{-1}$$

Bei dem Resultat oben entspricht $M_{N_{CM}} \cdot M_{N_{CM}}^{cnstr\,-1}$ der Differenz der aktuellen und der eingeschränkten Matrix und $T_{N_{adj} \to p(N_{CM})} \cdot \ldots \cdot T_{N_{adj} \to p(N_{CM})}^{-1}$ der Transformation in das Koordinatensystem des entsprechenden Szenengraphknotens.

Ablauf eines Matrix-Constraint-Mediators:

```
Falls (Matrixfeld geändert)
   Finde Constraint-Parameter passend zu dem neuen Matrix Wert
   Setze Parameter-Felder auf Constraint-Parameter*
   erstelle Neue Matrix, welche durch die eingeschränkten Parameter
      vorgegebenen wird.
   setze Matrixfeld auf die Neue Matrix
   Falls (ursprünglich gesetzte Matrix != Neue Matrix)
      Setze Adjust Feld auf Differenz der Matrizen
Falls (Parameterfeld geändert*)
   Setze Matrixfeld auf den durch das Parameterfeld vorgegebenen
Wert

      *durch die Trigger der Parameter-CM werden diese direkt evaluiert und können die Parameter
        einschränken, bevor diese Werte weiterverwendet werden.
```

Für die genauen Propagierungsregeln der Mediatoren und ihrer Parameter für den Fall von Konflikten oder zyklischen Constraints siehe auch Abschnitt 5.2.

Beispiel: Rotations-CM

Ein Constraint-Mediator für Rotationen schränkt seine überwachte Matrix auf reine Rotationstransformationen ein, d.h. eventuelle nicht zur Rotation gehörige Translationen oder Skalierungen der Matrix werden vom Constraint zurückgewiesen und gegebenenfalls durch das Adjust-Feld an andere Transformationen im Bauteil weitergeleitet. Ist die Rotationsachse frei wählbar, werden von dem Rotationsanteil der Matrix die Rotationsachse und der Rotationswinkel bestimmt und diese als Parameterwerte für das entsprechende Vektor- bzw. Zahlenfeld gesetzt. Ist die Rotationsachse fest vorgegeben, wird der Anteil der Rotation bezüglich dieser Achse errechnet und als Parameterwert für den Winkel der Rotation gesetzt. Die direkte Evaluierung eventueller zusätzlicher Parameter-CM kann den gesetzten Rotationswinkel z.B. durch ein vorgegebenes Maximum wieder einschränken und diesen Parameterwert direkt wieder auf einen mit dem Constraint konformen Wert setzen. Erst dann wird mit diesem neuem Rotationswinkel die neue Rotationsmatrix gebaut und das überwachte Matrix-

feld gesetzt. Alle Anteile der ursprünglich gesetzten Matrix, welche nicht in der gerade gesetzten Matrix enthalten sind, werden durch die Propagierungsregeln der Matrix-Constraint-Mediatoren (siehe oben) an eine Matrix eines im der Szenengraphhierarchie höheren Transformationsknoten weitergeleitet.

5.3.2 Parameter-Constraint-Mediatoren

Zur Überwachung von zusätzlichen Constraints an den Constraint-Parametern werden weitere CM an die jeweiligen Parameter gebunden. Dieses ermöglicht eine flexible Steuerung der Constraints, ohne eine Vielzahl unterschiedlicher CM bereitzuhalten.

Die einfachsten *Const-Parameter-CM* setzen den überwachten Wert auf eine Konstante und überschreiben jede Änderung des Wertes mit diesem konstanten Wert. Dadurch können Matrix-CM, welche mehr Freiheitsgrade erlauben, als durch den konkreten Constraint vorgegeben sind, in diesen Freiheitsgraden gezielt blockiert werden.

MinMax-Parameter-CM für skalare Werte geben einen Wertebereich zwischen einem Minimum und einem Maximum vor, in dem sich der überwachte Wert befinden darf. Bei der Überschreiung der Grenzen wird dieser Wert wieder zurück auf das Minimum bzw. auf das Maximum zurückgesetzt.

Quant-Parameter-CM können eine erlaubte Menge von Werten für das überwachte Feld vorgeben. Bei der Änderung des überwachten Wertes wird der Wert aus der Menge gesucht, welcher dem aktuellen Feldwert bezüglich einer vorgegebenen Metrik am nächsten ist, und dieser gesetzt. Quant-Parameter-CM können skalare Werte einschränken, kommen aber auch bei Vektoren – z.B. bei der Festlegung von Rotationsachsen auf konkrete Werte – zum Einsatz.

Diese Mediatoren können auch mehrere Freiheitsgrade eines externen Mediators zueinander in Beziehung setzen oder in Abhängigkeit voneinander einschränken (siehe am folgenden Beispiel der Freiheitsmatrizen). Die Parameter-CM werden hierbei auch zur Kopplung von Parameterwerten zwischen zwei Mediatoren benutzt. Dabei ist zu beachten, dass die Feldwerte erst auf die vom Constraint vorgegebenen Werte eingeschränkt und dann erst propagiert werden und die jeweiligen Mediatoren, die durch eine Feldwertänderung betroffen sind, benachrichtigt werden müssen.

```
Falls Feldwert1von außen geändert
   Falls Feldwert1 nicht innerhalb der Einschränkungen
      Setze Feldwert1 auf nahesten Wert*
      Setze Feldwert2 entsprechend der Anwendung der
         Kopplungsfunktion auf den eingeschränkten Wert
Falls Feldwert2 von außen geändert
   Falls Feldwert2 nicht im Constraint
      Setze Feldwert2 auf nahesten Wert*
      Setze Feldwert1 entsprechend der Anwendung der
         Kopplungsfunktion auf den eingeschränkten Wert

*durch die Trigger weiterer Parameter-CM werden diese direkt evaluiert und können die
  Parameter einschränken, bevor diese Werte weiterverwendet werden.
```

Parameter-CM müssen aber nicht immer auf gleichartigen Datentypen operieren. Sie können auch komplexere Datentypen in einfache Komponenten aufspalten und zu diesen in Beziehung setzen. Ein Beispiel ist ein *Component-Parameter-CM*, welcher einen Vektor in seine skalaren Komponenten aufspaltet und bei jeder Änderung entweder des Vektors oder einer der Komponenten jeweils den anderen Wert abgleicht. Kompliziertere CM in diesem Bereich zerlegen z.B. Quaternionen in Repräsentationen über Winkel und Achse, bzw. in Eulerwinkel, d.h. in die Rotationen pro Raumachse (vergleiche auch Abschnitt 3.2.3).

Beispiel: Implementierung der Erweiterten Freiheitsmatrizen

Mittels der Matrix-CM und der Parameter-CM lassen sich einfach die Freiheitsmatrizen aus Abschnitt 3.2.5 implementieren. Sei ein Transformationsknoten vorgegeben, dessen Matrix, die relative Transformation der beiden Verbindungsteile repräsentiert, dann kann eine allgemeine Freiheitsmatrix abgebildet werden durch einen Matrix-CM, dessen Translationsvektor durch einen Component-Parameter-CM in seine Komponenten aufgeteilt wird, welche dann jeweils durch MinMax-Parameter-CM eingeschränkt werden. Die Rotation, welche durch den Matrix-CM als Quaternion repräsentiert wird, wird durch einen Parameter-CM zunächst in seine Rotationskomponenten pro Achse zerlegt. Diese werden dann auch entsprechend durch MinMax-Parameter-CM eingeschränkt. Da die MinMax-Parameter-CM nur über Zahlenwerte eingeschränkt werden, wird für den Wert BLOCKED in den Freiheitsmatrizen Null bzw. für FREE ein hinreichend großer Wert gesetzt. Eine Skalierung durch die Transformation wird unterbunden, in dem der Skalierungsanteil der Matrix durch einen Const-Parameter-CM auf die Werte der Einheitsskalierung eingeschränkt wird. Ähnlich ist der Ablauf des Component-Param-CM, der den Translationsvektor des Matrix-CM in seine drei Komponenten zerlegt und auch durch Min-Max-Parameter-CM einschränken lässt.

Abbildung 55 zeigt eine allgemeine Freiheitsmatrix und einen Graphen von Constraint-Mediatoren, der diese implementiert. Im Falle des Rotationsfeldes zerlegt der Euler-Angle-Param-CM die Quaternion des Parameterfeldes weiter in die einzelnen Rotationskomponenten der Achsen. Werden diese Winkelwerte der Parameterfelder des Euler-Angle-Param-CM gesetzt können diese jeweils direkt durch überwachenden Min-Max-Parameter-CM eingeschränkt werden. Aus diesen eingeschränkten

Werten wird am Ende des Ablaufs des Euler-Angle-Param-CM eine neue Quaternion berechnet und als Wert des Rotationsfeldes des Matrix-CM gesetzt.

Ähnlich ist der Ablauf des Component-Param-CM, der den Translationsvektor des Matrix-CM in seine drei Komponenten zerlegt und auch durch Min-Max-Parameter-CM einschränken lässt.

$$\begin{pmatrix} T(+x) & T(-x) & R(+x) & R(-x) \\ T(+y) & T(-y) & R(+y) & R(-y) \\ T(+z) & T(-z) & R(+z) & R(-z) \end{pmatrix}$$

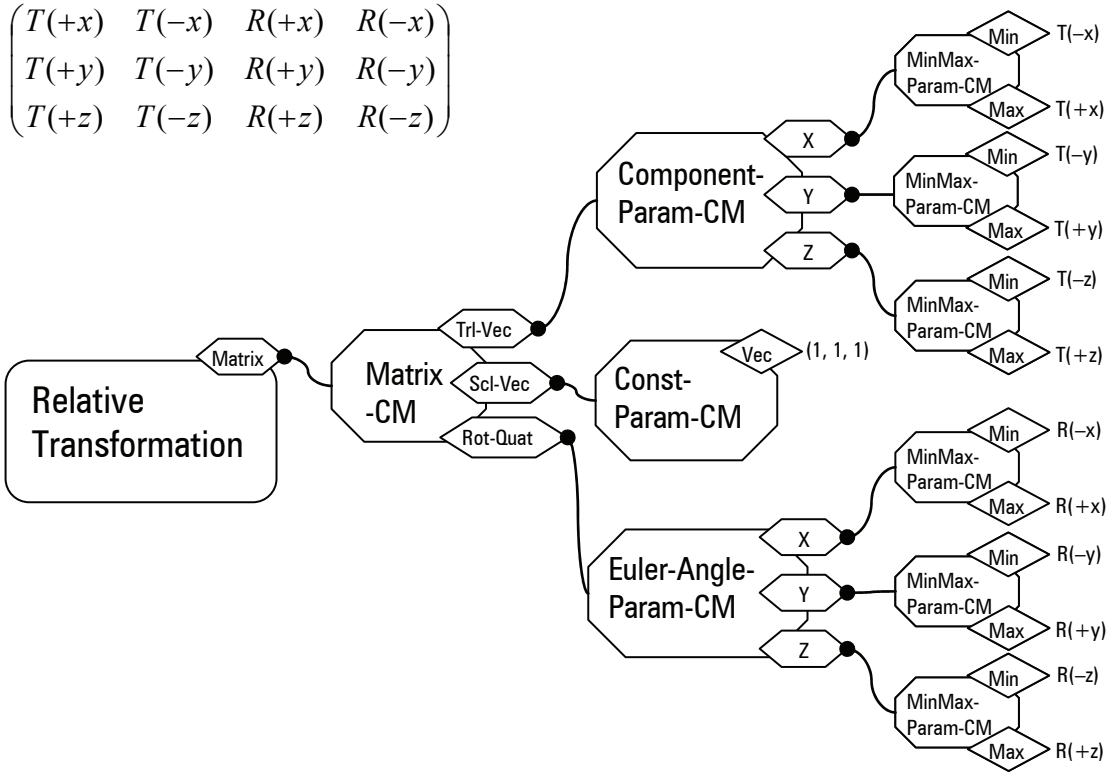


Abbildung 55: Erweiterte Freiheitsmatrix und ihre Implementierung durch Constraint-Mediatoren

Einfacher ist der Fall bei der Skalierung. Hier kann der Const-Parameter-CM, falls sich der Wert des Feldes ändert, den Feldwert ohne weiteres wieder auf den Vektor (1, 1, 1) setzen, da Skalierungen für Freiheitsmatrizen nicht erlaubt sind und direkt unterbunden werden.

Abbildung 56 zeigt ein Ablaufdiagramm der Constraint-Mediatoren aus der Implementation der Freiheitsmatrizen. Hier sieht man, dass eine Veränderung der Matrix von dem Transformationsknoten die Evaluation des Matrix-CM auslöst. Dieser teilt die Transformationsmatrix in ihre einzelnen Komponenten auf und setzt die Werte auf seine entsprechenden Parameterfelder. Jedes Setzen eines Feldes hat wiederum über einen Trigger die direkte Evaluierung des zuständigen Parameter-CM noch innerhalb der Abarbeitung der Matrix-CM zur Folge.

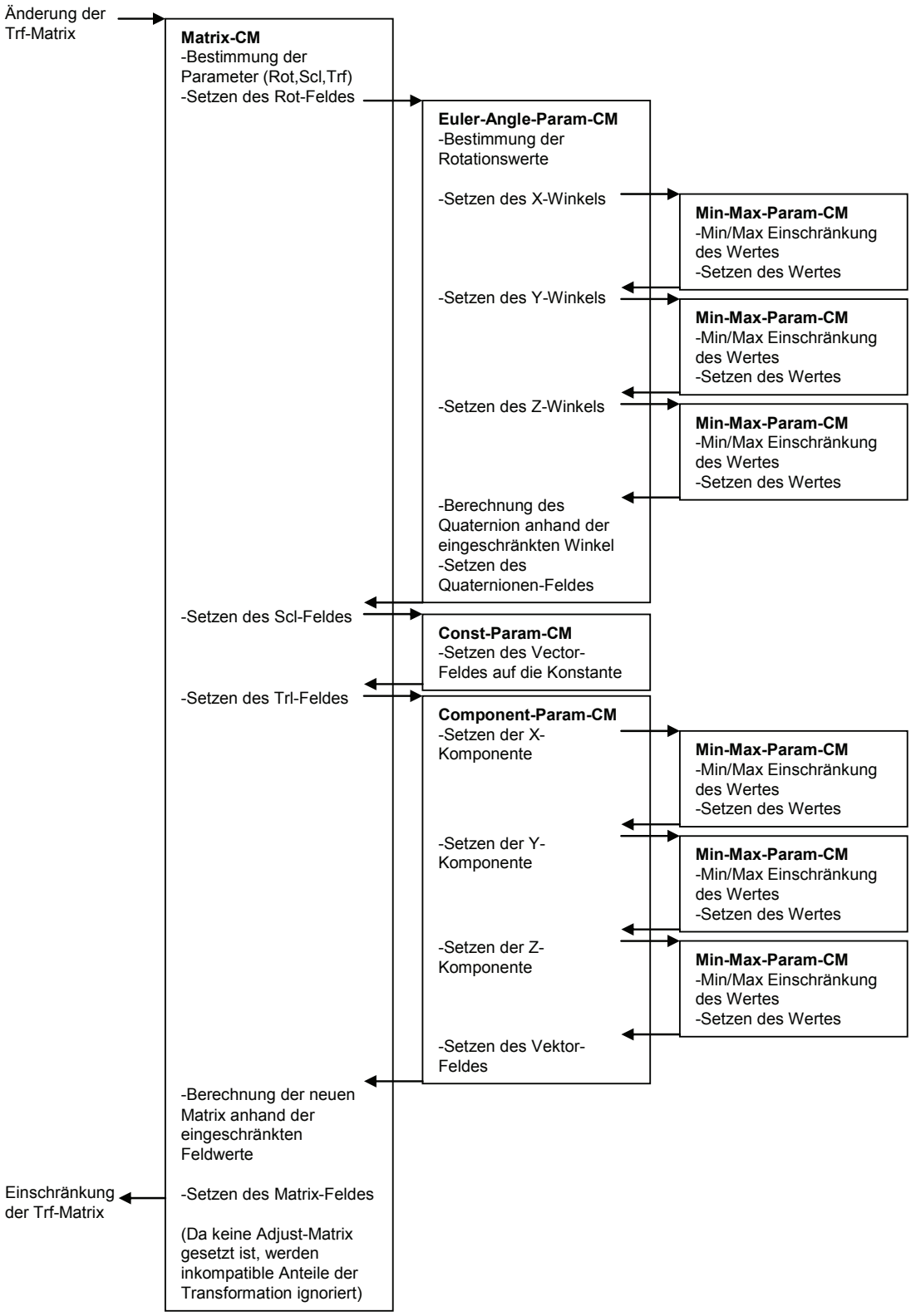


Abbildung 56: Ablaufdiagramm der CM aus Abbildung 55

Die Kopplungen der Freiheitsmatrizen können einfach durch zusätzliche Connect-Parameter-CM realisiert werden, falls die Kopplungen symmetrisch sind, d.h. die negativen und positiven Werte mit dem gleichen Faktor gekoppelt werden. Für asymmetrische Kopplungen, in denen die Bewegung in eine Richtung einen anderen Kopplungsfaktor hat als die Bewegung in die andere Richtung, kann problemlos ein spezieller CM mit unterschiedlichen Faktoren für die beiden Propagierungsrichtungen realisiert werden. Abbildung 57 zeigt eine symmetrische Kopplung von Translation und Rotation bezüglich der X-Achse, wie sie z.B. für eine Schraubverbindung eingesetzt wird.

$$c = \{(<1,1>,<1,3>,r),(<1,2>,<1,4>,r)\}$$

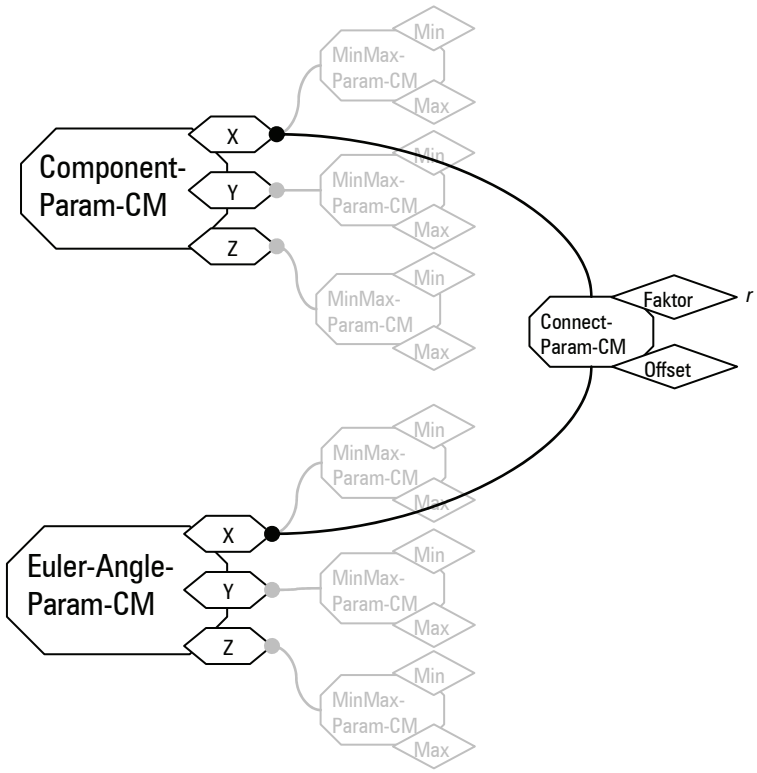


Abbildung 57: Beispiel für eine Kopplung von Parametern einer Freiheitsmatrix

5.3.3 Port-Matrix-Constraint-Mediatoren

Verbindungen an vordefinierten Verbindungsstellen sind modelliert als zwei Verbindungspunkte, denen je nach Verbindungstyp unterschiedliche Bewegungs-Freiheitsgrade zugeschrieben werden können, um die Bewegungsmöglichkeiten der resultierenden Port-Verbindung simulieren zu können.

In der ursprünglichen Definition der Ports wurden die Freiheitsgrade durch die Verbindungsbeziehung mittels der erweiterten Freiheitsmatrizen definiert (siehe Abschnitt 3.2.5). Die erlaubten Bewegungen der Verbindung wurden durch einen externen Prozess bei der Manipulation überwacht. In der Modellierung durch Constraint-Mediatoren ergeben sich die möglichen Bewegungen durch die Freiheitsgrade der lokalen Verbindungsstelle(n) eines Ports. Dadurch kann ein Port direkt im

Szenengraph die Informationen und die Constraints für die Verbindung bei sich tragen.

Dabei sind unterschiedliche Aufteilungen der Freiheitsgrade auf die beiden Verbindungspunkte möglich. Oft ist ein Verbindungspunkt als unbeweglich modelliert und der Port des Verbindungspartners trägt sämtliche Freiheitsgrade. Aber auch eine verteilte Modellierung der Freiheitsgrade oder sogar eine stellenweise redundante Modellierung, sodass Freiheitsgrade von beiden an der Verbindung beteiligten übernommen werden können, ist sinnvoll. Ein Beispiel hierfür ist die Simulation von zwei Flächen, welche so miteinander verbunden sind, dass immer ein Teil der beiden Flächen überlappen soll (siehe am Beispiel der Plane-Ports unten).

Diese Mediatoren überwachen die lokalen Matrizen an den Verbindungsstellen eines Bauteils. Die lokalen Matrizen der Ports können durch die Bewegung der mit ihnen verbundenen Bauteile verändert werden. Die Freiheitsgrade dieser Port-Matrizen werden entsprechend der kinematischen Paare aus Abschnitt 3.2.5 mit Hilfe von Matrix-Constraint-Mediatoren modelliert. Falls die Änderung der Matrixwerte nicht mit dem Port-Constraint übereinstimmt, kann der CM den nicht kompatiblen Teil der Änderung über die Adjust-Matrix an übergeordnete Transformationsknoten propagieren, die wiederum durch Matrix-CM überwacht werden können.

In den folgenden Abschnitten werden nur Einzelverbindungen zwischen zwei Ports betrachtet. Die Behandlung von mehreren Verbindungen an einem Port, z.B. für Flächenports oder Extrusionsports, wird in Absatz 5.4 vorgestellt.

Feste Port-Verbindungen mit Fixed-Ports

Solange die Verbindung aktiv ist, ist keine Relativbewegung zwischen den beiden verbundenen Bauteilen möglich. Die Verbindungsstellen an den Bauteilen bestimmen eindeutig die Lage der verbundenen Bauteile zueinander. Dies ist eine häufige Verbindungsart von Bauteilen, da oft eine Relativbewegung der Bauteile nicht gewünscht ist.

Beide Ports bilden jeweils eine feste Verbindungsstelle (*Fixed-Port*, 0+0 FG) an den beiden Bauteilen. Der entsprechende Port-Matrix-CM überwacht die lokale Matrix der Verbindungsstelle und schränkt sie auf fest vorgegebene Werte ein. Eine Veränderung dieser Matrix, z.B. durch die Bewegung des durch den Port verbundenen Bauteils, wird direkt an den in der Bauteilhierarchie nächst höheren Mediator – bis hin zur Position des gesamten Bauteils – weitergegeben. Manche Extrusion-Ports (siehe unteren Abschnitt) werden als fester Port modelliert, da sie nur eine sinnvolle Stellung bei der Verbindung haben und die Freiheitsgrade der Verbindung durch den Verbindungspartner modelliert werden.

Flächenverbindung über Plane-Ports

Eine weitere, häufige Verbindungsart ist die Verbindung von zwei Flächen. Das entsprechende kinematische Paar ist das „Plane-Pair“. Diese Verbindung hat somit drei Freiheitsgrade: Zwei translatorische und einen rotatorischen, dessen Achse senkrecht zu den beiden translatorischen liegt. Die Freiheitsgrade können während der Interaktion je nach Bearbeitungsschritt noch weiter eingeschränkt werden, bis hin zu einer festen Verbindung.

Beide Ports der Bauteile sind jeweils mit einer Verbindungsstelle modelliert, welche sich auf der Fläche bis zum Rand bewegen und außerdem um die Achse senkrecht zur Fläche drehen kann (*Plane-Port*, 2+1 FG). Da beide Ports drei Freiheitsgrade besitzen, sind die drei Freiheitsgrade der Verbindung redundant modelliert. Theoretisch könnte ein Port als Fixed-Port definiert sein und der andere die drei Freiheitsgrade modellieren. Der Vorteil der redundanten Modellierung liegt in der Möglichkeit, eine Überlappung der beiden Portgeometrien sicherzustellen. Die beiden Flächen können beliebig aufeinander verschoben werden. Dadurch, dass die bewegliche Verbindungsstelle sich immer innerhalb von beiden Portflächen befindet, ist aber eine Überlappung garantiert (siehe Abbildung 58). Ein spezieller Matrix-CM, der zwei Translations-Freiheitsgrade und eine dazu senkrechte Rotation erlaubt, bindet an die Matrix dieses Ports. Ein zusätzlicher Parameter-Constraint-Mediator überwacht, dass die beiden Translationsparameter des CM sich innerhalb der zum Port gehörigen Fläche befinden. Es sind Parameter-CM für einfache Flächen wie Quadrate und auch für konvexe Polygone definiert. Für kompliziertere Flächen können auch beliebige CM eingesetzt werden, welche die zwei Parameter der Fläche entsprechend einschränken.

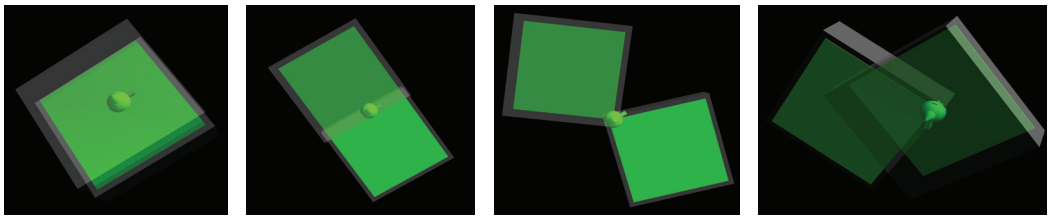


Abbildung 58: Plane-Port-Verbindung

Abbildung 58 zeigt zwei über Plane-Ports verbundene Quader in unterschiedlichen Stellungen zueinander. Die Bauteile sind halb transparent dargestellt, um die beiden Verbindungsstellen der Ports sichtbar zu machen. Die beiden Verbindungsstellen sind als Kugel visualisiert und liegen bei aktiver Verbindung immer exakt aufeinander. Um entsprechend einer realen Verbindung eine gewisse Überschneidung der beiden Flächen zu garantieren, sind die Planeports so definiert, dass sich die beweglichen Verbindungsstellen nicht ganz bis zum Rand der Fläche verschoben werden dürfen.

Auch nicht planare Formen können durch diese Parameter-CM vorgegeben werden, was eine einfache Erweiterung des oben angegebenen Matrix-CM um den dritten Translationsparameter erfordert. Dies ermöglicht die einfache Realisierung von Verbindungsports auf beliebig geformten konvexen Oberflächen (*Surface-Ports*), wenn dazu der Rotationsfreiheitsgrad auf die Flächennormale der Portoberfläche angepasst wird. Mittels dieser Surface-Ports können z.B. zwei konvexe Objekte beliebig aneinandergehaftet werden.

Zylindrische Steckverbindung mit einem Hole- und einem Shaft-Port

Diese Verbindungsart entsteht durch die Verbindung eines runden Stabes mit einem Loch. Ein runder Querschnitt der beiden Verbindungsgeometrien ergibt hierbei ein „Cylindrical-Pair“ mit jeweils einen rotatorischen und einen translatorischen Freiheitsgrad, deren Achsen parallel zueinander sind.

Der Nehmerport ist – wie in den beiden folgenden Fällen – modelliert durch eine Verbindungsstelle, die sich vom äußeren Ende des Lochs im Zentrum des Quer-

schnitts bis zum Ende des Loches bewegen kann (*Hole-Port,* 1 FG). Der Geberport ist in diesem Fall modelliert durch eine Verbindungsstelle, die sich entlang des Schaftes bewegen und außerdem um die Schaft-Achse rotieren kann (*Shaft-Port,* 1+1 FG). Also sind auch die zwei Freiheitsgrade der Portverbindung redundant von beiden Ports modelliert. Wie bei den Planports lässt sich dadurch relativ komfortabel die Überlappung der Kapazitäten der verbundenen Ports sicherstellen.

Ein spezieller Matrix-CM mit jeweils einem Freiheitsgrad für Rotation und Translation überwacht die Matrix des Ports. Die Länge der Translation wird dabei noch durch einen MinMax-Parameter-CM überwacht. Der Minimum-Wert des Parameter-CM ergibt sich aus einer geforderten Überlappung der Ports, damit diese als verbunden angesehen werden können. Der Maximal-Wert hingegen muss die Kapazitäten der beiden verbundenen Ports berücksichtigen. Wenn Geberport und Nehmerport jeweils nur an einer Seite offen sind, ergibt sich dieser Wert einfach aus dem Minimum der beiden Portkapazitäten. Komplexere Betrachtungen bezüglich der Kapazitäten von Extrusionsports auch bei Mehrfachverbindungen werden in Abschnitt 5.4 angestellt.

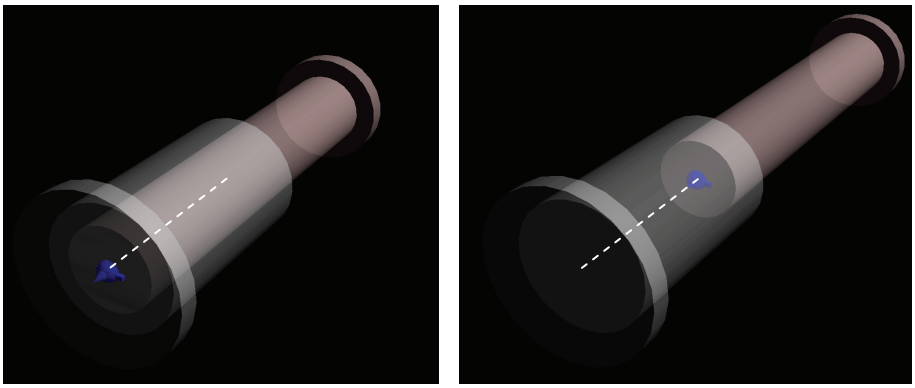


Abbildung 59: Extrusion-Port-Verbindung

Abbildung 59 zeigt einen Hole-Port und einen Shaft-Port und die Verbindung der beiden in zwei Stellungen, wobei beide Ports nur eine offene Seite haben. Um die Visualisierung der beweglichen Verbindungsstelle der verbundenen Ports sichtbar zu machen, wurden auch hier die Bauteile halb transparent dargestellt. Die Verbindungsstellen der beiden Ports liegen auch hier, wie es bei allen aktiven Portverbindungen der Fall ist, exakt übereinander. In diesem Beispiel bestimmt der Hole-Port die gesamte erlaubte, relative Translation der Verbindung[12], welche in der Abbildung als gestrichelte Linie eingezeichnet ist. Für die Verbindungsstelle des Schaftes ist daher der Translatorische Freiheitsgrad nicht freigegeben und die Verbindungsstelle ist fest am Ende des Schaftes platziert. Die unbeschränkte, relative Rotation der beiden Bauteile ist durch den rotatorischen Freiheitsgrad des Nehmerports modelliert.

Wenn einer der beiden oder beide Zylinder des Geber- oder Nehmerports an beiden Seiten offen sind, werden an beiden Enden der Verbindungselemente Ports modelliert. Hierbei beeinflussen sich die beiden Kapazitäten der Ports und im Allgemeinen können beliebig viele Verbindungen an diesen Ports möglich sein, wie z.B. mehrere Ringe auf einem Stab. Diese komplexeren Betrachtungen werden an gesonderter Stelle (siehe Abschnitt 5.4.2) gemacht.

[12] Wie in Abschnitt 5.4.2 gezeigt, definiert bei zwei, nur an einer Seite offenen, Ports der kürzere der beiden die gesamte Kapazität.

Gewinde-Schraubverbindung mit einem Hole- und einem Thread-Port

Sind beide Verbindungspartner mit einem Gewinde versehen, müssen die beiden Freiheitsgrade der zylindrischen Steckverbindung je nach Gewindesteigung entsprechend aneinander gekoppelt sein, sodass ein „Helical-Pair“ entsteht.

Während der Nehmerport, wie im Falls vorher, ein Hole-Port mit einem translatorischen Freiheitsgrad ist, wird der Geberport mit dem gleichen Matrix-CM wie bei der zylindrischen Verbindung modelliert (*Thread-Port,* 1+1 FG gekoppelt). Die zusätzliche lineare Kopplung der beiden Freiheitsgrade wird durch einen Parameter-Constraint-Mediator realisiert, der neben der Kapazität die funktionale Abhängigkeit der Parameter überwacht. Der Faktor der linearen Kopplungsfunktion definiert das Verhältnis zwischen dem rotatorischen und dem translatorischen Freiheitsgrad und somit die Gewindesteigung der Schraube.

Wenn die Schraubbewegung aber für die Modellierung der Bauteilverbindung nicht wichtig ist, werden Schraubverbindungen oft auch durch die einfacheren zylindrischen Steckverbindungen (siehe oberer Abschnitt) modelliert.

Allgemeine Extrusionen mit einem Hole- und einem Extrusion-Port

Bei nicht runden Querschnitten der Stangen bzw. Löcher fällt die Möglichkeit der Drehung der verbundenen Bauteile weg. Es entsteht ein „Prismatic-Pair“. Hierbei ist zu beachten, dass je nach Rotationssymmetrien der Querschnitte beim Zusammenbau der beiden Teile eventuell verschiedene Drehungen möglich sind, die aber während einer aktiven Verbindung nicht zu verändern sind.

Geberport und Nehmerport sind wie vorher, aber ohne rotatorischen Freiheitsgrad während der aktiven Verbindung modelliert (*Extrusion-Port,* 1+0 FG). Der Parameter-CM, der die Freiheitsgrade überwacht, kann aber die Rotation bei der Erstellung der Verbindung je nach Rotationssymmetrie der Extrusionsgeometrie für fest definierte Werte freigeben.

Scharnier-Gelenk mit zwei Hinge-Ports

Diese Art von einfachen Scharnier- oder Dreh-Gelenk tritt häufig bei der Modellierung von kinematischen Ketten auf. Diese Gelenke haben einen einzelnen, rotatorischen Freiheitsgrad und sind somit als „Revolute-Pair“ anzusehen.

Die Port-Matrix wird durch einen einfachen Matrix-CM mit einem Rotations-Freiheitsgrad überwacht, deren Parameter auf den gültigen Bereich des Scharnierwinkels eingeschränkt ist (*Hinge-Port,* 0+1 FG).

5.3.4 Port-Verbindungs-Constraint-Mediatoren

Die Verbindung von zwei Bauteilen wird über die Verbindung von entsprechenden Ports der Bauteile realisiert. Während die relativen Positionen der Verbindungsstellen von den im vorigen Abschnitt beschriebenen Port-Constraint-Mediatoren kontrolliert werden, wird die Verbindung zwischen zwei Ports über einen Mediator überwacht, der die globale Position und Orientierung der Ports überwacht und einander angleicht. Dafür muss diesem CM die absolute Transformation des Ports in globalen Koordinaten zur Verfügung stehen. Im Szenengraphen müssen dafür die Transforma-

tionsmatrizen des gesamten Zweiges des Szenengraphen aufmultipliziert werden. Diese Funktionalität wird entweder durch den universellen Szenengraph-Layer (siehe Abschnitt 4.12) implementiert, oder – falls das eingesetzte VR-Tool eine entsprechende Funktion bereitstellt – durch den USG-Layer gekapselt.

Die Differenz in der globalen Transformation der Ports wird zunächst direkt an die lokale Matrix eines Ports propagiert. Falls der Port diese Bewegung nicht durch eine Veränderung der eigenen Matrix innerhalb seines Constraints kompensieren kann, wird von dort aus diese Veränderung an andere CM im Bauteil propagiert. Enthält das Bauteil gar keine beweglichen Teile kann die Bewegung direkt an die Transformation des gesamten Bauteils weitergegeben werden.

Der CM hat zwei Referenzfelder für die beiden zu überwachenden Ports (P_1, P_2) und zwei Referenzen auf die Matrizen (M_1, M_2) der anzupassenden Port-Transformationsknoten.

Als erstes wird der Constraint überprüft, ob die globalen Transformationen der Ports übereinstimmen. Ist das nicht der Fall, muss die Transformation eines Ports so verändert werden, dass die globalen Transformationen wieder übereinstimmen. Dafür wird die Differenz der globalen Transformationen $T_{Glob}(P_1)^{-1} \cdot T_{Glob}(P_2)$ auf die Matrix des Ports aufmultipliziert. Die Bestimmung, welche der beiden Portmatrizen zu verändern ist, erfolgt über die letzte Veränderung der jeweiligen globalen Transformation. Die Veränderung der passiven Verbindungsstelle erlaubt die Propagierung von Bewegungen verbundener Bauteile. Abb. X zeigt vereinfacht den Ablauf innerhalb eines Port-Verbindungs-CM. Sollten die globalen Transformationen beider Bauteile sich geändert haben – falls z.B. beide Teile der Verbindung gleichzeitig bewegt werden – muss eine übergeordnete Instanz über die Propagierungsrichtung entscheiden (siehe Abschnitt 5.2).

Falls $T_{Glob}(P_1) \neq T_{Glob}(P_2)$

Falls $T_{Glob}(P_1)$ geändert

Setze $M_1 = M_1 \cdot T_{Glob}(P_1) \cdot T_{Glob}(P_2)^{-1}$

Falls $T_{Glob}(P_2)$ geändert

Setze $M_2 = M_2 \cdot T_{Glob}(P_2) \cdot T_{Glob}(P_1)^{-1}$

Durch das setzen der Portmatrix kann das überwachte Constraint – die Identität der globalen Transformationen der beiden Ports – wie im Folgenden gezeigt, wieder eingehalten werden:

Sei $T_{Glob}(P_1)$ die globale Transformation von P_1 und

$T_{GParent}(P_1)$ die globale Transformation des SG-Elter von P_1 und

M_1 die Matrix von P_1

Dann gilt:

$T_{Glob}(P_1) = T_{GParent}(P_1) \cdot M_1$

durch $M_1' = M_1 \cdot T_{Glob}(P_1)^{-1} \cdot T_{Glob}(P_2)$ gilt:

$$T_{Glob}(P_1)' =$$
$$T_{GParent}(P_1) \cdot M_1' =$$
$$T_{GParent}(P_1) \cdot M_1 \cdot T_{Glob}(P_1)^{-1} \cdot T_{Glob}(P_2) =$$
$$T_{Glob}(P_1) \cdot T_{Glob}(P_1)^{-1} \cdot T_{Glob}(P_2) =$$
$$T_{Glob}(P_2)$$

Der Port-Verbindungs-CM propagiert also die gesamte Bewegung in die lokale Portmatrix des nicht bewegten Bauteils. Da im Allgemeinen die Portmatrix selber Constraints unterliegt, wird diese Veränderung durch den Port-Matrix-CM, der die Werte der Matrix des Ports überwacht, im Bauteil weiter propagiert. Hierbei können weitere über Parameter veränderbare Transformationen im Bauteil angepasst werden, oder das ganze Bauteil wird so transformiert, dass die Portverbindung erhalten bleibt. Wenn eine Transformation des gesamten Bauteils nicht erwünscht ist und z.B. durch weitere Constraints verhindert wird, kann der mit den Constraints für dieses Bauteil inkompatible Teil der Bewegung durch den Port-Verbindungs-CM an das ursprünglich bewegte Bauteil zurück propagiert werden (siehe Beispiel in Abschnitt 6.3.5). Dadurch kann sich allein aufgrund der Verbindung eines Bauteils mit einem anderen eine Einschränkung der Bewegungsmöglichkeiten des Bauteils ergeben.

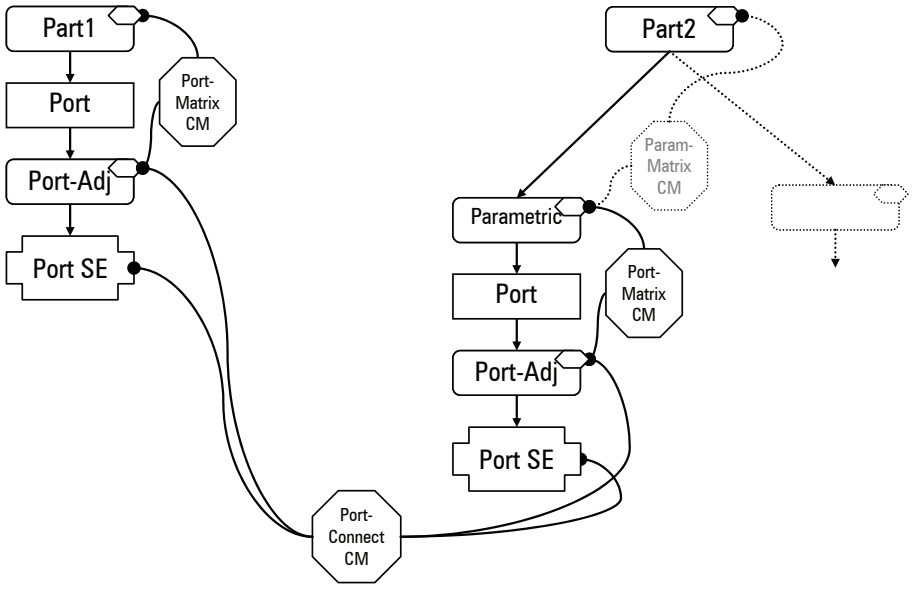


Abbildung 60: Szenengraph mit Port-Connect-Mediator und Port-Matrix-Mediatoren

5.3.5 Skalierungs-Constraint-Mediatoren

Bei komplexen, inhomogenen Skalierungen von Objekten, welche in Szenengraphen repräsentiert sind, müssen die Skalierungsvorschriften auch durch Constraints überwacht werden. Neben den Matrix-Constraint-Mediatoren, welche die Matrix der entsprechenden Transformation auf eine Skalierung innerhalb der vorgegebenen Parameter einschränken, müssen weitere CM das Zusammenspiel mehrerer Ebenen von Skalierungen im Szenengraph überwachen. Da durch die Anwendungslogik des Szenengraphmodells die aufaddierten Transformationen der Elternknoten automa-

tisch an die Kindknoten vererbt werden, muss für den Fall, dass ein Elternknoten ohne Einfluss auf seine Kinderknoten skaliert werden soll, eine inverse Skalierung der Kinderknoten aufgebaut werden. Dafür müssen diese Skalierungs-Constraint-Mediatoren die Skalierungswerte der Elternknoten kennen, um diese ganz oder teilweise durch eine entsprechende Skalierung des Kindknotens aufheben zu können. Ein Anwendungsbeispiel ist die Skalierung einer Leiste mit runden Löchern. Hierbei sind die Löcher sinnvollerweise als Kinderknoten des Leistenkörpers definiert, schließlich sollen sie ja z.B. bei einer Translation der Leiste mitbewegt werden. Allerdings sollte eine Skalierung der Leiste die Radien der Löcher konstant lassen, da sie z.B. als Verbindungsstellen für eine bestimmte Sorte von Schrauben dienen sollen. Noch deutlicher wird dies bei einer inhomogenen Skalierung der Leiste entlang der Längsrichtung, wodurch bei der Übertragung der Skalierung die Löcher oval würden.

5.4 Mehrfachverbindungen und gekoppelte Kapazitäten

Da die Modellierung von Verbindungspartnern mittels beweglicher Verbindungspunkte, wie sie im vorherigen Abschnitt vorgestellt wurden, erstmal nur Einfachverbindungen von Ports zulassen, muss für Mehrfachverbindungen an einem Port das Konzept der Verbindungsstelle erweitert werden. In den vorherigen Beispielen betrifft dieses die Flächenports und die Extrusionsports, wenn sie an zwei Enden offen sind. Zusätzlich ergibt sich bei den Extrusionsports eine Abhängigkeit der Kapazitäten, zum einen der Verbindungen gegenüberliegender Ports an den beiden Enden einer Extrusionsgeometrie, zum anderen der zusätzlichen Verbindungsstellen für multiple Verbindungen von den Kapazitäten und Belegungen der bereits vorhandenen Verbindungen dieses Ports.

5.4.1 Mehrfachverbindungen an Plane-Ports

Um mehrere Verbindungen an einem Flächenport etablieren zu können, prägen Planeports jeweils eine neue Verbindungsstelle aus, sobald an der aktuellen freien Verbindungsstelle eine Verbindung aufgebaut wird. Hier entsteht das Problem, dass der Bereich des Flächenports, welcher schon von dem ersten Verbindungspartner belegt wird, nicht mehr für die weiteren Verbindungen benutzt werden darf. Eine mögliche Modellierung wäre, die Kapazität des Ports mit den Mehrfachverbindungen so anzupassen, dass belegte Teile nicht mehr als belegbare Fläche auftauchen. Mehrere Gründe sprechen aber gegen so eine Modellierung. Zum einem besteht die so entstehende Fläche im Allgemeinen aus einem konkaven Polygon mit Löchern und der CM für die Portverbindung müsste einen komplexen und relativ rechenaufwendigen Test für die Überlappung der jeweiligen Kapazitätsgeometrien durchführen. Zum anderen sind die Bewegungen der verbundenen Bauteile nicht nur durch die Überlappung der Portgeometrien, sondern vor allem durch die Kollision der gesamten Bauteilgeometrien untereinander eingeschränkt.

Besonders der letzte Punkt legt nahe, diese Einschränkung nicht in den Constraints der Portverbindungen zu modellieren, sondern durch eine davon unabhängige Kollisionserkennung. Da eine Kollisionserkennung an mehreren Stellen des Systems benötigt wird – z.B. auch bei der Interaktion mit den Bauteilen, wobei Durchdringungen erkannt und vermieden oder Verbindungen bei Kollision von zwei Bauteilen etabliert werden – wird die Kollisionserkennung und Vermeidung dieser externen Komponente überlassen. Daher bleibt die Kollisionserkennung, welche zwischen zwei

über CM verbundenen Bauteilen generell deaktiviert wird, zwischen den Bauteilen, die mit der gleichen Fläche verbunden sind, weiterhin aktiv.

Auch eine Kollisionserkennung ließe sich über das Konzept eines Constraint-Mediators, der Informationen über die Geometrien der Bauteile hat und die Transformationsmatrizen dieser Bauteile überwacht, realisieren. Da CM aber als lokale Constraints ausgelegt sind, die hauptsächlich schnell zu berechnende Bedingungen überwachen und außerdem sehr gute Systeme zur Kollisionserkennung vorhanden sind, ist hier der Weg über die Anbindung einer externen Simulationskomponente vorzuziehen.

5.4.2 Mehrfachverbindungen an Extrusion-Ports

Auch bei Extrusionen werden Mehrfachverbindungen über zusätzliche Verbindungspunkte realisiert, die bei der Verbindung eines freien Verbindungspunktes hinzugefügt werden. Allerdings kann man in diesem Fall die einzelnen Extrusion-Port-Typen je nach Verbindungspartner unterschiedlich behandeln. So können weitere Mehrfachverbindungen an einem Extrusion-Port nur auftreten, wenn der aktuelle Verbindungspartner ein an beiden Seiten offener Port ist. Ansonsten wird durch das verbundene Bauteil der Port abgedeckt. Beispiele dafür sind bei Löchern eine Schraube mit Kopf, welche verhindert, weitere Schrauben mit diesem Loch zu verbinden, oder bei einer Stange eine Kappe, welche den Port an diesem Ende für andere Ringe oder Kappen blockiert. Außerdem kann ein Bauteil sämtliche Kapazitäten des Ports konsumieren. Auch dann ist eine weitere Verbindung an diesem Port nicht möglich.

5.4.3 Kopplung der Kapazitäten von Extrusion-Ports

Die Ports werden über Verbindungsstellen realisiert, bei denen je nach Typ des Ports bestimmte Bewegungen eingeschränkt werden. Bei Extrusion-Ports ergeben sich nach den Betrachtungen in Kapitel 3.2.5 jeweils ein rotatorischer und ein translatorischer Freiheitsgrad. Die Rotation ist je nach Art der Extrusionsgeometrie entweder fest, frei oder an die Translation gekoppelt. Bei der Translation ergeben sich je nach Porttyp und den Kapazitäten der verbundenen Ports unterschiedliche Bewegungsfreiheiten. Erschwert werden die Berechnungen durch die gegenseitigen Abhängigkeiten der Bewegungsfreiheiten bei Mehrfachverbindungen eines Ports, bzw. bei der gleichzeitigen Belegung von zwei gegenüberliegenden Verbindungsstellen, welche zu einem Extrusion-Port gehören. Zunächst die Betrachtung eines einfach verbundenen Extrusion-Ports.

Satz 5.1 (**Translation bei einfach verbundenen Extrusionsports**): Die Translation bei Extrusion-Ports, welche jeweils nur mit einem anderen Port verbunden sind, lassen sich wie folgt berechnen:

Sei

P ein Extrusion-Port, C_P seine Kapazität,

p1 die erste Verbindungsstelle von P,

P' ein zu P kompatibler Extrusion-Port, $C_{P'}$ seine Kapazität,

p'1 die erste Verbindungsstelle von P',

T_{p1}, T_{p2} die Translation von p1 bzw. p2,

$C_{p1}^{Min}=0$, $C_{p'1}^{Min}=0$, die untere Schanke für T_{p1}, bzw., T_{p2}

$C_{p1}^{Max}=C_P$, $C_{p'1}^{Max}=C_{P'}$, die obere Schanke für T_{p1}, bzw.,

P nur verbunden mit P', über p1 bzw. p'1,

dann gelten für T_{p1} und T_{p2} Einschränkungen gemäß folgender Tabelle:

	P einseitig und $C_{p1}^{Max} \leq C_{p'1}^{Max}$	P einseitig und $C_{p1}^{Max} > C_{p'1}^{Max}$	P zweiseitig
P' einseitig	$Tp_1 \in [C_{p1}^{Min} \quad C_{p1}^{Max}]$ $Tp'_1 = C_{p'1}^{Min}$	$Tp_1 = C_{p1}^{Min}$ $Tp'_1 \in [C_{p'1}^{Min} \quad C_{p'1}^{Max}]$	$Tp_1 = C_{p1}^{Min}$ $Tp'_1 \in [C_{p'1}^{Min} \quad C_{p'1}^{Max}]$
P' zweiseitig	$Tp_1 \in [C_{p1}^{Min} \quad C_{p1}^{Max}]$ $Tp'_1 = C_{p'1}^{Min}$	$Tp_1 \in [C_{p1}^{Min} \quad C_{p1}^{Max}]$ $Tp'_1 = C_{p'1}^{Min}$	$Tp_1 \in [C_{p1}^{Min} \quad C_{p1}^{Max}]$ $Tp'_1 \in [C_{p'1}^{Min} \quad C_{p'1}^{Max}]$

Die Summe der beiden Translationen Tp_1 und Tp'_1 ergeben die Eindringtiefe Ip_1 des verbundenen Ports. Die initiale, minimale Eindringtiefe ist in allen Fällen gleich Null. In dieser Konfiguration befinden sich die beiden Extrusion-Ports genau Kante an Kante. Um eine minimale Eindringtiefe der Ports größer als Null vorzugeben, kann die untere Schranke C_{p1}^{Min} und $C_{p'1}^{Min}$ auf dieses Mindestmaß gesetzt werden. Dieses Mindestmaß ist bei der Definition der Ports dann mit anzugeben.

Wenn beide Ports einseitig begrenzt sind, ist für den kleineren der beiden Ports die Translation bis zu der oberen Grenze entsprechend seiner Kapazität freigegeben, da in diesem Fall die geringere Kapazität die mögliche Eindingtiefe bestimmt. Für den Fall, dass ein einseitiger und ein zweiseitiger Port verbunden werden, bestimmt die Kapazität des einseitigen Ports die Eindringtiefe, da bei einem zweiseitigen Port die Kapazität keine Beschränkung für die Bewegung darstellt. Sind beide Ports zweiseitig, können beide Kapazitäten der Ports ausgenutzt werden. Damit kann das verbundene Bauteil über die Kapazität des lokalen Ports hinausgeschoben werden.

Diese Betrachtungen sind unabhängig davon, welcher der beiden Extrusion-Ports der Nehmerport bzw. der Geberport ist. Auch sind die Gleichungen in der Tabelle in Satz

5.1 symmetrisch bezüglich der Vertauschung von P und P'. Daher können beide Ports unabhängig voneinander nach den gleichen Regeln definiert werden.

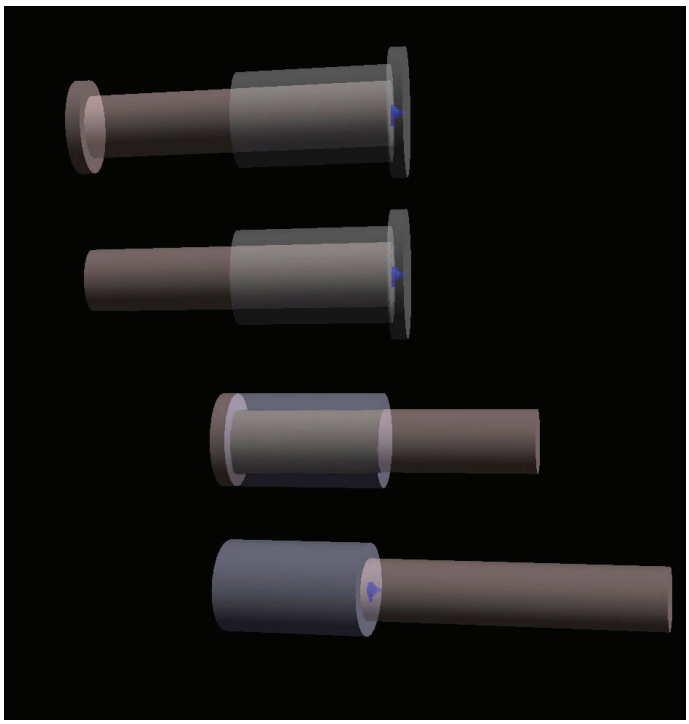


Abbildung 61: Maximale Eindringtiefen bei längerem Geberport

Abbildung 61 zeigt von links nach rechts die unterschiedlichen maximalen Eindringtiefen je nachdem, ob beide Ports einseitig, der größere zweiseitig und der kleinere einseitig bzw. umgekehrt, oder beide zweiseitig sind.

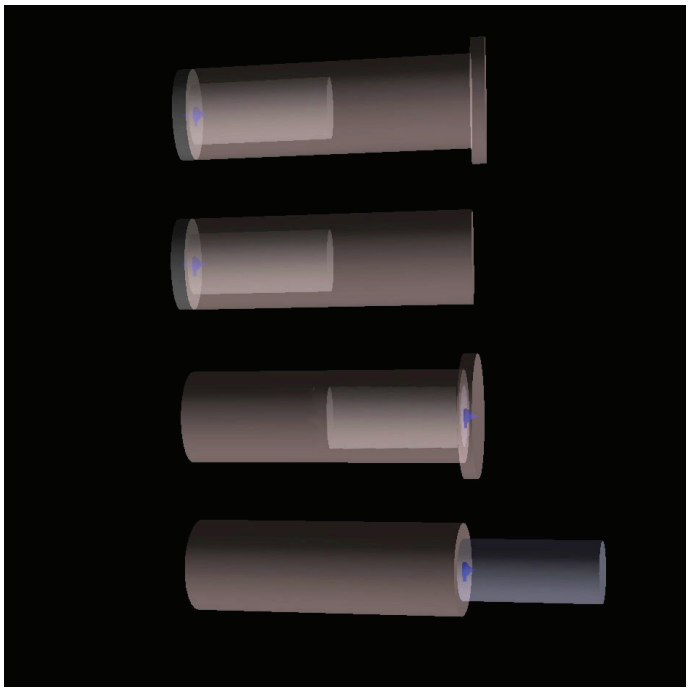


Abbildung 62: Maximale Eindringtiefen bei längerem Nehmerport

Abbildung 62 zeigt dieselben vier Konfigurationen von Abbildung 61 für den Fall, dass der Nehmerport länger als der Geberport ist.

Da die Translationswerte T_{p1} und $T_{p'1}$ Parameter der Port-Matrix-CM, welche die Freiheitsgrade der beiden Verbindungsstellen p1 bzw. p'1 überwachen, sind, können diese Werte über jeweils einen zusätzlichen MinMax-Parameter-CM eingeschränkt werden. Abbildung 63 zeigt den Aufbau der Constraint-Mediatoren für die Realisierung des Constraints an der Verbindungsstelle p1 entsprechend Satz 5.1. Benutzt wird hierfür neben dem zusätzlichen MinMax-Parameter-CM ein Switch-Knoten der visuellen Programmierung, der anhand der Eigenschaften der verbundenen Ports den korrekten Wert für die obere Begrenzung der Translation auswählt.

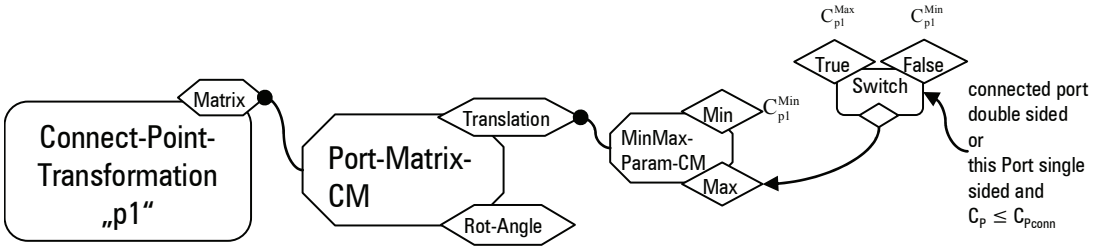


Abbildung 63: Aufbau der Constraint-Mediatoren für die einfache Kapazitätsprüfung einer Verbindungsstelle

Falls der Verbindungspartner ein geschlossener Port ist, sind keine weiteren Verbindungen an dieser Seite des Ports mehr möglich. Auch ein zweiseitiger Port als Verbindungspartner eines einseitigen Ports hat zur Folge, dass eine Verbindung alle Kapazitäten eines Ports konsumiert, wenn der zweiseitige die größere Kapazität hat. Bei den anderen Portverbindungen, welche weitere Verbindungen ermöglichen, werden zusätzliche Verbindungsstellen instanziiert. Da sich bestehende und zusätzliche Verbindungen die Kapazität eines Ports teilen müssen, werden sich die möglichen Translationen dieser Verbindungsstellen gegenseitig einschränken.

Satz 5.2 (**Translation bei mehrfach verbundenen Extrusionsports**): Die erlaubte Translation bei Extrusion-Ports, welche mit mehreren Ports verbunden sind, lässt sich wie folgt berechnen:

Sei

p2 die zweite Verbindungsstelle von P,

P" der mit p2 verbundenen Port,

p"1 die entsprechende Verbindungsstelle

dann gilt für die Schranken C_{p2}^{Min}, C_{p2}^{Max} der Translationen T_{p2} und

die Schranken C_{p1}^{Min}, C_{p1}^{Max} der Translation T_{p1}:

$C_{p1}^{Min} = T_{p2}\text{-}C_{p'}$

$C_{p1}^{Max} = C_P$ (Standard)

$C_{p'1}^{Min} = 0$ (Standard)

$C_{p'1}^{Max} = C_{p'}$ (Standard)

$C_{p2}^{Min} = 0$ (Standard)

$C_{p2}^{Max} = Tp_1+Tp'_1\text{-}C_{p'}$

$C_{p"1}^{Min} = 0$ (Standard)

$C_{p"1}^{Max} = C_{p"}$ (Standard)

und die Einschränkungen für T_{p1} und T_{p2} gemäß den folgenden Tabellen:

	P einseitig und $C_{p1}^{Max} > C_{p'1}^{Max}$	P zweiseitig
P' zweiseitig	$Tp_1 \in [C_{p1}^{Min} \quad C_{p1}^{Max}]$ $Tp'_1 = C_{p'1}^{Min}$	$Tp_1 \in [C_{p1}^{Min} \quad C_{p1}^{Max}]$ $Tp'_1 \in [C_{p'1}^{Min} \quad C_{p'1}^{Max}]$

	$C_{p2}^{Max} \leq C_{p"1}^{Max}$	$C_{p2}^{Max} > C_{p"1}^{Max}$
P' einseitig	$Tp_1 \in [C_{p2}^{Min} \quad C_{p2}^{Max}]$ $Tp'_1 = C_{p"1}^{Min}$	$Tp_1 = C_{p2}^{Min}$ $Tp'_1 \in [C_{p"1}^{Min} \quad C_{p"1}^{Max}]$
P' zweiseitig	$Tp_1 \in [C_{p2}^{Min} \quad C_{p2}^{Max}]$ $Tp'_1 = C_{p"1}^{Min}$	$Tp_1 \in [C_{p2}^{Min} \quad C_{p2}^{Max}]$ $Tp'_1 = C_{p"1}^{Min}$

Die Tabellen aus Satz 5.2 entsprechen denen aus Satz 5.1, wobei sie dadurch vereinfacht werden, dass als Vorbedingung P' zweiseitig und seine Kapazität geringer als die von P sein muss, damit überhaupt weitere Verbindungen an P entstehen können, und P für p2 und alle weiteren Verbindungspunkte als einseitig angesehen werden muss, da die bereits bestehende Verbindung den Port nach unten abschließt. Die oberen und unteren Schranken der Ports werden überwiegend auf die gleichen Werte gesetzt wie bei der einfachen Verbindung. Ausnahme ist die untere Schranke der Translation des ersten Ports, die durch die Eindringtiefe der zweiten Verbindungsstelle begrenzt wird, und die obere Schranke des zweiten Ports, welche durch die Eindringtiefe des ersten Ports bestimmt wird. Da für den ersten verbundenen Port noch Platz zwischen den beiden Intervallen frei gehalten werden muss, wird bei den beiden Schranken zusätzlich noch die Kapazität des verbundenen Ports berücksichtigt.

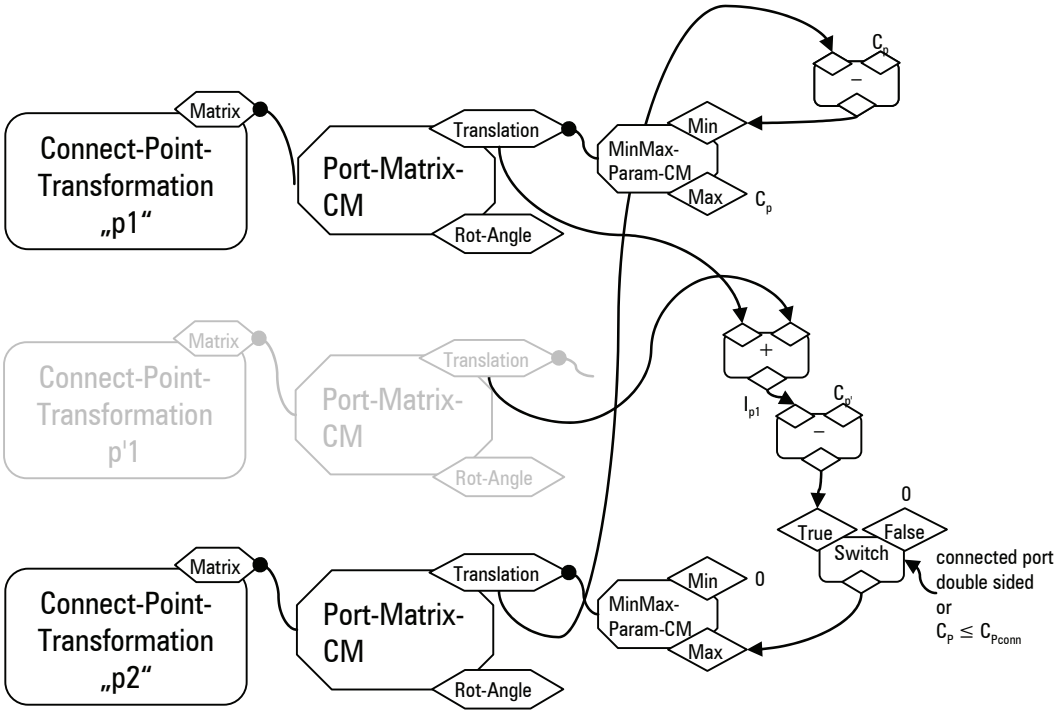


Abbildung 64: Aufbau der Constraint-Mediatoren für die Kapazitätsprüfung bei Mehrfachverbindungen

In Abbildung 64 ist ein CM-Netzwerk aufgezeigt, mit dem die Formeln aus Satz 5.2 implementiert werden. Zwei Port-Matrix-CM schränken die Matrix der Transformationsknoten der beiden Verbindungsstellen ein. Die jeweilige Translation der Verbindungsstelle entlang der Portachse wird durch Min-Max-Parameter-CM eingeschränkt, deren jeweilige Grenzen durch die Verschaltung von einfachen Programmknoten festgelegt werden. Die beiden Fälle in der Tabelle aus Satz 5.2 werden durch einen Switch-Knoten realisiert, der je nachdem, ob der verbundene Port zweiseitig ist, oder eine größere Kapazität hat, das passende Maximum der Translation auswählt.

Satz 5.3 (**Translation bei an beiden Seiten verbundenen Extrusionsports**): Die erlaubte Translation bei Extrusion-Ports, welche an beiden Seiten mit Ports verbunden sind, lässt sich wie folgt berechnen:

Sei P ein zweiseitiger Extrusion-Port,

p, $\overline{p}$ seine zwei gegenüberliegende Verbindungsstellen,

P', $\overline{P}$' potentielle Verbindingspartner

$C_P = C_{\overline{p}}$ die Gesamtkapazität von P,

$T_{\overline{p}}$, T_p die aktuelle Translation von der Verbindungsstelle $\overline{p}$ bzw. p,

$C^{Max}_{p1} = C_P - T_{\overline{p}1}$

$C^{Max}_{\overline{p}1} = C_P - T_{p1}$

	$\overline{p}1$ verbunden und $C^{Max}_{p1} \leq C^{Max}_{p'1}$	$\overline{p}1$ verbunden und $C^{Max}_{p1} > C^{Max}_{p'1}$	$\overline{p}1$ nicht verbunden
P' einseitig oder $\overline{p}'1$ verbunden	$T_{p1} \in [C^{Min}_{p1} \quad C^{Max}_{p1}]$ $T_{\overline{p}1} = C^{Min}_{\overline{p}1}$	$T_{p1} = C^{Min}_{p1}$ $T_{\overline{p}1} \in [C^{Min}_{\overline{p}1} \quad C^{Max}_{\overline{p}1}]$	$T_{p1} = C^{Min}_{p1}$ $T_{\overline{p}1} \in [C^{Min}_{\overline{p}1} \quad C^{Max}_{\overline{p}1}]$
P' zweiseitig oder $\overline{p}'1$ nicht verbunden	$T_{p1} \in [C^{Min}_{p1} \quad C^{Max}_{p1}]$ $T_{\overline{p}1} = C^{Min}_{\overline{p}1}$	$T_{p1} \in [C^{Min}_{p1} \quad C^{Max}_{p1}]$ $T_{\overline{p}1} = C^{Min}_{\overline{p}1}$	$T_{p1} \in [C^{Min}_{p1} \quad C^{Max}_{p1}]$ $T_{\overline{p}1} \in [C^{Min}_{\overline{p}1} \quad C^{Max}_{\overline{p}1}]$

Die Formeln in der Tabelle in Satz 5.3 entsprechen den Formeln in Satz 5.2, wobei in diesem Fall nicht zwischen ein- und zweiseitigen Ports unterschieden wird, da Ports mit gegenüberliegenden Verbindungsstellen zwangsläufig zweiseitig sind. Unterschieden werden die Ports danach, ob die gegenüberliegende Verbindungsstelle belegt ist, was den Port an der Seite abschließt. In dem Fall entspricht er einem einseitigen Port. Die obere Schranke der ersten Verbindungsstelle wird um die Eindringtiefe der gegenüberliegenden Verbindungsstelle verringert. Durch die gegenüberliegende Verbindungsstelle wird nur jeweils die erste Verbindungsstelle des Ports eingeschränkt, da nur diese beiden Verbindungsstellen in direkten Kontakt miteinander kommen können. Die weiteren Verbindungsstellen werden indirekt über diese erste Verbindungsstelle eingeschränkt.

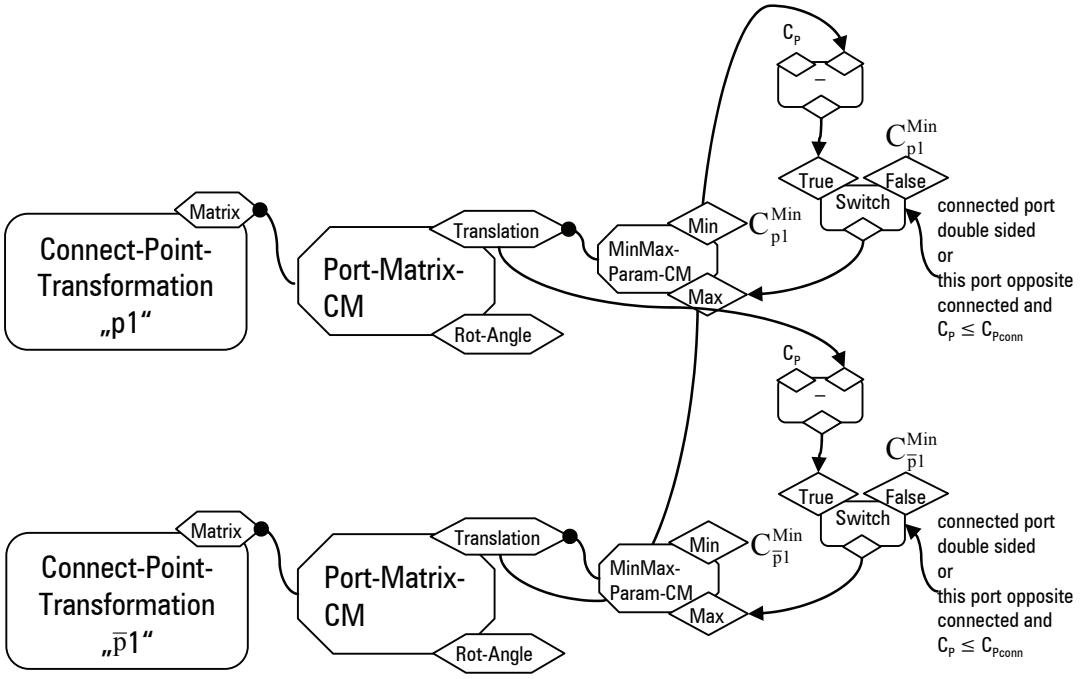


Abbildung 65: Aufbau der Constraint-Mediatoren für die Kapazitätsprüfung bei zweiseitigen Extrusion-Ports

Abbildung 65 zeigt eine Konfiguration von Constraint-Mediatoren, welche die Abhängigkeiten, die in Satz 5.3 angegeben sind, realisiert. Der Aufbau ist ähnlich wie bei dem CM-Netzwerk für die Mehrfachverbindungen. Die beiden CM-Netzwerke sind aber kombinierbar, da sie unterschiedliche Grenzen oder unterschiedliche Falle des Switch-Knotens einschränken.

6 Virtuelle Konstruktion

Dieses Kapitel gibt eine Übersicht über den Einsatz des visuellen Programmiersystems und der Constraint-Mediatoren im Bereich der Virtuellen Konstruktion, insbesondere im DFG Forschungsprojekt „Virtuelle Werkstatt“ (Biermann et al., 2002; Jung et al., 2002) im Rahmen dessen ein Großteil des Systems entstanden ist. Das Projekt „Virtuelle Werkstatt“ beschäftigt sich mit dem Ausbau von Forschungsarbeiten aus den Bereichen Multimodale Interaktion und Virtuelle Konstruktion.

Für die multimodale Interaktion in der Virtuellen Konstruktion wird eine möglichst natürliche Steuerung über Sprache und Gestik angestrebt. Dafür müssen die Sprache und die Gesten des Benutzers in der Virtuellen Umgebung erkannt, klassifiziert und zusammen zu einer Anweisung an das System integriert werden.

Zuvor definierte Bauteile sollen während der virtuellen Konstruktion zusammengebaut und exploriert werden. Hierbei soll durch wissensbasierte Methoden eine Variantenkonstruktion der Bauteile erstellt und in ihrer Funktion überprüft werden können. Die Varianten der Bauteile betreffen die parametrisierte Transformation, aber auch komplexe inhomogene Skalierungen von Teilstücken. Ein Ziel der Virtuellen Konstruktion ist, Aggregate mit Hilfe der Simulation von kinematischen Ketten in ihren Abhängigkeiten der Bewegung direkt beim Zusammenbau in der Virtuellen Umgebung testen zu können. Das ermöglicht einen Rapid-Prototyping-Prozess für den Entwurf und die Exploration von neuen mechanischen Objekten in einer immersiven Virtuellen Umgebung.

Um die Eigenschaften der Bauteile für den Konstruktionsprozess, aber auch für die Referenzierung über Sprache und Gestik, bereitstellen zu können, müssen zunächst die semantischen Informationen der virtuellen Objekte in geeigneten Datenstrukturen repräsentiert und für das VR-System zugreifbar gemacht werden.

6.1 Semantische Bauteil-Informationen

Die gesamte Struktur der Bauteile für die virtuelle Konstruktion ist in einer angebundenen Wissensstruktur, dem *Knowledge-Representation-Layer (KRL)* repräsentiert (Latoschik et al., 2005b). Diese Struktur enthält sämtliche Informationen der Bauteile, die für die Instanziierung und Interaktion in der Virtuellen Umgebung erforderlich sind. Neben den Geometrieinformationen und Materialeigenschaften der einzelnen Bestandteile sind in dieser semantischen Repräsentation auch die statischen und parametrisierten Transformationen und ihre eventuellen Abhängigkeiten untereinander enthalten. Der Zugriff auf diese Informationen für Objekte im Szenengraphen erfolgt über *Semantic-Entities* (*SE*). Diese Knoten werden im Szenengraphen unter den Wurzelknoten des Objektes gehängt, für das die zusätzlichen Informationen der SEs zur Verfügung stehen sollten. Die Verortung von semantischen Informationen in Form von Semantic-Entities hat mehrere Vorteile.

Ein Vorteil ist die einfache Suche von Objekten und ihrer zugeordneten Informationen direkt im Szenegraphen. Da die SE im Szenengraphen an derselben Stelle wie die zugehörigen Objekte selber platziert sind, können sie über eine einfache Traversierungsfunktion gefunden und ihr aktueller Ort bestimmt werden. Die Knoten der visuellen Programmierung zur Abfrage von Objektinformationen im Szenengraphen,

wie z.B. das Finden von Objekten in der Nähe des Zeigestrahls, benutzen bei der Suche die SE-Knoten.

Anhand der SEs der Objekte können dann einfach weitere Informationen abgefragt werden. Auch weitere Knoten im Szenengraph oder Programmknoten der visuellen Programmierung werden als Information über die Semantic-Entities gespeichert. Somit können z.B. die Constraint-Mediatoren, welche die geometrischen Constraints eines Bauteils oder seiner Verbindungen festlegen, direkt über die SEs gefunden und neue CM angemeldet werden.

6.2 Definition der virtuellen Bauteile

Um die Informationen der Bauteile möglicht elegant definieren zu können wurde für die Beschreibung der Bauteile für die Interaktion in der virtuellen Konstruktion ein eigenes XML-Format, die ***V**ariant **P**art **M**arkup **L**anguage (VPML)(Biermann & Jung, 2004)*, entwickelt. Dieses Format ist speziell für diese Art von parametrisierbaren Bauteilen ausgelegt und erlaubt dadurch eine sehr übersichtliche Definition der Bauteileigenschaften, was ihre Unterteilung und Parametrisierbarkeit betrifft. Eine besondere Stärke der VPML-Struktur liegt in der Möglichkeit, komfortabel die Abhängigkeiten von Skalierungen von Untergruppen in der Bauteilhierarchie und Kopplungen von parametrisierten Transformationen beschreiben zu können.

Die Informationen aus den VPML-Daten können entweder in die semantische Repräsentation eingelesen oder direkt in Szenengraphstrukturen übersetzt werden. Bei der Übersetzung der Dateien in eine Szenengraphstruktur werden für eventuelle parametrische Transformationen und Kopplungen die entsprechenden Constraint-Mediatoren mit aufgebaut. Auch die Definition von beweglichen Verbindungsstellen in einer VPML-Datei wird im Szenengraph über Constraint-Mediatoren realisiert, welche die beschriebenen eingeschränkten Bewegungen überwachen. Damit wird für jedes Bauteil ein Szenengraph instanziiert, der durch die angelagerten Constraint-Mediatoren die erlaubten Skalierungen und Bewegungseinschränkungen der Teilstücke und Verbindungsstellen automatisch mit sich trägt.

Zusätzliche Semantic-Entity-Knoten werden an die Wurzelknoten der Bauteile, Teilstücke und Ports instanziiert. Diese Knoten erlauben einen direkten im Szenengraph verorteten Zugang zu den semantischen Informationen der Teile und somit eine direkte und komfortable Schnittstelle zwischen der Virtuellen Umgebung und der Ebene der Wissensrepräsentation.

Tabelle 8 zeigt eine Auswahl der in der VPML benutzten XML-Tags. Neben der Strukturierung der Bauteile durch Part-/SubPart-Beziehungen kann die Geometrie der Einzelteile, Informationen für die komplexe Skalierung und der unterschiedlichen parametrische Attribute der Bauteile angegeben werden. Zusätzlich definiert die VPML auch die Verbindungsstellen der Bauteile über Ports und weitere semantische Informationen, welche z.B. für die sprachliche Referenzierung der Objekte in der multimodalen Interaktion wichtig sind.

Tag	Sub Tags	Attributes
Part	SubPart, Geometry, ProxyGeometry, Parametric, Port, SemanticInformation	Name
SubPart	SubPart, Geometry, ProxyGeometry, Parametric, Port, ScaleModes, SemanticInformation	name, translation, rotation, scaling, scale_reference
Geometry	File, Cylinder, Box, Sphere, …	translation, rotation, scaling, csg
ScaleModes	ParentScaling	fixpoint
ParentScaling	-	axes, mode
Parametric	Rotation, Translation, Scaling	-
Rotation	-	name, min, max, init, axis, center, connect, transmission, offset
Translation	-	name, min, max, init, direction, connect, transmission, offset
Scaling	-	name, min, max, init, axes, connect, transmission, offset
Port	Capacity, PortGeometry, Hotspot, Connectability	name, type, translation, orientation
SemanticInformation	SuperClass, Color, Lingustic	-

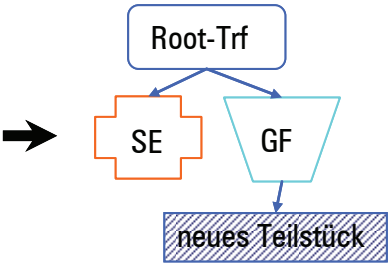
Tabelle 8: XML-Tags der VPML – aus (Biermann & Jung, 2004)

Der Aufbau des Szenengraphen für ein Bauteil erfolgt aus der Beschreibung in der VPML-Datei. Prinzipiell entsteht aus jedem „Part“ und „Subpart“-Tag ein Gruppierungsknoten, der die weiteren Unterteile unter sich gruppiert. Hinzu kommen Transformationsknoten, welche die festen relativen Positionen und Skalierungen der Teile angeben. Bei dynamisch skalierbaren bzw. parametrischen Teilstücken werden noch zusätzliche Transformationsknoten dazwischen gehängt, die veränderbare Transformationen implementieren und durch entsprechende Matrix-Constraint-Mediatoren (siehe Abschnitt 5.3.1) überwacht werden. Die „Geometry“-Tags definieren primitive, parametrisierte Formen oder können Geometrien aus externen Fileformaten einbinden.

Die Definition der Bauteile erlaubt auch einfache Operationen aus der *Constructive-Solid-Geometry* (*CSG*), um komplexe Bauteile aus einfacheren Teilen zusammenzusetzen. Damit lassen sich auch negative Teile, wie z.B. Löcher für die Verbindung, erstellen. Ein optionaler, lokaler Goldfeather-Knoten (siehe Abschnitt 6.4.1) übernimmt als Draw-Callback das interaktive Rendering falls negative CSG-Teile im Bauteil auftreten.

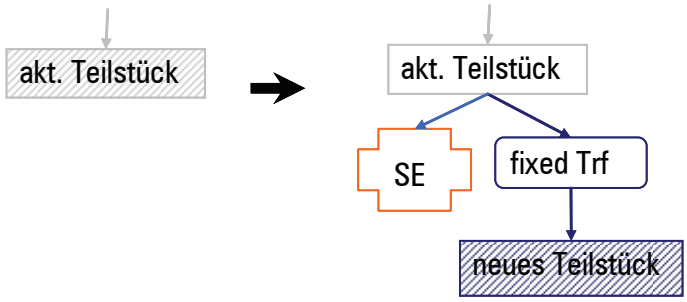
Die folgenden visuellen Grammatiken geben jeweils das Tag der VPML Beschreibung an und die entsprechende Operation, welche sie im Szenengraph ausführen. Die schraffiert dargestellten Objekte in der Grammatik geben jeweils die aktuell zu bearbeitenden Knoten an.

<Part>:



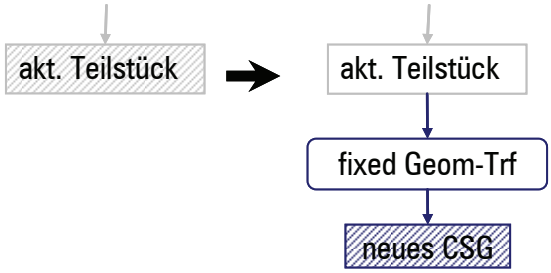
Dieses Tag instanziiert die Wurzel des Szenengraphen für das beschriebene Bauteil, bestehend aus der Haupttransformation des Bauteils, einem Semantic-Entity-Knoten (SE), in dem die direkt die Informationen aus dem XML-Attributen (wie z.B. Name, Typ, etc.) eingetragen werden, den Goldfeather-Knoten (GF) für das CSG-Rendering und einem Gruppenknoten für die Teilstücke. Dieser Knoten wird als aktuelles Teilstück gesetzt, auf den sich die nachfolgenden Graphtransformationen beziehen können.

<Subpart>:



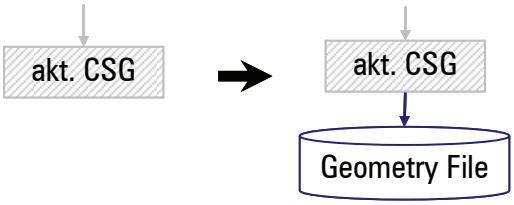
Das Subpart-Tag instanziiert unter dem aktuellen Teilstück-Knoten einen Semantic-Entity-Knoten, in dem die dieses Teilstück betreffende Informationen eingetragen werden und die relative Transformation des Teilstücks mit einem neuen Gruppenknoten. Dieser Gruppenknoten wird für die Graph-Transformationen innerhalb des aktuellen Subpart-Tags als aktuelles Teilstück gesetzt. Beim Schließen des Tags wird der aktuelle Knoten wieder auf den Ausgangsknoten zurückgesetzt.

<Geometry>:



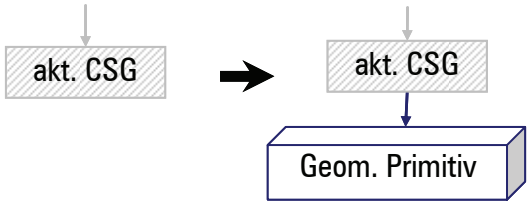
Mit Hilfe dieses Tags werden ein Transformationsknoten und ein CSG-Knoten unter den aktuellen Teilstück-Knoten gehängt. Der CSG-Knoten enthält die Information, ob es sich bei seinen Kindern um positive oder negative CSG-Primitive handelt und gilt innerhalb des Tags als aktueller Gruppenknoten für die folgenden beiden Transformationen „File“ oder „Primitive“.

<File>:



Das Geometry-Subtag instanziiert einen Knoten, der die Geometrie eines externen Fileformats enthält, unter dem aktuellen CSG-Knoten. Der Name des Geometriefile ergibt sich aus dem entsprechenden XML-Attribut. Die möglichen Fileformate ergeben sich aus den Importmöglichkeiten des eingesetzten Szenengraph-Tools.

<Primitive>:



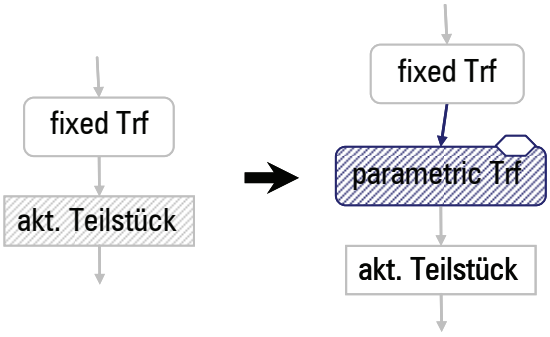
Verschiedene Primitive können direkt ohne externes File angegeben werden. Die resultierenden Geometrieknoten werden auch unter den aktuellen CSG-Knoten gehängt.

6.2.1 Gelenke und Getriebe

Veränderbare Parameter der Transformation von Bauteilen erlauben die Definition von beweglichen Teilstücken in einem Bauteil. In der XML-Definition wird dafür ein zusätzliches „Parametric“-Tag (s. Abb. X) eingeführt. Dieses resultiert in einer eingeschränkten, durch einen oder mehrere Parameter repräsentierten Transformation. Parametrische Translationen und Rotationen können so verschiedene Arten von Dreh- und Schiebegelenken zwischen zwei Teilstücken simulieren.

Die Simulation eines einfachen Drehgelenks mit einem rotatorischen Freiheitsgrad (entsprechend einem „Revolute-Pair“ – siehe Abschnitt 3.2.5) wird durch einen Transformationsknoten bewerkstelligt, dessen Matrix durch einen Matrix-Constraint-Mediator überwacht wird. Die Drehachse und die „min“- bzw. „max“-Attribute des „Parametric“-Tags werden benutzt, um den Mediator zu konfigurieren, damit die möglichen Extrempositionen des Teilstückes festgelegt werden können.

<Parametric>:

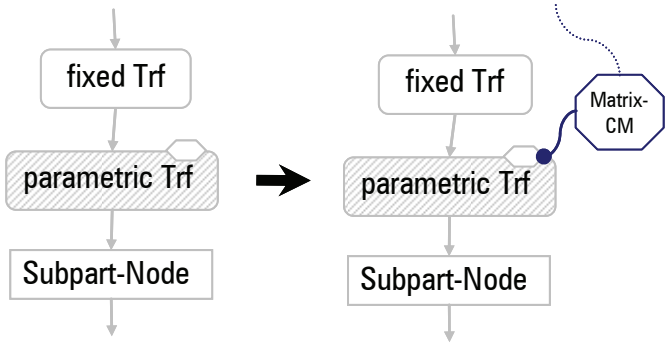


Das Parametric-Tag erzeugt für jeden Subtag (Rotation, Translation, Scaling) einen Transformationsknoten über dem aktuellen Teilstück-Knoten, dessen Matrix die parametrische Veränderung repräsentiert.

<Rotation>:

<Translation>:

<Scaling>:



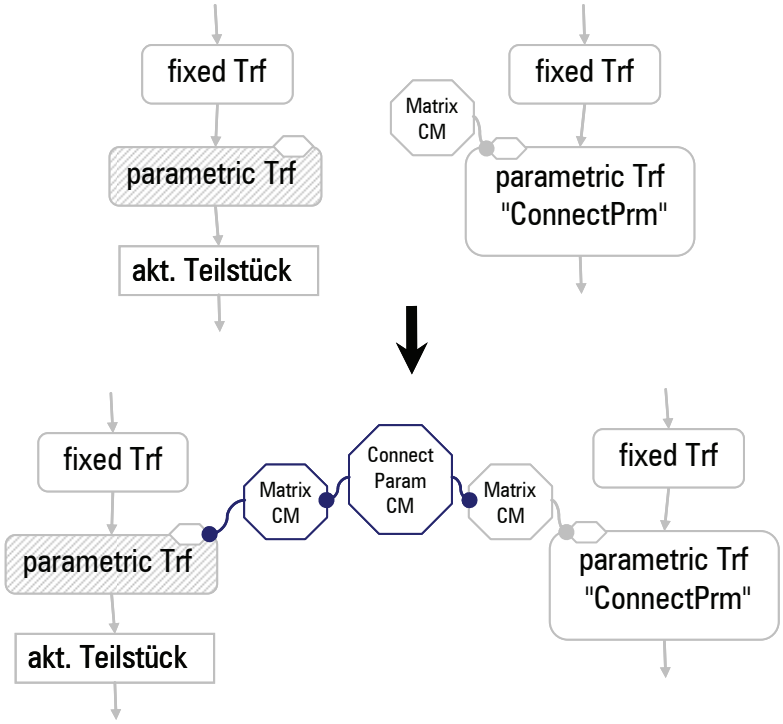
Diese Tags instanziieren und konfigurieren den Matrix-Constraint-Mediator des aktuellen Parametric-Transformationsknotens und realisieren dadurch entweder eine eingeschränkte Rotation um eine vorgegebene Rotationsachse, eine Translation entlang eines festen Richtungsvektors oder eine Skalierung in Richtung einer der Hauptachsen. Der Matrix-CM besitzt zusätzlich eine weitere Referenz auf die Matrix der nächst höheren, dynamischen Transformation, um die lokal nicht erfüllbaren Transformationsanteile an diese Matrix weiter zu propagieren (siehe Abschnitt 5.3.1).

Während der Interaktion mit den Bauteilen können gezielt einzelne parametrische Veränderungen freigegeben oder fixiert werden, um die Konfiguration des Aggregates möglichst komfortabel anzupassen.

Die Kopplung von Parametern innerhalb eines Bauteils ermöglicht außerdem die Simulation von Getrieben. Hier werden mehrere parametrische Rotationen oder Translationen über Parameter-Constraint-Mediatoren aneinander gekoppelt, so dass die Veränderung des einen Parameters über die Mediatoren an den anderen Parameter weitergegeben wird. Zusammen mit der Propagierung durch die Verbindungsstellen eines Aggregates (siehe auch die Beispiele in Abschnitt 6.3) können kinematische Ketten simuliert werden.

<Rotation
connect=
"P_ConnectPrm"
/>:

<Translation
connect=
"P_ConnectPrm"
/>:



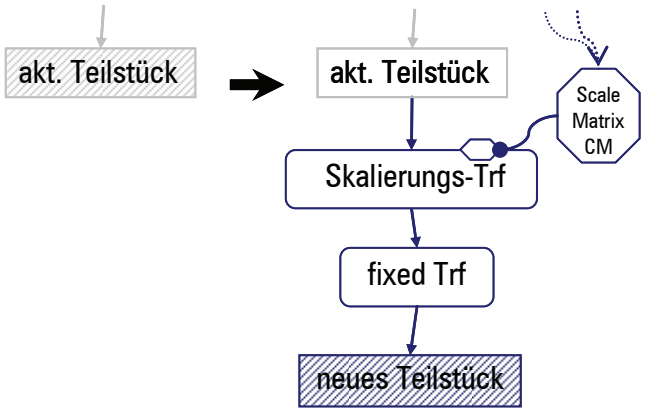
Neben den sog. Rigid-Body-Transformationen (Translation und Rotation), welche die Bewegungen von realen Bauteilen beschreiben, können die parametrischen Veränderungen auch auf die Skalierung der Bauteile Einfluss nehmen.

6.2.2 *Bauteil-Skalierung*

Bei der komplexen Skalierung von Bauteilen, bei der die verschiedenen Unterteile des Bauteils unterschiedlich skalieren können, überwachen die Skalierungs-Constraint-Mediatoren die Einhaltung der durch die XML-Definition gegebenen Skalierungsvorschriften. Die CM werden bei der Erstellung der Szenen-Graph Repräsentation des Bauteiles aus der XML-Beschreibung (VPML) während der Interpretation automatisch aufgebaut.

Die CMs zur Überwachung der Skalierungen binden direkt an die Matrizen der Transformationsknoten im Szenengraph. Sie werden dadurch zum einem eingeschränkt auf Skalierungen und zum anderem werden die Skalierungswerte der Unterteile aneinander gekoppelt.

<Subpart> +
<ParentScaling>:



Dieser Tag kann für die drei Hauptachsen festlegen, wie sich die eigene Skalierung zur Skalierung der Elterknoten verhalten soll. Es werden vier Modi betrachtet:

- **Parent:** Das Teilstück skaliert entsprechend der vom Elterknoten vorgegebenen Skalierungswerten. Dieses ist auch das Standartverhalten in einer hierarchischen Szenengraphstruktur.
- **Factor:** Das Teilstück skaliert in linearer Abhängigkeit mit einem vorgegebenen Faktor im Verhältnis zum Elterknoten.
- **None:** Die Skalierungen der Elterknoten werden nicht übernommen. Da durch die Semantik der Szenengraphstruktur alle Transformationen auf die Kinderknoten übertragen werden, muss die Skalierung der Elterknoten durch eine lokale Transformation aufgehoben werden.
- **Reference:** Das Teilstück richtet seine Skalierung nicht nach dem direkten Elterknoten, sondern nach einem zuvor als *Referenz* markierten Teilstück aus, der in der Szenengraphhierarchie sich über dem aktuellen Teilstück befindet. Hierbei wird immer auf das Referenzsteil Bezug genommen, das dem aktuellen Teilstück in der Hierarchie am nächsten ist.

Alle Modi – außer dem ersten – erfordern Informationen über die Skalierungswerte der übergeordneten Teilstücke, einerseits die lokalen Skalierungswerte des direkten Elter-Teilstück und andererseits die kumulierten Skalierungswerte zwischen dem aktuellen Teilstück und dem Referenzteil. Diese werden über Feldverbindungen dem aktuellen Scale-Matrix-CM zur Verfügung gestellt. Dieses ist in den Graphgrammatiken durch die gestrichelten Pfeile dargestellt.

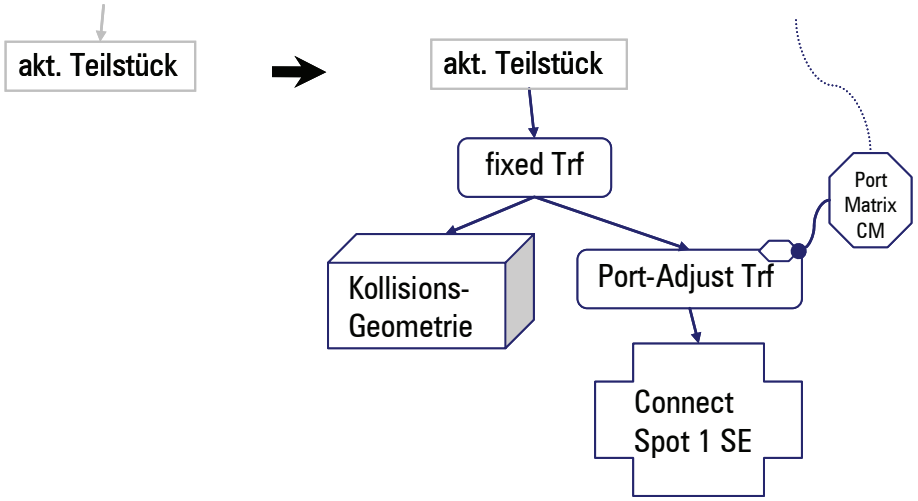
Da die hier betrachteten Skalierungen auf die Hauptachsen beschränkt sind, können Skalierungen von Teilstücken, die gegenüber dem aktuellen Stück gedreht sind, nicht über mehrere Ebenen von Subparts propagiert werden, weil sich die Achsen in dem Fall beliebig verändern können. Dieses wird in der Bauteildefinition dadurch modelliert, dass ein Teilstück, welches eine lokale Drehung enthält, automatisch zu einem Referenzteil wird. Der Wurzelknoten eines Bauteils gilt auch stets als Referenz.

6.2.3 Verbindungs-Ports

Um die Verbindungen bei der Interaktion korrekt aufbauen zu können, werden Informationen über die Verbindungsstellen im Semantischen Netz gespeichert

(Latoschik et al., 2005a). Bei der Anforderung einer neuen Verbindung zweier Bauteile werden anhand dieser Informationen die besten Ports der beiden zu verbindenden Bauteile ausgewählt und eventuell deren Initialkonfiguration festgelegt. Die Informationen umfassen z.B. verschiedenen Verbindungstypen und freie oder bereits belegte Kapazitäten. Auch der Constraint Mediator, der die neu entstandene Verbindung und seine Bewegungsconstraints überwacht, wird anhand dieser Informationen konfiguriert.

<Port>:



Mit diesem Tag wird unter dem aktuellen Knoten das Semantic-Entity für die Portinformationen erzeugt, das durch einen festen Transformationsknoten und einen Knoten für die Modellierung der Port-Freiheitsgrade transformiert wird. Der Port-Matrix-CM, überwacht die Transformation der ersten Verbindungsstelle des Ports entsprechend der für diesen Port gegebenen Freiheitsgrade. Die genaue Verschaltung des Port-Matrix-CM auch bei mehreren Verbindungsstellen ist in Abschnitt 5.4 beschrieben. Der Port-Matrix-CM kann wie alle Matrix-CM durch eine weitere Referenz auf eine Matrix die durch den modellierten Constraint nicht erfüllbare Resttransformationen an höhere Szenengraphebenen propagieren.

Zusätzlich wird eine dem Port entsprechende Kollisionsgeometrie instanziiert, welche später während der Interaktion hilft, Portverbindungen mittels Kollisionserkennung beim Zusammenführen von Bauteilen zu etablieren.

6.3 Beispiele von Bauteilen

6.3.1 Eine skalierbare Dreiloch-Leiste

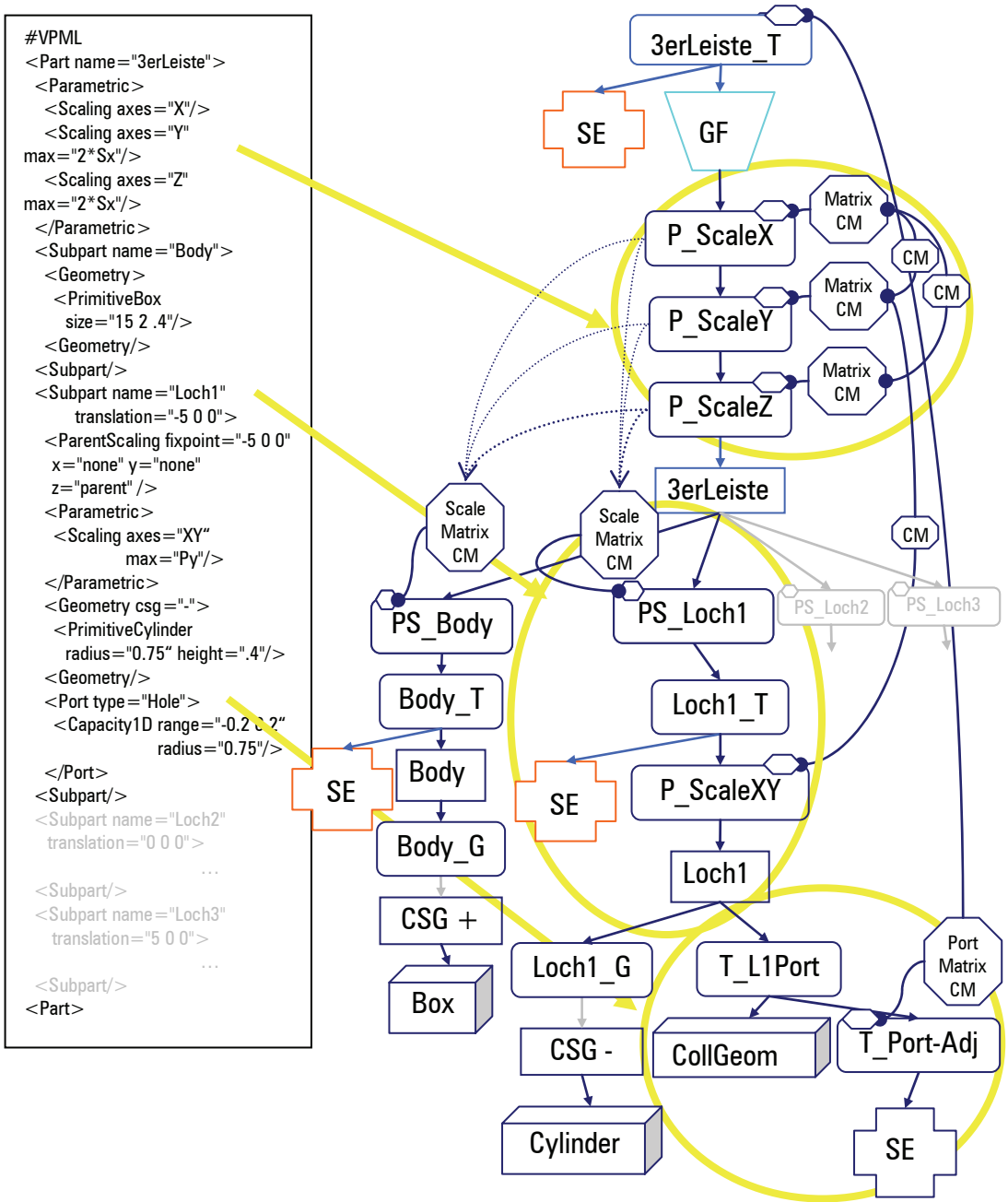


Abbildung 66: VPML-Datei und daraus resultierender Szenengraph einer Dreilochleiste

Das Beispiel aus Abbildung 66 beschreibt eine längliche Leiste mit drei Löchern. Auf der linken Seite der Abbildung ist das VPML-File abgebildet, das den Szenengraphen auf der rechten Seite definiert. In beiden Teilen der Abbildung ist der Übersicht wegen nur das erste Loch komplett ausgeführt.

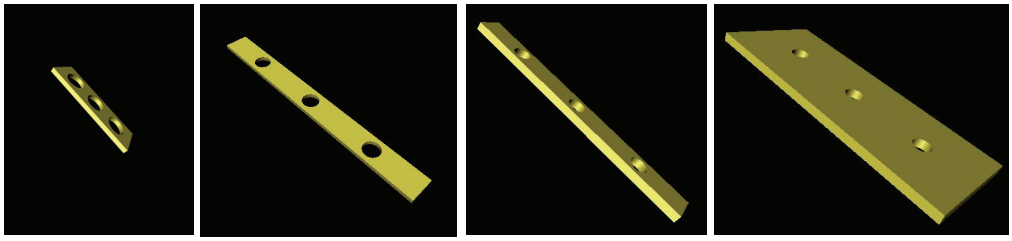


Abbildung 67: Skalierungen der Leiste in Länge und Breite – aus (Biermann & Jung, 2004)

Die Leiste ist in allen drei Achsen skalierbar, wobei der maximale Skalierungsfaktor der Y-Achse und der Z-Achse – was der Breite bzw. Höhe der Leiste entspricht – auf das Doppelte der Länge der Leiste (X-Achse) beschränkt ist. Damit wird sichergestellt, dass die X-Achse die dominierende Achse bildet und somit die längliche Grundform der Leiste erhalten bleibt.

Die Leiste besteht aus vier Teilstücken: Dem rechteckigen Leistenkörper und den drei einzeln skalierbaren, zylindrischen Löchern. Abbildung 67 zeigt verschiedene Konfigurationen der Leiste. Während der Körper die globalen Skalierungswerte der Leiste direkt übernimmt, sind für die Löcher spezielle Abhängigkeiten der Skalierung definiert. Die Höhe der Zylinder (Z-Achse) übernimmt den globalen Skalierungswert, da die Länge des Zylinders, der das Loch aus dem Leistenkörper ausschneidet, immer der Dicke des Körpers entsprechen muss. Die Skalierungswerte der beiden anderen Achsen sind hingegen von der Skalierung des Elternteils unabhängig modelliert. Zum einem soll damit erreicht werden, dass eine inhomogene Skalierung der Leiste die Löcher nicht elliptisch verformt, und zum anderen kann damit für die Löcher jeweils eine eigene, unabhängige Skalierung definiert werden. Diese Skalierung kombiniert die Skalierungswerte der X-Achse und die Y-Achse des Zylinders, um die Form der Löcher zu erhalten. Außerdem sind die Werte dieser Skalierung durch den Skalierungswert der Breite der Leiste begrenzt. Dieses verhindert, dass die Löcher über die Breite der Leiste hinausgehen und diese in mehrere Teile zerlegt (siehe Abbildung 68).

Abbildung 68: Fehlerhafter Skalierungswert eines Loches in einer Dreilochleiste

Jedes Loch der Leiste definiert einen Steckport. Der Port-Matrix-Constraint-Mediator definiert die Freiheitsgrade der Porttransformation entsprechend der in Abschnitt 5.3.3 beschriebenen Modellierung. Da keine parametrischen Translationen oder Rotationen für die Leiste definiert sind, wird die Transformation eines verbundenen Bauteils über die Referenz für die Resttransformation direkt an die Transformation der gesamten Leiste gebunden. Dadurch kann ein mit einem Loch der Leiste verbundenes Bauteil seine Transformation über diese Portverbindung an das Bauteil propagieren, wobei

immer erst die lokalen Freiheitsgrade der Verbindung ausgenutzt werden und dann erst die globale Transformation verändert wird.

6.3.2 *Ein skalierbarer und parametrisch veränderbarer Citymobil-Sitz*

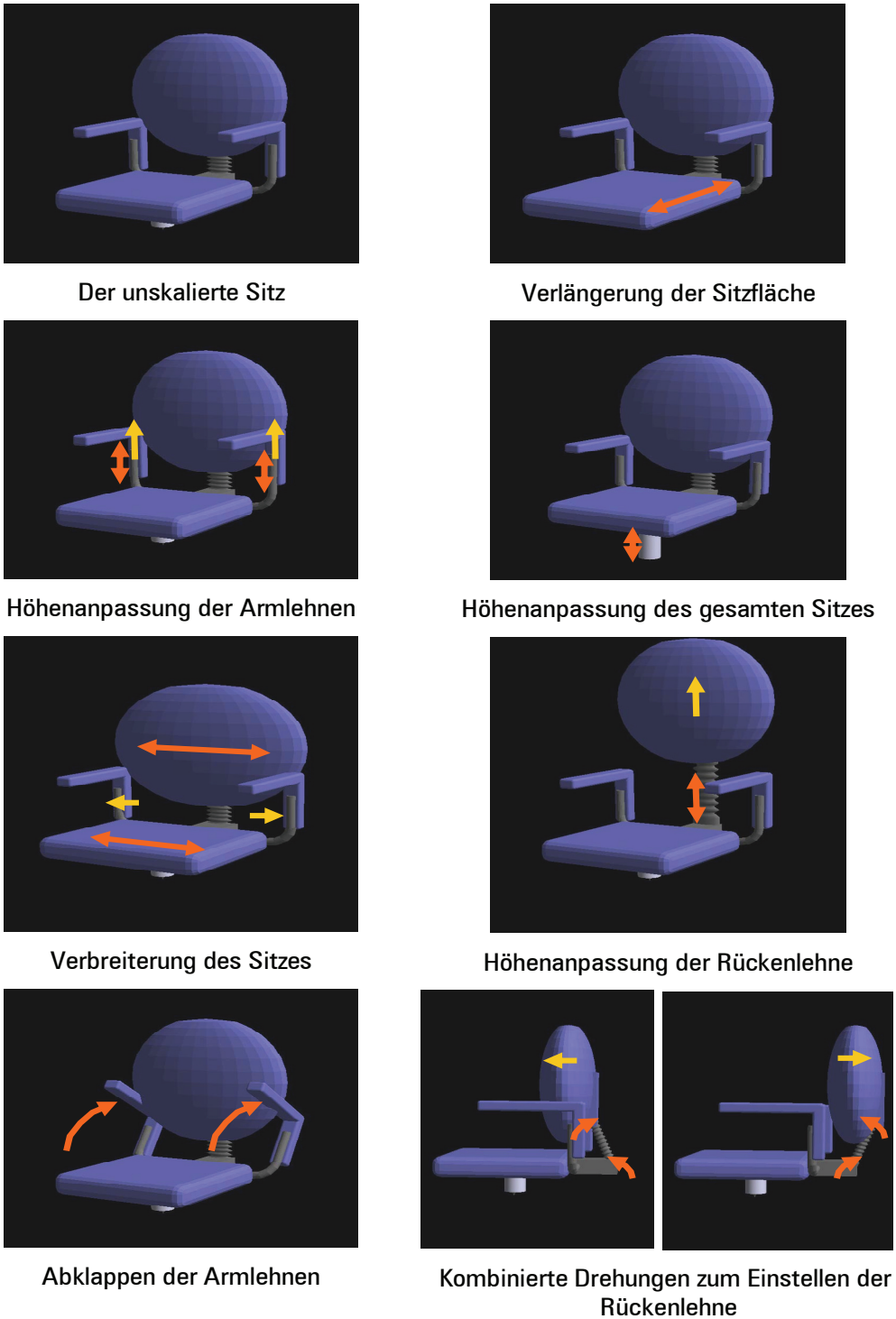


Abbildung 69: Verschiedene Konfigurationen des Citymobilsitzes – aus (Biermann & Jung, 2004)

Das Beispiel des skalierbaren Sitzes des in der „Virtuellen Werkstatt“ als Testszenario benutzten Citymobiles zeigt unterschiedliche Möglichkeiten der Skalierung und anderer Parametrisierung der Relativpositionen eines Bauteils mit komplexerer Geometrie.

Abbildung 69 zeigt eine Übersicht von verschiedenen parametrischen Anpassungen, welche für diesen Sitz modelliert sind.

Es handelt es sich dabei zum Teil um Konfigurationsanpassungen, wie sie auch bei einem realen Sitz direkt vorgenommen werden können. Zu diesen gehören vor allem die möglichen parametrischen Rotationen, z.B. der Armlehnen oder auch der Rückenlehne. Auch ein Teil der Skalierungen, wie die Höhenanpassung des gesamten Sitzes und die Höhe der Rückenlehne können bei einem realen Sitz direkt vorgenommen werden. Viele Skalierungen hingegen beschreiben Anpassungen, welche nur bei der Fabrikation des Sitzes vorgenommen werden können. Hierzu gehören z.B. die Verbreiterung des Sitzes und die Verlängerung der Sitzfläche. Diese Anpassungen können bei dem virtuellen Prototyp – innerhalb der Werte wie sie z.B. in der Produktion später möglich sind – direkt beliebig verändert werden. Dadurch ergeben sich Hinweise auf die möglichen Parameter bei der realen Konstruktion, wenn z.B. eine verlängerte Armlehne der Lenkstange oder anderen Teilen des gesamten Fahrzeuges im Weg ist.

Abbildung 70 zeigt einen Ausschnitt aus der Definition der Rückenlehne des Sitzes. In diesem Ausschnitt werden die Kopplung der Rotation der Haltestange der Rückenlehne und der Rotation der Rückenlehne selber und die Abhängigkeit der Skalierungen der beiden Teile beschrieben.

```
<Part name="citymobile_seat">
  ...
  <SubPart name="backrest_holder">
    ...
    <Parametric>
      <Scaling name="adjust_height"
        min="0.8" max="2.0" axis="Y"/>
      <Rotation name="backrest_hldr_rotate"
        min="-20" max="20" center="0 0 -0.3"/>
    </Parametric>
    <SubPart name="backrest">
      ...
      <ScaleModes fixpoint="0 0.4 -0.3">
        <ParentScaling axis="Y" mode="none"/>
        ...
      <ScaleModes/>
      <Parametric>
        <Rotation name="backrest_rotate"
          center="0 0.4 -0.3"
          combine="backrest_hldr_rotate"
          transmission="-1"/>
      </Parametric>
    </SubPart>
  </SubPart>
  ...
</Part>
```

Abbildung 70: Ausschnitt aus der VPML-Definition des Citymobilsitzes

Die Kopplung der Rotation ergibt sich durch das ‚combine’-Attribut bei der zweiten Rotation mit den Namen ‚backrest_rotate’. Dadurch werden die beiden Rotationen voneinander abhängig. Weil der Wert des Attributs ‚transmission’ auf ‚-1’ gesetzt ist,

ergibt sich eine gegenläufige Rotation der beiden Teile, wie sie durch die Pfeile in Abbildung 69 in dem Bild unten rechts angedeutet ist. Damit die Rückenlehne den Transformationen seiner Stange folgt ist er als ‚SubPart' der Haltestange modelliert. In einer normalen Szenengraphhierarchie würde allerdings auch die Skalierung der Haltestange auf die Lehne übertragen und sich wie diese bei einer Skalierung in die Länge ziehen. Um dieses unerwünschte Verhalten zu unterbinden, wird bei der Rückenlehne für die betroffene Achse der Modus des ‚ParentScaling' auf ‚none' gesetzt. Die entsprechenden Skalierungs-Constraint-Mediatoren beim Aufbau des Szenengraphen für dieses Bauteil sorgen dann dafür, dass eine eventuelle Skalierung des Vaterknotens durch eine lokale Skalierung aufgehoben wird. Das Attribut ‚fixpoint' beeinflusst das Zentrum dieser lokalen Skalierung und gibt somit die Position des Bauteils an, bei der im Falle einer Skalierung die relative Position der Bauteile gleich bleibt. Im vorliegenden Fall ist diese Position am oberen Ende der Haltestange, da hier die beiden Teile miteinander verbunden sind.

6.3.3 Ein Zahnstangengetriebe

Ein Zahnstangengetriebe realisiert in der Praxis die Kopplung zwischen einer Rotation und einer Translation, indem ein drehbares Zahnrad in eine Zahnstange greift und diese entsprechend der Drehung verschieben kann. Das virtuelle Getriebe in diesem Beispiel besteht drei Teilen: Dem rechteckigen Körper, einer Stange mit zwei Point-Ports jeweils am Ende und einem Loch dem auch ein Point-Port in der Mitte zugeordnet ist. Alle Ports des Bauteils sind als Point-Ports modelliert, obwohl ihre zugeordneten Geometrien klassische Extrusionsgeometrien sind. Diese Modellierung ergibt sich dadurch, dass die Verbindungen, die an diesen Ports eingegangen werden, keine Relativbewegungen erlauben sollen. Abbildung 71 zeigt das Getriebe in zwei Stellungen, wobei die Pfeile die Bewegungsmöglichkeiten der Teilstücke zeigen und die Verbindungsstellen durch Punkte an der jeweiligen Position der Geometrie dargestellt sind.

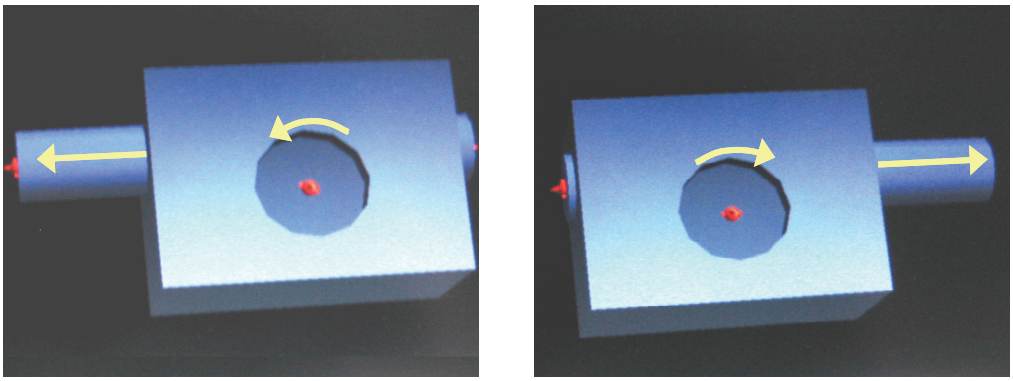


Abbildung 71: Rotation/Translation gekoppeltes Getriebe – aus (Biermann & Wachsmuth, 2004)

Eine korrekte geometrische Modellierung würde z.B. dem Verbindungsloch des Getriebes eine Form geben, die keine freie Rotation des Verbindungspartners zulässt und eventuell zusätzlich eine Fixierung zulässt, so dass die verbundene Stange nicht mehr herausgezogen werden kann. In der aktuellen Modellierung konkretisieren die definierten Verbindungsstellen die abstrakt gehaltene, geometrische Form, da hier vor allem die Funktionsweise des Verbindungskonzeptes gezeigt werden soll. Im Falle

einer realen Modellierung eines solchen Getriebes kann und sollte die Geometrie entsprechend komplexer ausfallen und die Verbindungsstellen korrekt wiedergeben.

Eine weitere Abstraktion wird durch die Simulation der Kopplung der Rotation des Loches mit der Translation der Stange eingeführt. Diese Kopplung beruht nicht auf einer physikalischen Simulation eines Zahnrads, welches im Inneren des Getriebekastens in eine Zahnstange greift, sondern auf einer abstrakten funktionalen Kopplung der beiden Transformationen über Constraint-Mediatoren, die um ein vielfaches einfacher zu berechnen und in ihrer funktionalen Abhängigkeit zu Konfigurieren ist. Der innere Aufbau der in diesem System eingesetzten Getriebe und Scharniere wird nicht betrachtet, sondern nur ihrer Funktionsweise nachgebildet.

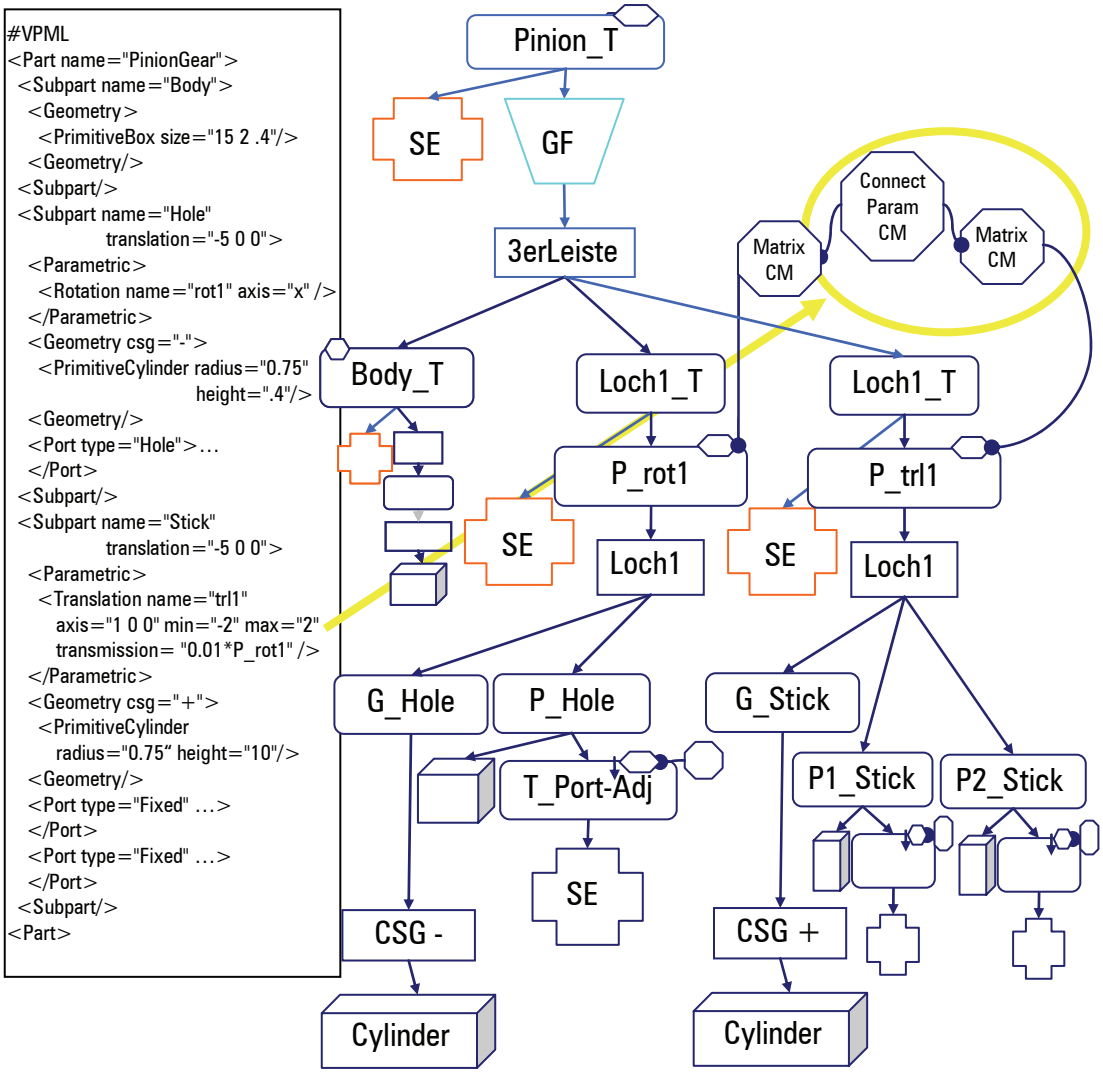


Abbildung 72: VPML-File und resultierender Szenengraph des Zahnstangengetriebes

Da die Ports in der Szenengraphhierarchie jeweils unterhalb der zugehörigen Geometrie modelliert sind, ergibt sich deren Position aus der Position dieser Geometrie. Das ermöglicht eine relativ einfache Modellierung des Bauteils. Abbildung 72 zeigt das VPML-File und den daraus aufgebauten Szenengraphen des Getriebes. Besonders hervorgehoben ist die Realisierung der funktionalen Kopplung der Rotation des Lochs mit der Translation der Stange durch die Constraint-Mediatoren. Der Connect-

Parameter-CM sorgt hierbei für die Propagierung des Rotationswertes bzw. Translationswerte. Die Propagierung der Werte kann in beide Richtungen erfolgen, je nachdem welcher Wert sich im letzten Simulationsschritt verändert hat. Sollten sich beide Werte verändert haben, was unter normalen Bedingungen eher selten vorkommt, da es in der Regel nur eine externe Transformation von Bauteilen gibt, die durch das Aggregat propagiert werden, muss der Constraint-Mediator zwischen diesen beiden Werten vermitteln. Wenn beide Transformationen mit der funktionalen Kopplung der CM kompatibel sind, können diese Werte unverändert bleiben, ansonsten muss ein Wert überschrieben oder ein mittlerer Wert der beiden genommen werden. Kompatible Werte der Transformationen können auftreten, wenn die Bauteile mehrfach mit Constraints belegt sind, die miteinander kompatibel sind, was z.B. bei Mehrfachverbindungen auftreten kann.

6.3.4 Eine Lenkmechanik mit Constraint-Propagierung im Aggregat

Die Lenkmechanik besteht aus 10 einzelnen Bauteilen: 4 Getrieben, 3 Verbindungsstangen, 2 Rädern und einer Lenkstange. Diese Bauteile sind über Portverbindungen miteinander verbunden. Da die Portverbindungen eine exakte Relativposition der Bauteile vorgeben sollen, wurden als Verbindungsstellen Point-Ports verwendet. Dieser Verbindungstyp hat keine eigenen Freiheitsgrade und sorgt für eine direkte Propagierung der Transformationen eines Bauteils an die verbundenen Teile. Alle Freiheitsgrade der möglichen Bewegungen und ihre Kopplung sind in den vier Getrieben realisiert. Alle anderen Bauteile des Aggregates tragen außer den festen Verbindungsstellen keine weiteren Informationen für die virtuelle Konstruktion. Für alle Getriebe im Aggregat soll nach dem Zusammenbau keine Transformation erlaubt sein. Normalerweise wird das durch die Fixierung der Getriebe über die Verbindung mit einer starren Geometrie (Karosserie) erreicht. Für die Übersichtlichkeit des Beispiels wurde diese nicht mit modelliert, sondern die Getriebe einfach direkt über Matrix-CM an ihren Wurzeltransformationen nach dem Zusammenbau des Aggregates auf die aktuelle Transformation eingeschränkt.

Das erste Getriebe simuliert eine einfache Kopplung der Rotationen seiner beiden Löcher an denen die Verbindungsports angebracht sind und bildet die Verbindung zwischen der Lenkstange und der ersten Verbindungsstange. Das nächste Getriebe in der Verbindungshierarchie ist das Zahnstangengetriebe aus dem vorherigen Beispiel. Es übersetzt die Rotation der ersten Verbindungsstange über seine Portverbindungen in die Translationen der zweiten und dritten Verbindungsstange. Die Verbindungen am anderen Ende der Stangen propagieren die Translation an das vierte bzw. fünfte Getriebe, welche die Translation in Rotationen für die verbunden Räder übersetzen. Abbildung 73 zeigt die Lenkmechanik in den beiden Extremstellungen, wobei die Räder ganz eingeschlagen sind. Die Position dieser Stellungen ergibt sich aus den Maximalwerten für die Transformationsparameter der einzelnen Getriebe. Hierbei bestimmt das Getriebe die Extremstellung, bei dem der Maximalwert als erstes erreicht wird.

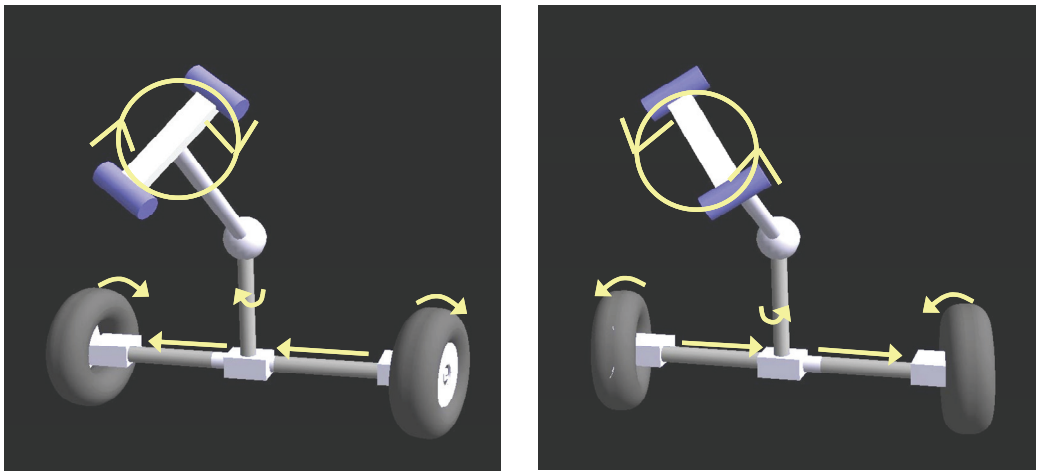


Abbildung 73: Lenkmechanik mit vier Getrieben – aus (Biermann & Wachsmuth, 2004)

Ein zusätzlicher Freiheitsgrad des ersten Gelenks betrifft eine weitere Rotation, mit deren Hilfe die Höhe des Lenkrades angepasst werden kann. Dieser Freiheitsgrad kann gesperrt werden, damit bei der Interaktion mit der Lenkstange nur die Lenkbewegung exploriert werden kann, ohne dass die Lenkung sich nach oben oder unten bewegt. Hierfür wird einfach ein Switch-Knoten erstellt, der den überwachenden MinMax-Parameter-CM für diese Rotation auf den Bereich der möglichen Rotation oder auf einen festen Wert festlegt. Für Testzwecke kann der Switch-Knoten über einen GUI-Sensor (siehe Abschnitt 4.5.1) direkt gesteuert werden.

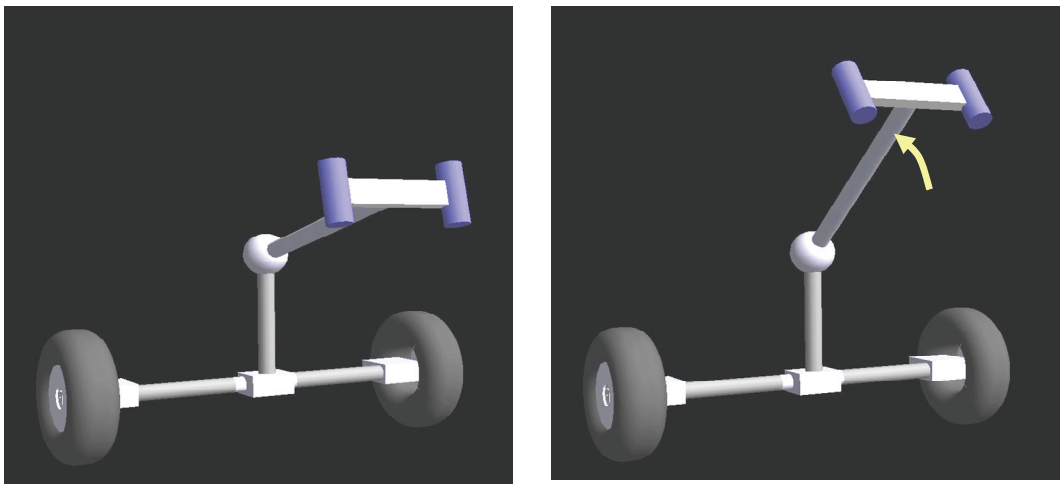


Abbildung 74: Höhenjustierung des Lenkrades – aus (Biermann & Wachsmuth, 2004)

In der späteren Interaktion wird der GUI-Sensor dann entfernt und der Parameter durch eine sprachliche Anweisung (wie z.B. „Fixiere die Höhe der Lenkstange“) gesetzt. Die Abbildung des Ausdrucks „Höhe der Lenkstange“ auf den Rotationsparameter des Getriebes muss dabei durch die Sprachverarbeitung mit Hilfe der semantischen Repräsentation geleistet werden, welche durch die Definition der einzelnen Bauteile und deren Verbindungsbeziehungen mit Informationen gefüllt wird. Das Fixieren und Lösen von Freiheitsgraden kann auch an anderen Stellen eingesetzt werden. So ist es z.B. sinnvoll, die relative Position von zwei verbundenen Planeports zu fixieren, sobald die gewünschte Stellung erreicht ist, damit eine Bewegung eines

verbundenen Teils diese Position nicht mehr verändern kann, sondern das Aggregat als Ganzes dieser Bewegung folgt.

6.3.5 Propagierung der Constraints in einem Aggregat

Anhand des Beispiels aus dem vorherigen Abschnitt lässt sich gut das Zusammenspiel der Constraint-Mediatoren für die Einschränkung und Propagierung der Bewegungen in einem komplexen, aus beweglichen Bauteilen zusammengesetzten Aggregat zeigen. Als Beispiel dient hier die Interaktion eines Benutzers, der zum Testen der Lenkbewegung die Lenkstange in der Virtuellen Umgebung bewegt. Anhand dieser Interaktion lassen sich im Prozess des „Virtual Prototypings“ schon viele Eigenschaften des späteren, realen Objektes testen. Zu einem können die Bewegungsmöglichkeiten der Lenkstange von einem Benutzer auf gute Erreichbarkeit und Handhabbarkeit getestet werden. Zum anderen können technische Einschränkungengetestet und schon im virtuellen Prototyp behoben werden. Beispiele hierfür sind maximale Bewegungsgrenzen bestimmter Getriebe oder nach dem Einbau des Lenksystems in das virtuelle Citymobil die Kollision der Räder mit dem Chassis bei bestimmten Lenkbewegungen. Letzteres kann im Szenario der „Virtuellen Werkstatt“ mit Hilfe der Einbindung des Kollisionserkennungssystems VCollide (T. Hudson *et al.*, 1997) schon erkannt und in der Virtuellen Umgebung angezeigt werden.

Das erste Getriebe der Lenkmechanik kann durch die Verbindung mit der Lenkstange nicht nur die Positionen der folgenden Teile verändern, sondern auch Einschränkungen in der Bewegung an die Lenkstange selber zurück propagieren. Diese interessante Eigenschaft bei der Popagierung von geometrischen Constraints wird im Folgenden anhand der Abfolge der Constraint-Mediator Aktivierung nachvollzogen. Abbildung 75 zeigt einen vereinfachten Aufbau des Szenengraphen des obersten Getriebes der Lenkmechanik mit der einfachen Rotationskopplung und die beiden damit verbundenen Bauteile, die Lenkstange und die Verbindungsstange. Für eine bessere Übersicht wurden alle nicht beteiligten Knoten des Szenengraphs weitgehend weggelassen. Zusätzlich ist eine aktive Kopplung der Lenkstange von der Hand eines Benutzers durch einen Interaktions-Constraint-Mediator (siehe auch Abschnitt 3.2.6) eingezeichnet. Aktive Komponenten oder sich verändernde Felder sind in den jeweiligen Abbildungen durch breite Linien hervorgehoben.

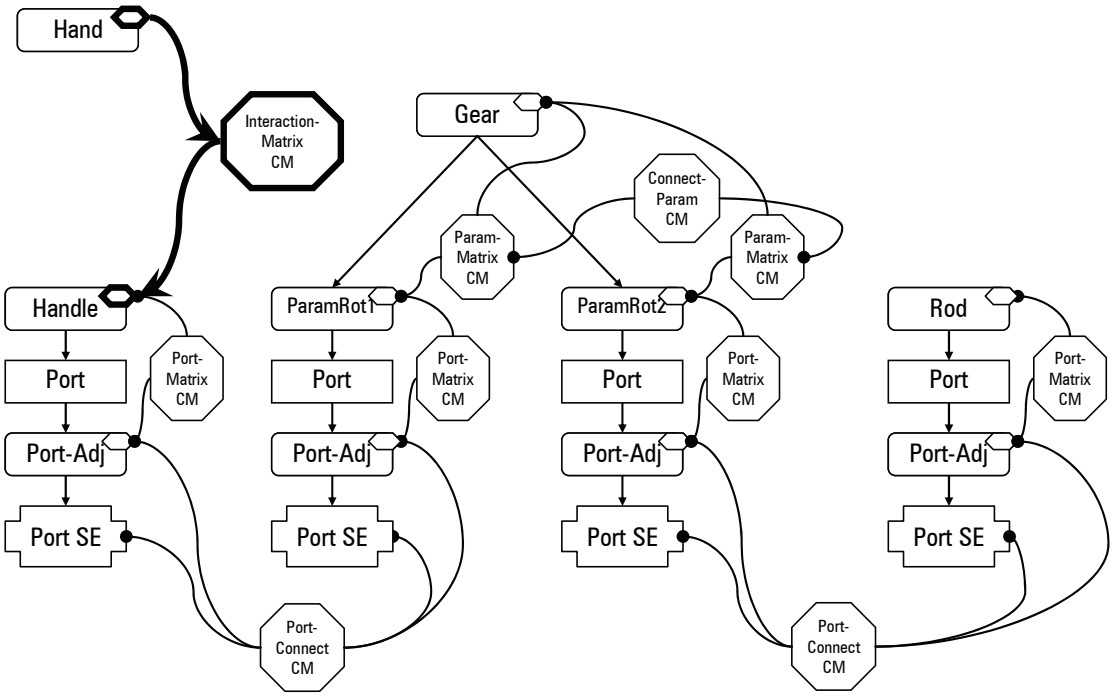


Abbildung 75: Vereinfachte Darstellung eines Getriebes mit zwei verbundenen Teilen

Die Bewegung der Hand, welche eine Drehbewegung des Lenkers herbeiführen soll, wird durch den Interaktions-Constraint-Mediator an die Matrix des Wurzelknotens der Lenkstange propagiert. Bevor jedoch die in der Regel ungenaue Bewegung der Lenkstange für den Benutzer dargestellt wird, werden weitere Constraint-Mediatoren aktiviert.

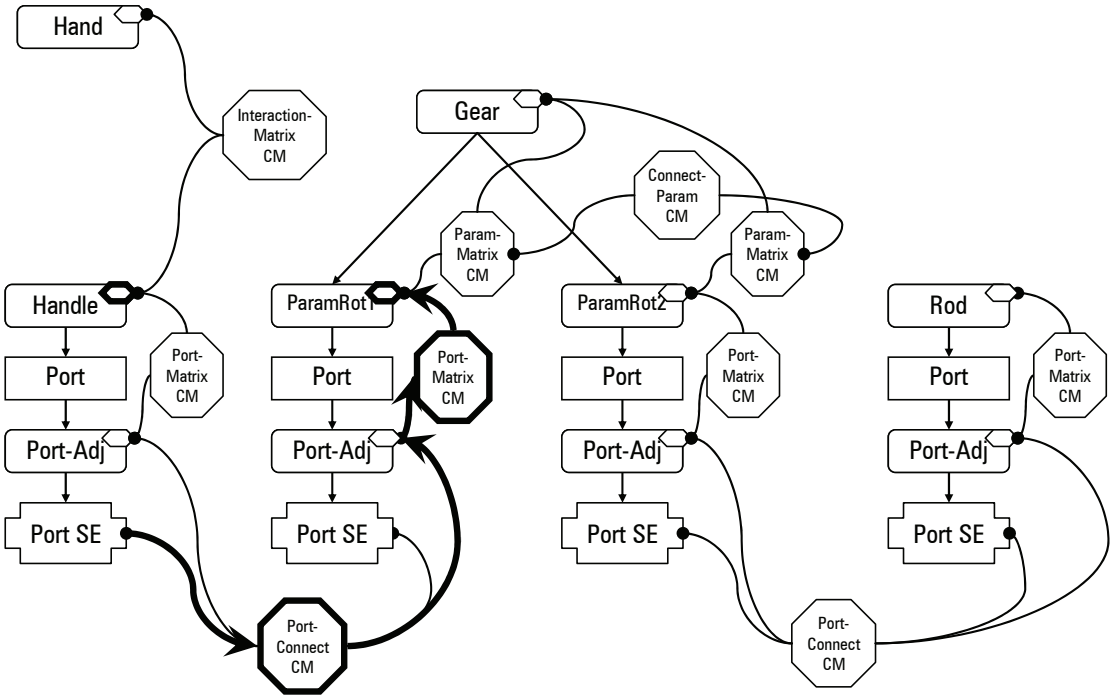


Abbildung 76: Propagierung an das Getriebe durch die Port-Verbindung

Wie in Abbildung 76 gezeigt, ist der erste Mediator, der durch die Veränderung im Aggregat aktiviert wird, ein CM der die Verbindung von Lenkstange und Getriebe überwacht. Dieser reagiert auf die globale Transformation des Semantic-Entities des Ports und, wie in Abschnitt 5.3.4 gezeigt, propagiert er die gesamte Transformation der Stange in die lokale Matrix des verbundenen Ports. Da dieser Port – wie die anderen beteiligten Ports im Aggregat auch – fest ist und daher keine Relativbewegung in der Verbindung erlaubt, leitet der zugehörige Port-Matrix-CM (genauer beschrieben in Abschnitt 5.3.3) die Transformation komplett über seine Adjust-Matrix an die nächst-höhere Instanz im Bauteil weiter. Dadurch bleibt die Port-Adjust-Matrix des verbundenen Ports im Endeffekt unverändert. In diesem Fall wird die Transformation an die Matrix der parametrischen Rotation für den Teil des Getriebes weitergegeben, der für den Abschnitt mit dem mit der Lenkstange verbundenen Port zuständig ist.

Die Kopplung der beiden rotatorischen Parameter im Getriebe sorgt mittels eines Connect-Parameter-CM dafür, dass der Rotationsanteil der Bewegung entsprechend der Kopplungsfunktion an die zweite parametrische Rotation weitergegeben wird (siehe Abbildung 77, Mitte). Die weitere Propagierung des nicht mit der Rotation vereinbaren Teils der Transformation an die nächst höhere Instanz – den Wurzelknoten des Getriebes – wird unterbunden, was in der Abbildung durch die gestrichelte Linie angedeutet ist. Die Folge wäre, dass man das gesamte Aggregat am Lenker durch den Raum ziehen könnte, was auch sinnvoll sein kann, in diesem Fall für die Erprobung der Lenkmechanik aber nicht gewünscht ist.

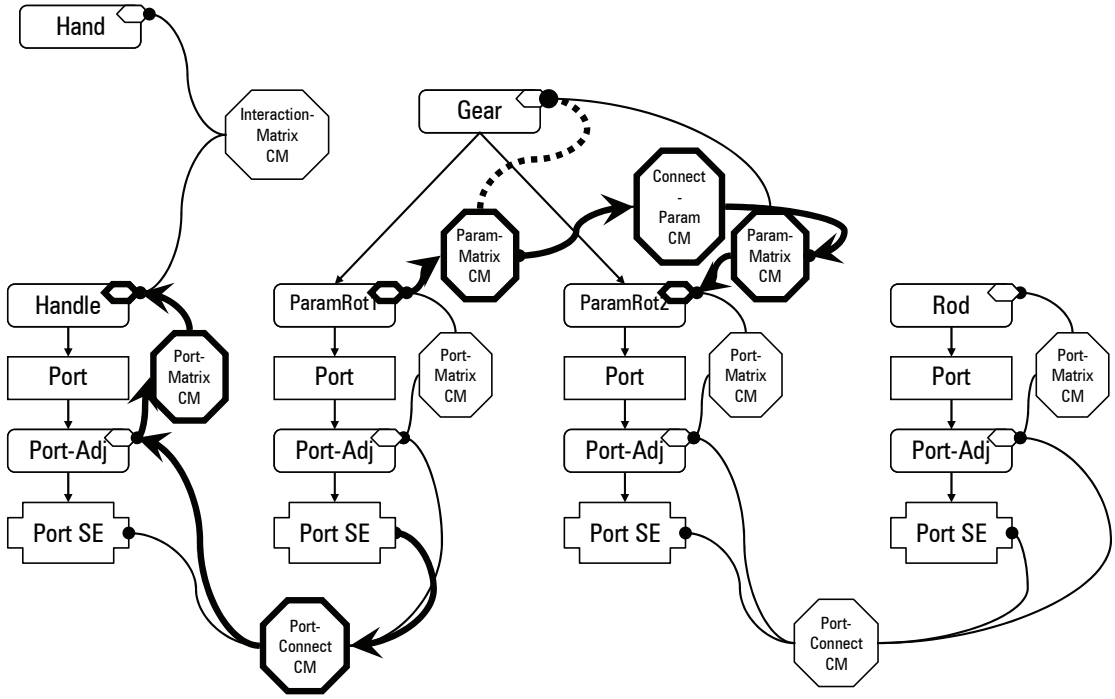


Abbildung 77: Propagierung an den gekoppelten Parameter und zurück an das Bauteil

Gleichzeitig reagiert der CM für die Verbindung zwischen Lenkstange und Getriebe auf die neue globale Transformation des Getriebeports und propagiert den nicht mit der Rotation kompatiblen Anteil der ursprünglichen Transformation, wie in Abbildung 77 auf der linken Seite zu sehen ist, zurück an die Lenkstange. Diese folgt also nicht komplett der Bewegung der Hand des Benutzers, sondern lässt sich nur – entsprechend der Festlegung durch das Getriebe – um die eigene Achse drehen.

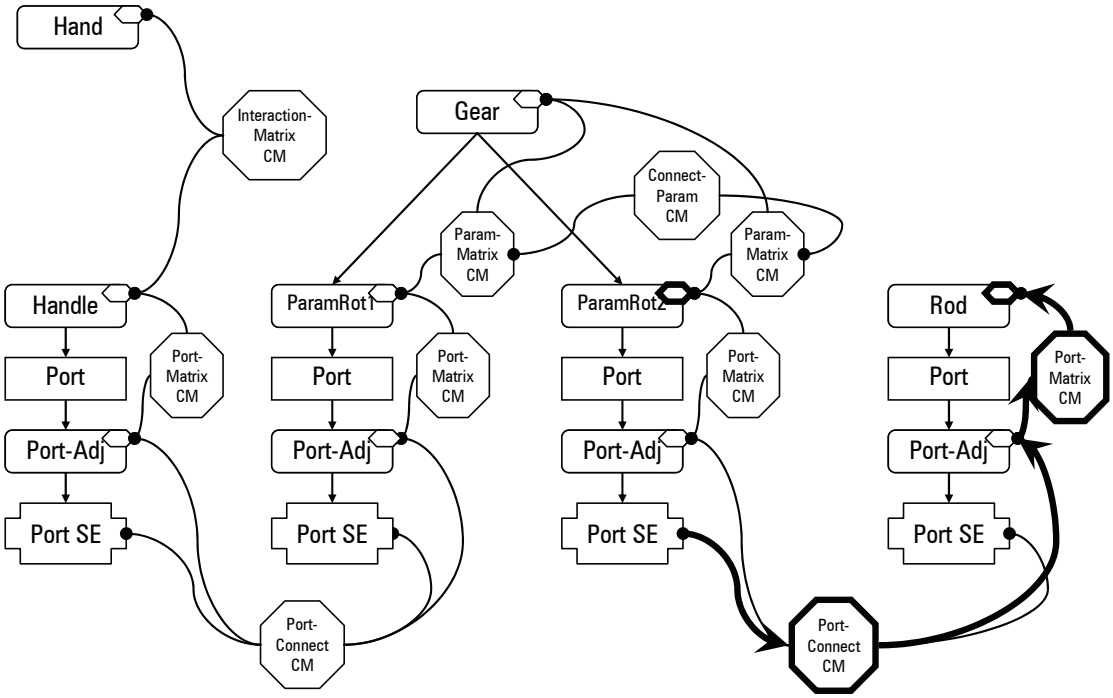


Abbildung 78: Propagierung an das zweite verbundene Bauteil

Letztendlich propagiert der CM, der die Verbindung von Getriebe und der unteren Verbindungsstange überwacht, die Rotation der zweiten parametrischen Rotation des Getriebes an diese Stange. Abbildung 78 zeigt, dass auch hier die Transformation zunächst an die lokale Port-Matrix der Verbindungsstange und von dort über den Port-Matrix-CM weiter an die Matrix des Wurzelknotens propagiert wird. Durch die Veränderung der Transformation der Verbindungsstange wird die Bewegung weiter an das nächste Getriebe und dann entsprechend an die restlichen Teile des Aggregats propagiert, was in der Abbildung aber nicht mehr ausgeführt ist.

6.4 CSG-Operationen

6.4.1 Bildbasiertes CSG-Rendering

Das VR-System zur virtuellen Konstruktion soll bei der Interaktion wie z.B. der interaktiven Veränderung von Skalierungswerten ein visuelles Feedback in Echtzeit erlauben. Da es nicht möglich ist, das Polygonmodell für jeden einzelnen Schritt neu zu berechnen, wurde ein bildbasiertes CSG-Rendering entwickelt, das für den Echtzeiteinsatz tauglich ist. Dieses Verfahren baut auf dem Ansatz des Goldfeather-Algorithmus (Goldfeather *et al.*, 1989) und seiner Übertragung auf heutige Graphikhardware auf. Anstatt der kompletten Berechnung eines Polygonmodells aus den CSG-Daten werden mit Hilfe des OpenGL Stencil-Buffers (T. Davis *et al.*, 1999) Teile der Geometrie so maskiert, dass der visuelle Eindruck einer korrekten CSG-Operation entsteht.

Der Algorithmus wurde an den eingeschränkten OpenGL-Befehlssatz der für die Virtuelle Umgebung eingesetzten Verteilung von Renderinstruktionen mit WireGL bzw. Chromium (Humphreys *et al.*, 2002) angepasst. Vor allem die Einschränkung,

den Tiefenbuffer der einzelnen Renderclients nicht auslesen und restaurieren zu können erforderte eine Überarbeitung des ursprünglichen Goldfeather-Algorithmus (siehe dazu auch (Biermann & Wachsmuth, 2004)).

6.4.2 Berechnung des Polygonmodells

Während die bildbasierte CSG für den Einsatz während der Interaktion mit den Bauteilen die Möglichkeit eines schritthaltenden Einsatzes für eine Echtzeit-Visualisierung geben kann, ist es für die folgende Berechnungen notwendig ein Polygonmodell des skalierten Bauteiles zu haben. Dieses Polygonmodell ist unerlässlich z.B. für die Kollisionserkennung, welche direkt auf den polygonalen Daten arbeitet. Außerdem ist das Multipass-Rendering für mehrere CSG-basierte Bauteile gleichzeitig aufgrund von Performanceeinbußen und wegen möglicher Störungen bei dem Einsatz des in Abschnitt 6.4.1 beschriebenen Algorithmus zu vermeiden. Daher wird am Ende der Interaktion durch den Benutzer die Berechnung eines Polygonmodells angestoßen.

Die Berechnung des Polygonmodells erfolgt in einem separaten Prozess, der als eigenständiger Agent modelliert und über ein Multi-Agenten-System – siehe auch (Wachsmuth, 1998) – an die VR-Applikation angebunden ist. Durch diese Modellierung kann die aufwendige Berechnung parallelisiert und komplett auf einen anderen Rechner ausgelagert werden. Das Polygonmodell wird durch das Agentenprogramm mit Hilfe der ACIS-CAD-Bibliothek (Jonathan Corney & Lin., 2002) berechnet. Die Grundstruktur der Bauteile für den Agenten wird mit Hilfe der VPML-Definitionen aufgebaut. Die jeweilige erforderliche Konfiguration der Bauteile (d.h. Skalierungen und andere Form- bzw. Transformations-Parameter) wird dem Agenten von der VR-Applikation jeweils bei Abschluss einer Interaktion übermittelt.

Ist die Berechnung erfolgt, was bei komplexeren Bauteilen einige Sekunden dauern kann, wird das neue Polygonmodell der VR-Applikation zur Verfügung gestellt und die bildbasierte CSG Darstellung mit dem berechneten Polygonmodell ausgetauscht.

6.5 Multimodale Interaktion

6.5.1 Gestenerkennung

Ein typisches Datenflussprogramm das zur Erkennung von Gesten eingesetzt wird, abstrahiert die kontinuierlichen Eingabedaten (z.B. Handpositionen und Winkel der Fingergelenke) zu abstrakten Konzepten (z.B. gradlinige oder kreisförmige Trajektorienabschnitte) und dann weiter zu den Features, die erkannt werden sollen. Hierdurch können z.B. die Detektor-Netze – wie in (Latoschik, 2001a) vorgestellt – einfach implementiert werden.

Beispiel: Handformerkenner

Für die einfache Interaktion in der Virtuellen Umgebung, die hauptsächlich aus der Selektion von Objekten – z.B. durch eine Zeigegeste mit einer entsprechenden sprachlichen Äußerung – und der Positionierung durch Drag-and-Drop besteht, werden vor allem folgende drei Posturen benötigt: Greifen, Loslassen (gestreckte Hand) und Zeigen

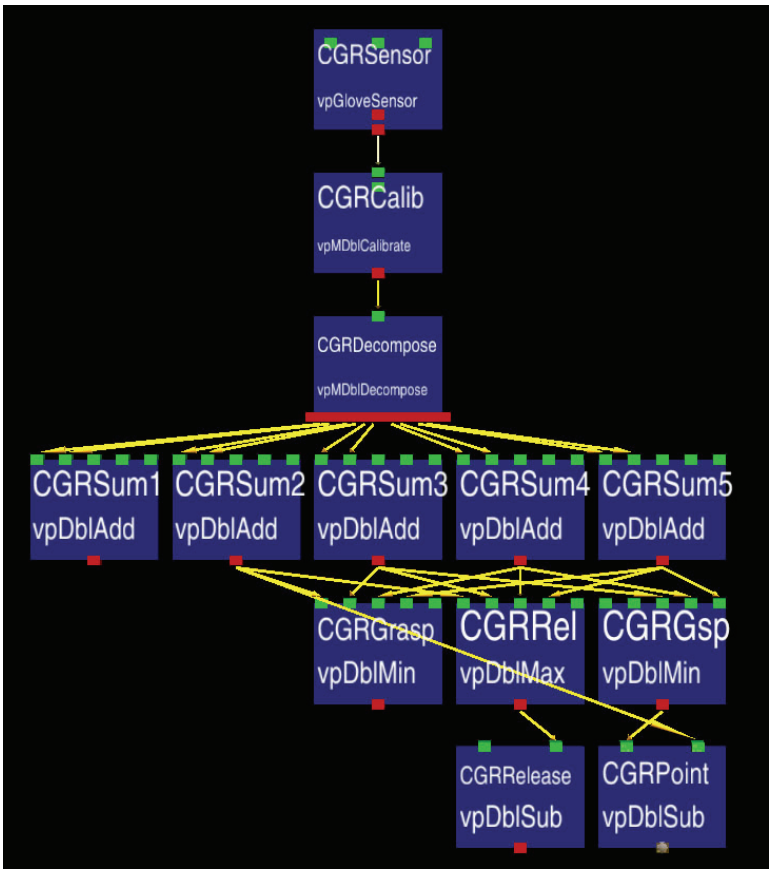


Abbildung 79: Visuelles Programm für die einfache Handposturerkennung

Das in Abbildung 79 dargestellte visuelle Programm realisiert eine sehr einfache aber robuste Erkennung dieser drei Gesten. Die Rohdaten aus dem Sensor für die Beugungswerte des Datenhandschuhs werden zunächst durch einen Kalibrierungsknoten auf einen einheitlichen Wertebereich gebracht. Um die einzelnen Beugungswerte der Gelenke einzeln weiterverarbeiten zu können wird der Datenstrom, der bis dahin als eine Liste von Werten über Multi-Felder weitergegeben wird, durch einen „Decompose"-Knoten in seine einzelnen Werte aufgeteilt. Hiervon werden jeweils zwei Werte – jeweils für die ersten beiden Gelenke von jedem Finger[13] – durch entsprechende Additionsknoten zusammengerechnet. Somit ergibt sich als deren Ausgabe die Gesamtkrümmung jeweils eines Fingers. Für die Erkennung dieser einfachen Handposturen wird der Daumen (Finger Nr. 1 in dem Programm) nicht betrachtet. Dadurch ist z.B. eine OK-Geste (eine geschlossene Hand mit nach oben gestreckten Daumen) des Erkenners nicht von einem einfachen Greifen zu unterscheiden. Eine Erweiterung um diese Feature ist aber im visuellen Programm sehr einfach zu realisieren und hier nur der Übersichtlichkeit wegen nicht aufgenommen. Somit wird eine Greifgeste einfach dadurch erkannt, dass der am wenigsten gebeugte Finger (der mit dem niedrigsten Wert) einen gewissen Schwellwert nicht unterschreiten darf. Für die Erkennung einer gestreckten Hand gilt im Gegenzug, dass der am meisten gebeugte Finger einen Schwellwert nicht überschreiten darf. Für die Erkennung einer Zeigeges-

[13] Das dritte Gelenk am Ende der Finger (abgesehen vom Daumen) wird in der Regel von den Datenhandschuhen nicht sensorisch erfasst, da sich dieses Gelenk nicht unabhängig von den anderen Gelenken des Fingers bewegen lässt.

te wird die Differenz zwischen der Beugung des Zeigefingers und der maximalen Streckung der drei anderen Finger berechnet und auch wieder mit einem passenden Schwellwert versehen.

Aber selbst bei diesen noch sehr einfachen Detektoren hat sich in der Praxis die Visualisierung der Ergebnisdaten, aber vor allem der Zwischenergebnisse als sehr hilfreich herausgestellt. Ein Hauptproblem ist bei der Erkennung von Handposturen die Qualität der Daten der Datenhandschuhe. Bei beiden in der „Virtuellen Werkstatt" eingesetzten Typen von Datenhandschuhen können diverse Fehler auftreten, die ohne eine Visualisierung der Eingangsdaten und Zwischenergebnisse oftmals nur schwer zu finden sind.

Fehlerursache	Hilfestellung zum Finden des Fehlers durch die Programmierumgebung
Komplette Aussetzer bei der Datengenerierung in der Sensorhardware	Werden in einem Datenkanal keine Daten übertragen wird dieses sofort bei der Visualisierung des Datendurchsatzes mittels Einfärbung der Verbindungen sichtbar. Der betroffene Strang „kühlt ab".
Zeitweilige Aussetzer durch eine unsaubere Funkübertragung bei kabellosen Datenhandschuhen	Im Gegensatz zum ersten Fall ist hier das direkte Feedback der Visualisierung („responsive VPL") in der Virtuellen Umgebung der Datenrate sehr hilfreich, um die eventuell nur sporadisch auftretenden Datenaussetzer zu beobachten.
Falsche Winkeldaten in den einzelnen Fingerdaten durch schlecht kalibrierte Werte	Hier können Ausreißer bei den kalibrierten Rohdaten der Handschuhe durch deren Visualisierung meistens direkt gefunden werden. Manchmal fallen Kalibrierungsprobleme aber auch in visualisierten Zwischenergebnissen der Berechnung auf, von wo aus dann durch die Verfolgung der Datenströme leicht Rückschlüsse auf den Ausgangsfehler zu schließen sind.
Invertierung der Winkeldaten durch Verdrehung der Bimetallstreifen im Handschuh	Hier sind bei der direkten Visualisierung der Rohdaten die gegenläufigen Daten während der Bewegung der Hände sehr schnell zu finden. Werden die Daten nicht im direkten Feedback zur Bewegung gezeigt, ist diese Art von Fehlern nur äußerst schwierig zu lokalisieren.
Fehlende oder inkorrekte Datenverbindung im visuellen Programm	Die Stelle kann schnell durch den Einbruch des Datendurchsatzes des gesamten Programmzweiges gefunden werden.

Tabelle 9: Mögliche Fehler bei der Gestenerkennung mit Datenhandschuhen und entsprechende Hilfe durch die visuelle Programmierumgebung

Tabelle 9 zeigt eine (nicht vollständige) Übersicht möglicher in der Praxis auftretender Fehler von den Daten der Datenhandschuhe bzw. des visuellen Programms und die entsprechenden Visualisierungsmethoden der visuellen Programmierung, die helfen können, diese Fehler aufzuspüren.

Beispiel: Trajektorienerkenner: Kreisbewegung

Ein weiteres Einsatzgebiet der visuellen Programme in der Gestenerkennung ist die Detektion von bestimmten Trajektorien. In diesem Beispiel soll eine kreisförmige Bewegung der Hand erkannt werden. Der Eingabestrom des Programms ist eine Transformationsmatrix, die z.B. von einem Tracker in der Virtuellen Umgebung erzeugt wird.

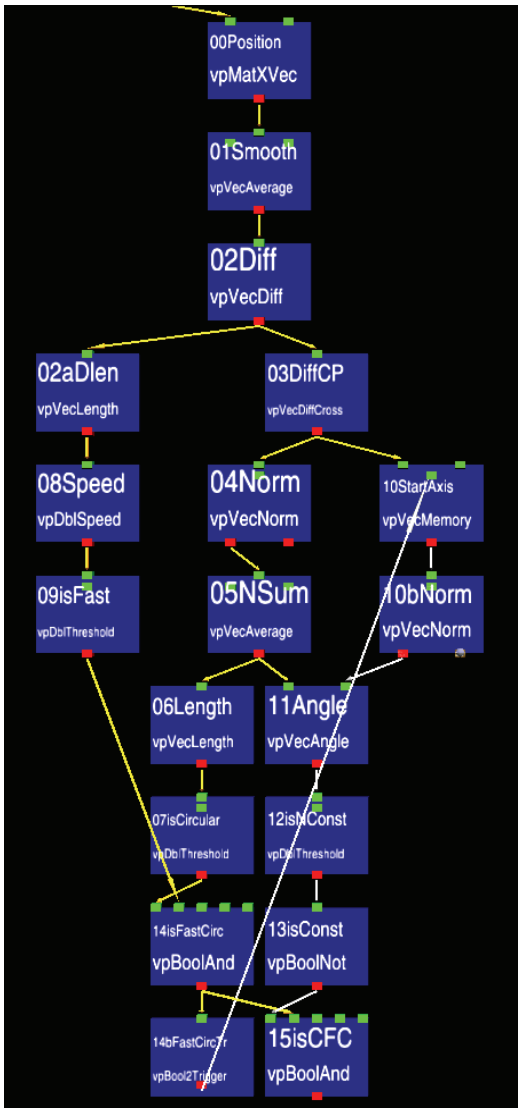


Abbildung 80: Visuelles Programm zur Erkennung einer kreisförmigen Trajektorie

Das visuelle Programm in Abbildung 80 realisiert diese Erkennung. Nach einer Glättung der eventuell verrauschten Werte wird die Trajektorie in einzelne Vektoren zerlegt, welche immer die Differenz zwischen der aktuellen und der letzten Position des Trackers enthalten. Aus diesen Vektorabschnitten werden dann in zwei getrennten Programmsträngen die Geschwindigkeit und die Kreisförmigkeit der Bewegung erkannt. Für die Detektion der kreisförmigen Bewegung, wird das Kreuzprodukte jeweils zweier aufeinander folgender Vektoren berechnet. Zeigt das über eine längere Zeit konstant in eine Richtung, liegt eine Kreisbewegung vor. Um zu erkennen, ob die Achse langsam die ursprüngliche Richtung verlässt, wird zu Beginn der Erkennung die Achse in einem „Memory“-Knoten gespeichert und mit der aktuellen Achse

verglichen. Die Verbindung, welche die Speicherung im „Memory“-Knoten triggert, ist als Rückwärtsverbindung im Programm angezeigt.

Die automatische Anordnung des Programmgraphen könnte durch das Verschieben des Knotens nach unten diese rückwärtige Verbindung auflösen. Da in dem Fall aber der gesamte davon anhängige Programmstrang auch nach unten wandert, was das Programm eher unübersichtlicher macht, wurde die Neuanordnung für diese Verbindung unterdrückt. Dieses ist möglich für Verbindungen von Parameter- und Triggerfeldern, welche keine Datensequenzen enthalten, da für diese Feedbackverbindungen im Graphen explizit erlaubt sind (siehe Abschnitt 4.4.1).

Beispiel: Skalierungsgeste

Die Geste, welche zur Skalierung von Bauteilen benutzt werden kann, besteht aus einen gleichzeitigen Greifgeste beider Hände, wobei die Innenseiten der Hände zueinander stehen müssen.

Die erkannte Geste entspricht der Übertragung der intuitiven Geste das Objekt an beiden Enden zu fassen und dann auseinander zu ziehen bzw. zusammen zu schieben. Der Skalierungswert ergibt sich dann aus dem Quotienten des aktuellen Abstandes der Hände zu dem Abstand am Zeitpunkt des Beginns der Greifgeste. Dafür wird dieser Abstand zu Beginn in einen Speicherknoten geschrieben und der aktuelle Wert durch diesen geteilt. Die Richtung des Differenzvektors der Positionen der Hände wird in der Übertragung auf die konkreten Skalierungswerte für die Bestimmung der Skalierungsachse benutzt.

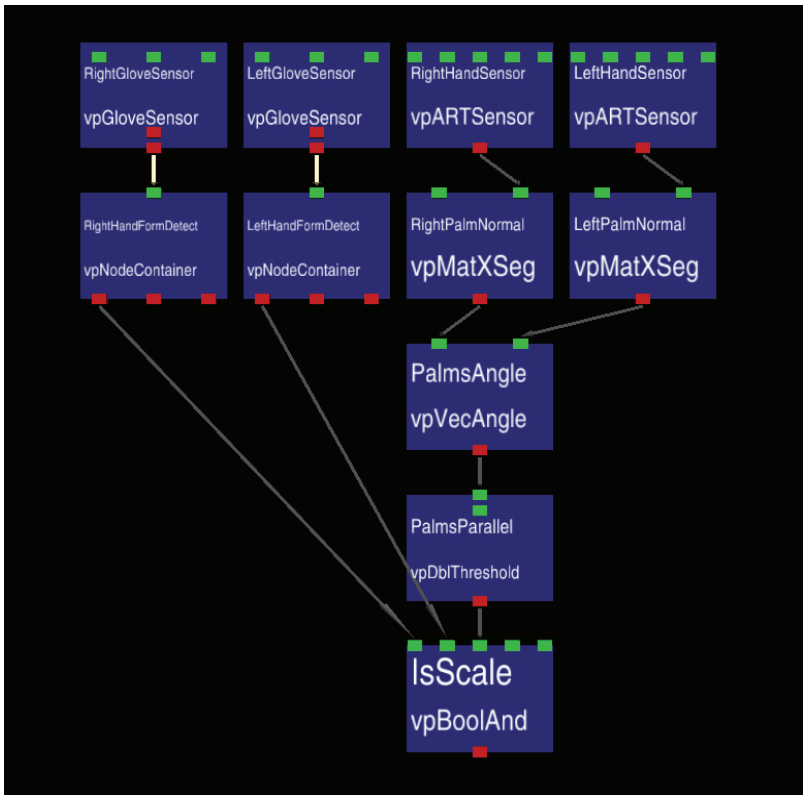


Abbildung 81: Visuelles Programm für die Erkennung einer Skalierungsgeste

Da für die Greifgeste schon ein Detektor besteht (siehe Beispiel oben) kann dieser direkt für die beiden Hände übernommen werden. Für die bessere Übersicht wurden diese beiden Detektoren in einem Knotencontainer gekapselt, um sie im visuellen Programm (siehe Abbildung 81) und auch anderen Programmen wieder verwenden zu können.

6.5.2 Sprach/Gesten-Integration

Da die Erkennung der Gesten immer exakt den Zeitpunkten der Eingabedaten zugeordnet werden kann, ist es möglich sehr genaue Aussagen über den zeitlichen Verlauf der Gesten zu machen. Auch wenn die Erkennung der Geste erst mit einer zeitlichen Verzögerung – z.B. durch den Einsatz von Glättungsoperatoren, welche immer einen zeitlichen Versatz der Daten zur Folge haben – erfolgt, kann der exakte ursprüngliche Zeitpunkt der erkannten Geste angegeben werden, da die Verrechnung der Daten immer zusammen mit den Zeitstempeln ihrer Ausgangswerte vorgenommen wird.

Diese zeitliche Exaktheit ist vor allem bei der multimodalen Integration der erkannten Gesten mit der dazu gesprochenen Sprache, wie sie in (Latoschik *et al.*, 1998) beschrieben werden, von großer Bedeutung. Die Integration von Sprache und Gestik wird in (Latoschik et al., 1998) durch ein spezielles *Augmented Transition Network (ATN)* realisiert.

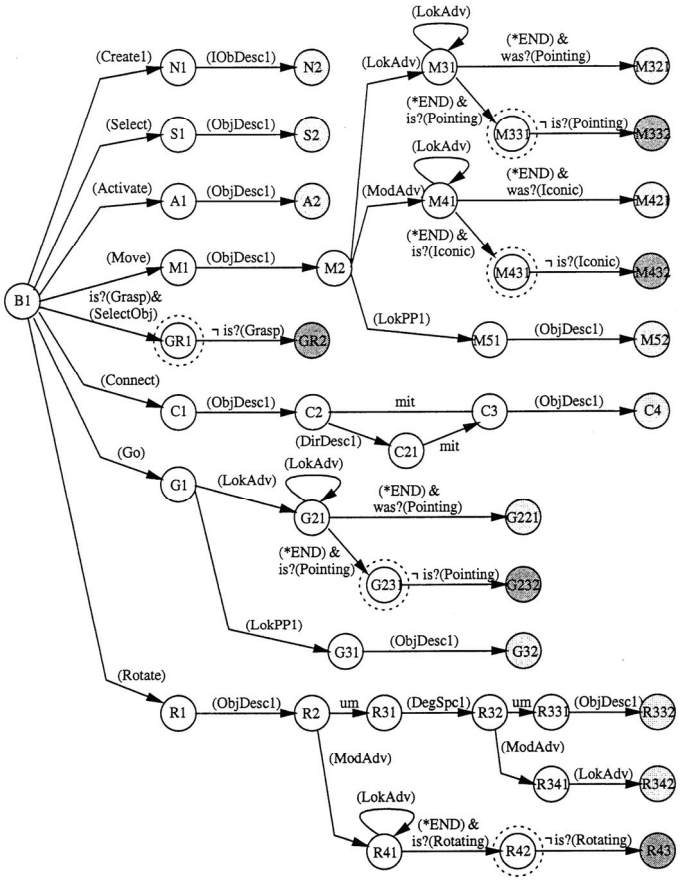


Abbildung 82: Beispiel eines ATN zur Sprach-Gesten-Integration – aus (Latoschik, 2001b)

Ein ATN entspricht einem endlichen Automaten, an dessen Übergängen der Zustände noch zusätzliche Bedingungen abgeprüft oder Aktionen ausgelöst werden können. Sowohl bestimmte Wörter als auch Gesten können die Übergänge im ATN veranlassen und schließlich bei erfolgreichem Ablauf einer Kette von Übergängen eine Interaktion in der virtuellen Welt auslösen.

Das Beispiel aus Abbildung 82 zeigt ein ATN das sprachliche Anweisungen mit erkannten Gesten integriert und gegebenenfalls Aktionen in der virtuellen Welt auslöst kann.

Der Mechanismus des ATNs hat sich für die multimodale Integration bewährt. Das vorliegende visuelle Programmiersystem bietet nun die Möglichkeit, diese externe Komponente mit in die visuellen Programme einzubauen. Wie in Abschnitt 4.8 gezeigt, können rein gestische Eingaben innerhalb des visuellen Programms schon direkt eine Interaktion auslösen. Da die Programmknoten durch die interne Programmierung mittels C++-Code problemlos externe Programmbibliotheken einbinden können, kann auch der Mechanismus des ATN innerhalb eines Programmknotens realisiert werden. Die Eingabeschnittstelle für diesen Knoten ist in dem Fall ein Feld, an dem die erkannten Worte eines Spracherkenners mit entsprechenden Zeitstempeln anliegen, und weitere Felder, über welche die erkannten Gesten an den Knoten weitergegeben werden können. Da die Gestenerkennung in den visuellen Programmen realisiert wird, sind diese Eingabedaten für die Gestik schon vorhanden. Das Spracherkenner kann als externes Modul die erkannten Wörter auch über einen Programmknoten zur Verfügung stellen. Soll der Erkenner als eigenständiges Programm – z.B. aufgrund von hohem Rechenaufwand auf einem dedizierten Rechner laufen – können diese Daten auch über Netzwerk an einen Multicast-Sensor (siehe Abschnitt 4.5.1) der virtuellen Programmierung gesendet werden. Dieser stellt die erkannten Worte dann über ein entsprechendes Feld dem visuellen Programm zur Verfügung. Als Ausgabe des ATN-Knotens stehen mehrere Triggerfelder bereit, welche mit den Knoten verbunden werden können, die dann gegebenenfalls die Interaktion in der Virtuellen Umgebung auslösen.

Da das ATN im aktuellen Einsatz nur als Implementierung in der Scheme-Scriptingsprache vorliegt und die Re-Implementierung als C++-Code in Arbeit, aber noch nicht abgeschlossen ist, konnte die konkrete Einbindung des ATN, wie oben beschrieben, leider noch nicht erfolgen. Daher liefert die Gestenerkennung in der aktuellen Implementation ihre Ergebnisse in Form von Datencontainern (siehe Abschnitt 4.5.1), welche durch ein Scheme-Interface die Daten der erkannten Gesten an das ATN liefern.

7 Einsatz des Systems

Der Vorläufer des hier vorgestellten visuellen Programmiersystems (Biermann & Wachsmuth, 2003), stellte die Rechenknoten und Constraint-Mediatoren, deren Verschaltung über das Scripting-Interface und eine Visualisierung der Daten und Programme bereit, ermöglichte aber bis dahin noch keine visuelle Editierung der Programme. Dieses Programmierumgebung wurde bereits in vielen Projekten der Arbeitsgruppe „Wissensbasierte Systeme“ in der Universität Bielefeld erfolgreich verwendet und kommt auch in den aktuellen Projekten zum Einsatz.

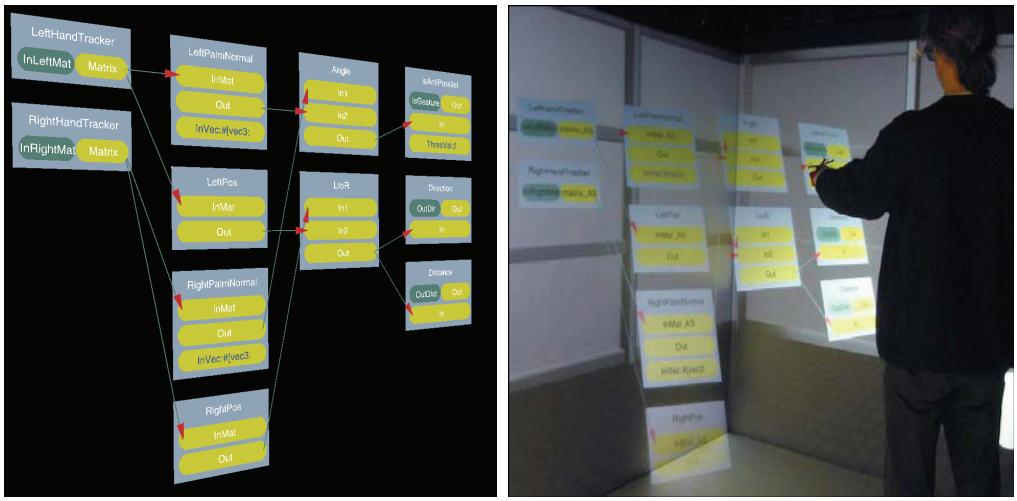


Abbildung 83: Visualisierung eines Datenflussprogramms im Vorläufer von VIPLIVE – aus (Biermann & Wachsmuth, 2003)

Abbildung 83 zeigt die Visualisierung eines Datenflussprogramms des Systems aus dem die aktuelle, visuelle Programmierumgebung entstanden ist. Das Programm realisiert die Erkennung einer Skalierungsgeste, bei der die Handflächen zueinander zeigen müssen, wobei zusätzlich die Achse zwischen den Händen und der Abstand der Hände erkannt werden.

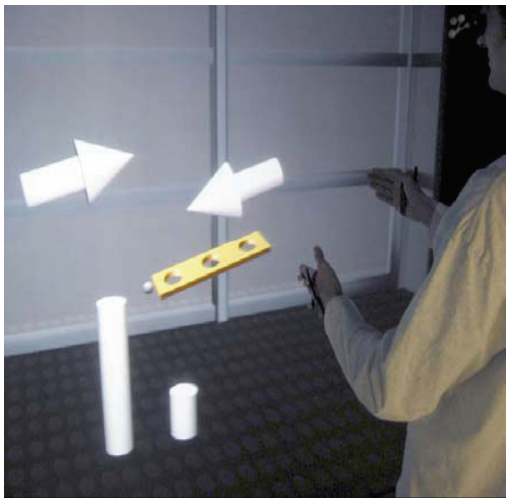


Abbildung 84: Visualisierung der Daten in der Virtuellen Umgebung – aus (Biermann & Wachsmuth, 2003)

In Abbildung 84 sind die Datenvisualisierungen von berechneten Werten des Datenflussprogramms zu sehen. Die Pfeile stellen Richtungsvektoren – in diesem Beispiel die Orientierung der Normalen auf den Handflächen des Benutzers – dar.

Die Visualisierung von einfachen numerischen Werten mit Hilfe der Zylinder gibt eine Abschätzung dieser Werte direkt während der Interaktion in der Virtuellen Umgebung. In diesem Fall zeigt der linke Zylinder eine Darstellung des Abstandes der beiden Hände (das Ergebnis des „Distance"-Rechenknotens im visuellen Programm) und der rechte Zylinder den Winkel zwischen den beiden Handflächennormalen (Ergebnis des „Angle"-Rechenknotens). Dazwischen ist das zu skalierende Objekt zu sehen.Der zentrale Einsatzzweck des Programmiersystems ist die Gestenerkennung und die Realisierung von geometrischen Constraints in dem Projekt „Virtuelle Werkstatt". Beispiele aus diesem Projekt sind schon in Kapitel 1 aufgezeigt worden.

Es konnten aber auch weitere Projekte von der Programmierumgebung und den damit realisierbaren Constraints profitieren. Da der Einsatz in verschiedenen Projekten mit zum Teil sehr unterschiedlichen Einsatzgebieten sehr gut die von der Konzeption des Systems angestrebte Wiederverwendbarkeit und die flexiblen, unkomplizierten Anpassungs- und Erweiterungsmöglichkeiten demonstriert, wird im Folgenden kurz der Einsatz des Programmiersystems in diesen Projekten aufgezeigt.

7.1 Erkennung ikonischer Gesten

Im Rahmen der Arbeiten von Timo Sowa (Sowa, 2006) wird das Programmiersystem für die Erkennung und Klassifikation von ikonischen Gesten benutzt. Die Erkennung der Gesten im Bereich dieser Arbeit geht über die bis dahin benutzten Programme zur Gestenerkennung im Bereich der „Virtuellen Werkstatt" hinaus. Realisiert wurden hauptsächlich Erkenner für Trajektorien und Positionen der Hände und für verschiedene Handposturen. Dafür wurden die schon vorhandenen Detektoren z.B. im Bereich der Segmentierung von Bewegungstrajektorien und der Erkennung komplexerer Handformen erweitert.

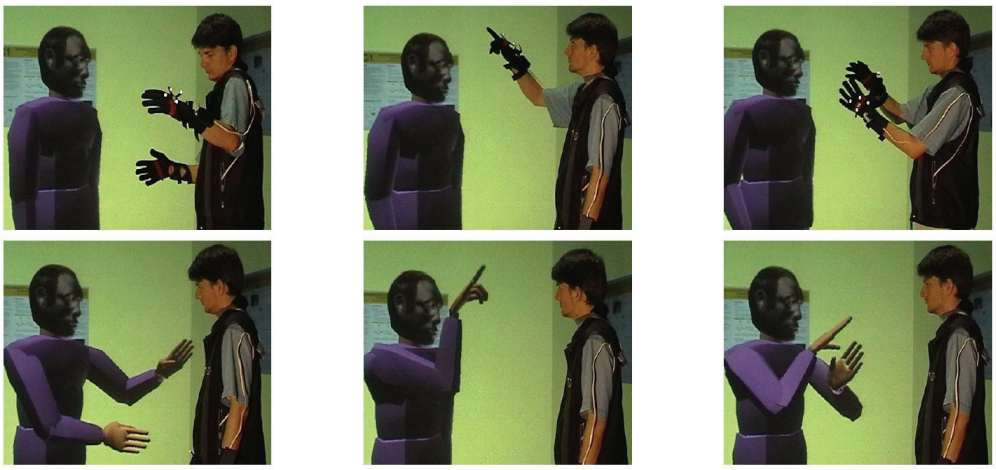


Abbildung 85: Imitationsspiel zur Demonstration der erfolgreichen Gestenerkennung – aus (Kopp et al., 2004)

Als Demonstrator für die erfolgreiche Erkennung und die mögliche Reproduktion der erkannten Gesten wurde ein Imitationsszenario aufgebaut in dem ein virtueller,

anthropomorpher Agent die erkannten Gesten eines Menschen in der Virtuellen Umgebung imitiert. Abbildung 85 zeigt die Imitation von drei Gesten, wobei jeweils die Position und Form der Hände vom virtuellen Agenten imitiert werden.

Für diesen Einsatz des Systems war auch die präzise Berechnung der Daten mittels Datensequenzen von großer Bedeutung, da eine Berechnung der Werte im Takt der Renderloop nicht ausreichte, um eine verlässliche Erkennung z.B. in der Analyse der Bewegungstrajektorien zu gewährleisten.

7.2 Auswertung von Zeigegesten

Im Projekt DEIKON (DEIxis in KONstruktionsdialogen, Teilprojekt B3 des Sonderforschungbereichs 360) wurde die Programmierumgebung genutzt, um Zeigegesten von Benutzern, aber auch des, in (Kopp *et al.*, 2003) vorgestellten, virtuellen Agenten „Max“, in der Virtuellen Umgebung auszuwerten und auch zu visualisieren. Bei der Auswertung der Zeigegeste kam es nach der Berechnung des aktuellen Zeigestrahls durch ein einfaches Datenflussprogramm vor allem auf die Auflösung der referenzierten Objekte an. Hierbei kam ein Rechenknoten zum Einsatz, der ausgehen von dem Segment, das die aktuellen Zeigerichtung und Position beschreibt, die Objekte im Szenengraphen sucht, die innerhalb eines vorgegebenen Winkelabweichung vom Zeigestrahl liegen. Die Suche erfolgt über die Semantic-Entities (siehe Abschnitt 6.1), welche die virtuellen Objekte im Szenengraph repräsentieren und über die die Eigenschaften der entsprechenden Objekte abgefragt werden können. Je nach der Anzahl und Position der Objekte, welche durch die Zeigegeste möglicherweise referenziert werden können, werden weitere Attribute der Objekte bestimmt, welche die Mehrdeutigkeit der Geste auflösen sollen.

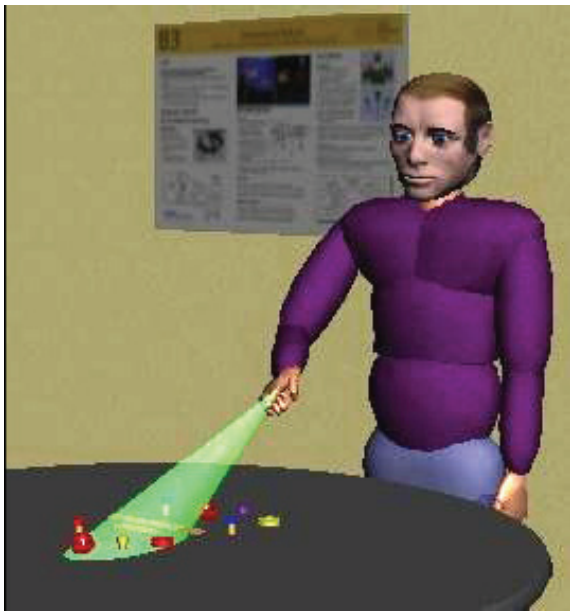


Abbildung 86: Visualisierung eines Zeigekegels des virtuellen Agenten – aus (Kranstedt & Wachsmuth, 2005)

Zusätzlich hilft eine Visualisierung des Zeigestrahls in Form eines Kegels, welcher zusätzlich zur Position und Richtung auch die maximale Winkelabweichung für die Referenzierung der Objekte anzeigen kann, bei der Übersicht über den Ablauf des Prozesses. Abbildung 86 zeigt die Visualisierung des Zeigekegels während einer Zeigegeste des virtuellen Agenten „Max“.

7.3 Visualisierungstool für Experimentierdaten

Die Implementierung eines Visualisierungstools für Experimentierdaten im Bereich der Gestenforschung („IADE“) (Pfeiffer *et al.*, 2006) wurde auch mit Hilfe der Programmierumgebung realisiert. In diesem Fall liegt der Schwerpunkt weniger auf einer komplexen Berechnung von Daten als einer zeitlich gesteuerten Visualisierung von Trackingdaten aus einem Experimentierdatensatz. Auch werden hier keine Daten direkt von den Trackingsensoren ausgewertet, sondern zuvor in Experimenten aufgenommene Datensätze zur genauen Analyse abgespielt. Da die Daten nur offline analysiert werden, ist dieses als ein Bereich anzusehen, in dem die Präzision der Daten auf jeden Fall Vorrang vor dem Latenzverhalten der Berechnung hat.

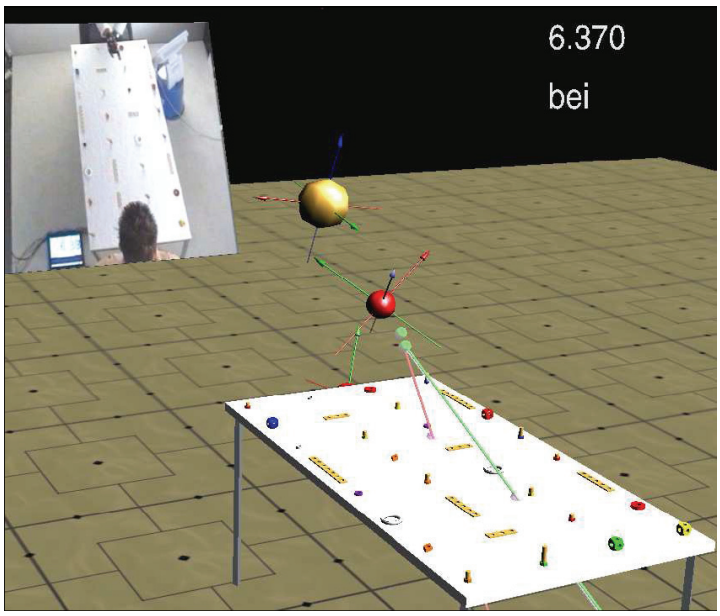


Abbildung 87: Das Visualisierungstool IADE – aus (Pfeiffer et al., 2006)

Abbildung 87 zeigt eine Szene im IADE-Tool, in der zum einem direkt die getrackten Daten der Versuchperson, wie die Position des Kopfes (braune Kugel), Position der Hände (rote Kugeln, jeweils wie beim Kopf mit Koordinatensystem für die Darstellung der Orientierung) und der Position zweier Marker an dem Zeigefinger der Versuchperson visualisiert werden. Zusätzlich werden zwei durch einfache Datenflussprogramme berechnete Segmente und deren Schnittpunkte mit der Tischfläche visualisiert, wobei das grüne Segment eine Zeigerichtung von Zeigefinger ausgehend in Richtung der Verlängerung des Kopf-Hand-Vektors und das rote Segment direkt die Zeigerichtung des Fingers darstellt. In der Szene ist außerdem eine virtuelle Rekonstruktion der Stimulusobjekte und gleichzeitig das aufgenommene Videobild des Versuchablaufes synchron zu der Datenvisualisierung angezeigt. In der rechten oberen Ecke wird der Zeitpunkt des Dargestellten Datensatzes und das zu dem Zeitpunkt gesprochene Wort der Versuchsperson angezeigt.

Aus dem vorhandenen Programmiersystem konnten sowohl die Visualisierung der Positionen der aufgenommenen Rohdaten, als auch die der Segmente direkt übernommen werden. Auch für die Berechnung der abgeleiteten Daten der beiden Zeigesegmente konnte durch die Verschaltung weniger Programmknoten komfortabel ein Datenflussprogramm erstellt werden.

Dieses Visualisierungstool kann aber vor allem durch den interaktiven Aufbau der Berechnungen und Visualisierungen noch ein zusätzliches Feature gewinnen. Damit ist dann eine Datenvisualisierung durch den Aufbau visueller Programme möglich, wie sie innerhalb professioneller Visualisierungstools (von denen Beispiele in Abschnitt 2.4.3 vorgestellt wurden) in Form von zweidimensionalen visuellen Programmen schon eingesetzt werden. Dieser Schritt zum Ausbau des Tools zur freien Konfiguration der visuellen Programme und Visualisierungen ist vorgesehen.

7.4 Erkenner für Turntaking-Signale

Im Projekt Systemintegration (Teilprojekt D3 des Sonderforschungbereiches 360) konnte mit Hilfe des Programmiersystems bei geringem Aufwand ein Detektor für einfache Turntaking-Signale in den Gesten des Benutzers erstellt und zum Einsatz gebracht werden. Der Detektor reagiert auf Gesten des Benutzers, die signalisieren sollen, dass der Benutzer die aktuelle Äußerung eines virtuellen Agenten unterbrechen möchte, um selber den das Gespräch fortzusetzen. Erkannt werden hierbei u. a. Handform, Richtung der Handfläche und die Höhe der Hand. Da für viele dieser Features schon Detektoren als Rechenknoten beziehungsweise Datenflussprogramme im Bereich anderer Projekte realisiert waren, konnte der konkrete Detektor für die Turntaking-Signale auch dank der einfachen Wiederverwendbarkeit des Berechnungsnetzwerke sehr schnell durch die Kombination und Erweiterung der vorhanden Programme realisiert werden.

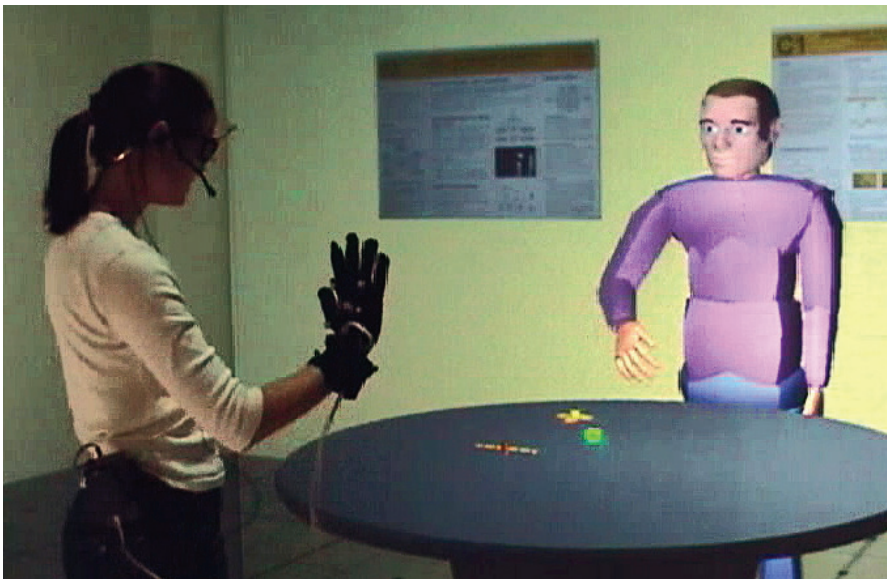


Abbildung 88: Der Benutzer unterbricht einen virtuellen Agenten mittels einer Geste – aus (Leßmann et al., 2004)

Abbildung 88 zeigt den Benutzer während er durch das Heben der Hand mit der Handfläche in Richtung des virtuellen Agenten – und eventuell einer zusätzlichen sprachlichen Äußerung – signalisiert, dass er die aktuelle Äußerung des Agenten unterbrechen möchte. Geben die Geste oder die sprachliche Äußerung genügend Anlass, unterbricht sich der virtuelle Agent und übergibt den Turn – unterstützt durch eine zusätzliche Nachfrage – an den Benutzer.

7.5 Definition von Positionen der Karten in einem virtuellen

Kartenspiel

Während die ersten drei Beispiele das Programmiersystem hauptsächlich im Bereich der Berechnungsnetzwerke ausnutzen, wurde auch die Möglichkeit der geometrischen Constraints über die Constraint-Mediatoren und der komfortablen Modellierung von Verbindungsstellen und der parametrischen Anpassung der Transformationen von virtuellen Objekten mit Hilfe des VPML-Formats von Projekten außerhalb der Virtuellen Werkstatt genutzt.

In diesem Beispiel konnten die möglichen Positionen von Karten während eines virtuellen Kartenspiels zwischen einem menschlichen Benutzer und einem virtuellen Agenten komfortabel definiert werden. Dafür wurde jeweils an der Oberseite und Unterseite der Karten, sowie an den möglichen Ablageplätzen, ein fester Port definiert. Mittels Verbindungen dieser Ports lassen sich die Karten sowohl auf den vorgegebenen Stellen als auch als Stapel übereinander platzieren. Die zentrale Steuerung des Spielablaufs muss dann nur noch die abstrakten Positionen (wie z.B. „Ablageposition 1“ oder „auf Karte Nr. 8“) der Karten vorgeben und die entsprechenden Portverbindungen etablieren bzw. lösen. Die geometrische Anordnung der Karten erfolgt dann automatisch durch den von der Portverbindung vorgegeben geometrischen Constraint. Das VPML-Format erlaubt hierbei die komfortable Anreicherung der vorgegebenen Geometrien mit den oben beschriebenen Ports. Die Constraint-Mediatoren, welche die Portverbindungen überwachen, sorgen für die korrekte Transformation der Karten.

```
<Part name="SaMCard">
   <Geometry csg="+">
      <File
filename="Data/Models/SaMCard_one.iv"/>
   </Geometry>
   <Port name="TopOfCard" type="pointport"
      rotation="90 0 1 0" translation="0 0 0.1"/>
   <Port name="BottomOfCard" type="pointport"
      rotation="90 0 1 0" translation="0 0 0"/>
</Part>
```

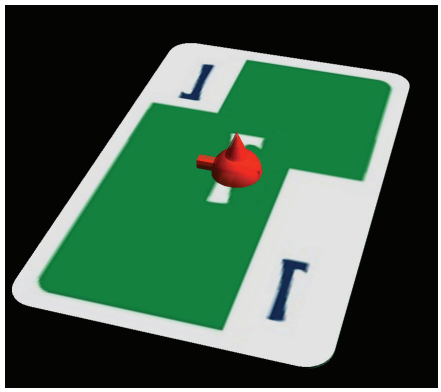


Abbildung 89: VPML-Definition und entsprechende Spielkarte mit Portvisualisierung

Der linke Teil von Abbildung 89 zeigt das VPML-File für die Definition der beiden Ports an einer Spielkarte und rechts eine Visualisierung einer Portstelle an der Spielkarte. Das VPML-File besteht nur aus der Angabe des Files für die Kartengeometrie und zwei Ports, jeweils an der Unter-, bzw. Oberseite der Karte.

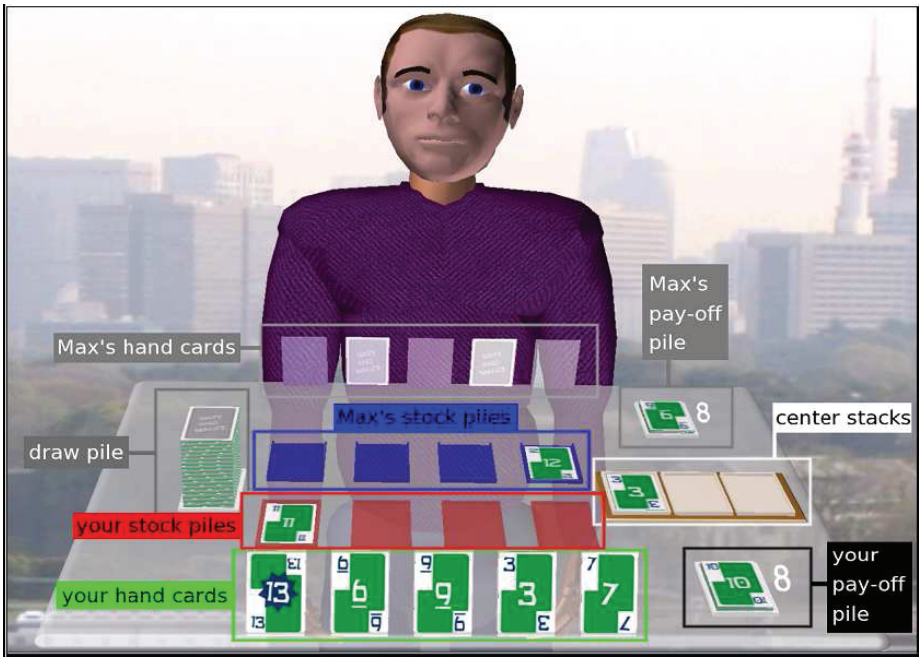


Abbildung 90: Szene aus dem virtuellen Kartenspiel mit Max – aus (Becker et al., 2005)

Abbildung 90 zeigt eine Szene aus einem Spiel zwischen „Max“ und einem menschlichen Benutzer.

7.6 Skalierbare virtuelle Bildschirme für Videokonferenzen

In dem EU-Projekt PASION konnte in einem Rapid-Prototyping Verfahren u. a. mit Hilfe der vorhandenen Interaktionen in der Virtuellen Umgebung, aber auch durch die schnelle und einfache Definition von parametrisierbaren Geometrien mittels VPML-Dateien, ein Prototyp für eine interaktive, Virtuelle Umgebung für die Kommunikation geschaffen werden. Sowohl die Drag-and-Drop Operationen als auch die Interaktion für die Skalierungsoperationen der virtuellen Bildschirme konnten mit wenigen Anpassungen von bestehenden Interaktionsszenarios der Virtuellen Werkstatt übernommen werden, was auch die gute Übertragbarkeit und Anpassungsmöglichkeit der vorhandenen Programme der Programmierumgebung zeigt.

Die in VPML-Files definierten, gekoppelten Parameter, welche ursprünglich für komplexe Skalierungen und funktional abhängigen Transformationen innerhalb von Bauteilen geschaffen wurden, zeigten sich hier nützlich bei der Festlegung von skalierbaren virtuellen Bildschirmen. Der rechte und linke Rahmenteil der Bildschirme sollte sich mit der Höhe der Bildschirme skalieren, die Breite des Rahmens hingegen nur innerhalb eines fest vorgegebenen Wertes.

Abbildung 91: Skalierbare virtuelle Bildschirme aus dem PASION-Projekt

In Abbildung 91 sieht man einen skalierbaren virtuellen Bildschirm in der Virtuellen Umgebung aus dem Prototyp für das PASION-Projekt. Die linke Seite der Abbildung zeigt eine Skalierung des Bildschirms der Breite, auf der rechten Seite eine Skalierung der Höhe. Auf dem virtuellen Tisch vor dem Benutzer liegen noch weitere, unskalierte Bildschirme.

Die Angabe eines unterschiedlichen Skalierungsverhaltes pro Achse eines Unterteils ist im VPML einfach durch die Angabe des „ParentScaling“ zu erreichen (siehe Abschnitt 6.2.2). Da die Rahmen der Bildschirme in der horizontalen Richtung ein spezielles Skalierungsverhalten zeigen sollten, musste das VPML-Format um zwei Attribute in der Definition des Tag „ParentScaling“ erweitert werden.

8 Resümee und Ausblick

Dieses Kapitel gibt eine Übersicht der erreichten Ziele der in dieser Arbeit entwickelten Programmierumgebung. Im zweiten Abschnitt wird auf die Erfahrungen mit dem System im praktischen Einsatz eingegangen. Der letzte Abschnitt gibt einen Ausblick auf mögliche, zukünftige Erweiterungen des Systems.

8.1 Erreichte Ziele

In dieser Arbeit wurde ein visuelles Programmiersystem vorgestellt, das die Möglichkeit bietet, lokale Constraints für den Einsatz in der Virtuellen Konstruktion zu realisieren. Die dafür eingebrachten Neuerungen in vorhandene Systeme und die Einbindung vorhandener Konzepte spielen sich in mehreren Bereichen ab:

Die Ansätze von Datenflusskonstrukten aktueller VR-Tools wurden zu einer vollständigen Datenflussprogrammierung ausgebaut. Die Visualisierung der Programmknoten und Interaktion zum Aufbau der Datenflussprogramme schafft eine visuelle Programmierumgebung in einer immersiven Virtuellen Umgebung. Diese Art der modularen visuellen Datenflussprogrammierung erlaubt neben der Parallelisierung von Programmabläufen – z.B. für Multiprozessormaschinen – auch eine sehr einfache Wiederverwendung von Programmteilen. Eine zusätzliche Visualisierung der Abläufe und Zwischenergebnisse der visuellen Programme erweitert diese zu einer reaktiven visuellen Programmiersprache, die es erlaubt, Programmabläufe während ihrer Ausführung in der Virtuellen Umgebung zu testen, analysieren und zu optimieren.

Für die speziellen Bedürfnisse an die Berechnungen in echtzeitbasierter Virtueller Realität wurde das existierende PrOSA-Framework für von der Framerate der Applikation unabhängige Berechnungen in die visuelle Datenflussprogrammierung übernommen und um eine flexible Anpassung auf die unterschiedlichen Erfordernisse nach Exaktheit bzw. geringe Latenzzeiten der Daten erweitert. Das ermöglicht mit Hilfe von Zeitstempeln auch die exakte zeitliche Einordnung der berechneten Daten. Dabei wurde bei der Realisierung des Systems auf die für Berechnungen in Echtzeit nötige Leistungsstärke der eingesetzten Programmiersprachen und Konzepte der Datenflussprogrammierung geachtet.

Für die Definition neuer Programmknoten in dem visuellen Programmiersystem wurde ein spezielle, XML-basierte Beschreibungssprache (VIPML) entwickelt. In dieser Sprache können das Feldinterface und die Berechnungsoperationen der Programmknoten komfortabel beschrieben werden. Die bereitgestellte Schnittstelle für den Zugriff auf Feldwerte und das Erzeugen neuer Datentoken auf den Ausgabefeldern nimmt dem Programmierer die komplexe Behandlung der Datenstrukturen für asynchrone Daten und deren Integration und Interpolation ab. Dadurch wird der Code in den Beschreibungen der Programmknoten sehr übersichtlich gehalten.

Um keine Spezialisierung auf ein VR-Tool unter der Vielzahl von eingesetzten VR-Systemen vorzunehmen, wurde ein universeller Szenengraph-Layer (USG) entwickelt. Dieser abstrahiert die konkreten Programmierschnittstellen der Systeme, damit der Code der Programmknoten kompatibel in verschiedenen VR-Systemen eingesetzt werden kann. Der USG-Layer wurde für zwei VR-Tools (AVANGO und OpenSG) implementiert, wodurch das virtuelle Programmiersystem auch in beiden Systemen benutzbar ist. Eine weitere Programmierschnittstelle der Feld-Event-Layer kapselt

den Zugriff auf die Feldwerte der VR-Tools und implementiert die für das Eventsystem der visuellen Programmierung eingesetzten Berechnungsknoten.

Das strikte Ablaufschema der Datenflussprogrammierung erlaubt nicht, Constraints mit Hilfe der Programmknoten zu realisieren. Dies wird aber durch geringfügige Erweiterungen der Programmknoten um Feldreferenzen und um eine zusätzliche Eventpropagierung bewerkstelligt. Die dadurch entstandenen Constraint-Mediatoren realisieren bei Berücksichtigung der vorhandenen Constraints durch die Szenengraphstruktur oder Feldverbindungen flexible lokale Constraints. Die dadurch definierbaren geometrischen Constraints verwirklichen komplexe Verbindungsbeziehungen der virtuellen Bauteile und parametrische Veränderungen, wie z.B. inhomogene Skalierungen. Die Propagierung von Transformationen über Constraints von Getrieben und Verbindungsstellen erlaubt die Simulation von kinematischen Ketten der in der virtuellen Konstruktion zusammengebauten Aggregate.

Für die Erstellung der Szenengraphabschnitte inklusive der benötigten Constraint-Mediatoren, wurde eine XML-basierte Beschreibungssprache für parametrisch veränderbare und skalierbare Bauteile (VPML) entwickelt. Diese ermöglicht die mühelose Definition der Eigenschaften von Bauteilen für die virtuelle Konstruktion.

Die weiteren Eigenschaften der Bauteile, welche in einem semantischen Netz bereitgehalten werden, sind über das eingebundene Konzept der Semantik-Entities direkt über die Szenengraphstruktur zugreifbar. Das erlaubt unter anderem, die komplette Ablauflogik der virtuellen Welt über die visuellen Programme zu definieren. Hier ermöglicht die Einbindung der Integration von Sprache und Gestik über den Mechanismus des ATN als Knoten in den visuellen Programmen die Steuerung der gesamten Applikation über eine natürliche Interaktion.

8.2 Der praktische Einsatz des Systems

Für den Einsatz des Systems in der Praxis sind vor allem die relativ kurze Einarbeitungszeit für die Erstellung der Programme (d.h. der Verschaltung der vorgegebenen Rechenknoten) und die komfortable Möglichkeit der Erweiterung der Rechenknoten über das XML-Format entscheidend. Die Möglichkeit der Visualisierung von Daten direkt in der Virtuellen Umgebung erleichtert das Finden von Fehlern in den Programmen, aber auch den Aufbau umfangreicher Visualisierungsumgebungen für komplexe Daten. So stellt das vorgestellte visuelle Programmiersystem eine komfortable Programmierumgebung zur Verfügung, in die sogar ungeübte Programmierer einen schnellen Einstieg finden und zumindest vorhandene Programme anpassen können.

Durch den streng modularen Aufbau der Programme, das feste Interface der Programmknoten-Felder und die Vorgaben der internen Programmstruktur des XML-Formats ergibt sich ein schnell wachsender Satz von Programmeinheiten. Auch wenn diese von unterschiedlichen Programmierern erstellt sind, können sie direkt von anderen als Programmknoten in eigenen visuellen Programmen eingesetzt, konfiguriert und verschaltet werden. Auch bereits zu komplexeren Programmen verschaltete Knoten können – indem sie in einem Knotencontainer zusammengefasst und mit einem Feldinterface versehen werden – einfach wieder verwendet und ausgetauscht werden.

Sollte dennoch aufgrund der Komplexität des simulierten Systems die Rechenkapazität eines einzelnen Rechners nicht ausreichen, können Teile des Programms als

Vorverarbeitungsschritte auf andere Rechner ausgelagert werden und deren Ergebnisse über eine vom System bereitgestellte Netzwerkkommunikation ausgetauscht werden. Diese Netzwerkanbindung erlaubt auch die einfache Einbindung der externen Sensorhardware, deren Daten auf dedizierten Rechnern berechnet und dann über das Netzwerk zur Verfügung gestellt werden.

Für andere Simulationskomponenten, wie z.B. die Kollisionserkennung oder eine physikalische Dynamikberechnung der Objekte in der Virtuellen Umgebung, ist die Anbindung eines externen Softwarepakets vorzuziehen, da in dem Bereich sehr gute Systeme existieren, und diese eine relativ fest vorgegebene Funktion haben. Die Gestenerkennung und Integration mit der Sprache des Benutzers bestimmen zusammen mit den modellierten Constraints ein hoch individuelles Verhalten der Applikation, das durch die Programmierung im Einzelnen festgelegt werden muss. Hingegen können Softwarepakete, die Kollisionen der Bauteile und die eventuell resultierenden physikalischen Kräfte berechnen, ohne größere Anpassung der internen Berechnungen eingesetzt werden. Da das VIPLIVE-Programmiersystem zum einem von den Frameraten unabhängige Berechnungen machen kann, zum anderen diese Berechnungen aber an fest definierten Zeiten in der Abarbeitungsschleife der VR-Applikation gemacht werden, ist die Möglichkeit der Einbindung dieser externen Programmpakete mit einem kontrollierten Abgleich der errechneten Daten gegeben. Dieses wurde auch am Beispiel der Kollisionserkennung mit Hilfe einer externen Kollisions-Engine im System realisiert.

Eine noch weitere Abkopplung der externen Berechnungen in der Virtuellen Umgebung wird über ein Agentensystem erreicht, in dem einzelne unabhängige Agenten die Berechnungen durchführen und die Ergebnisse über eine Agentenkommunikation austauschen – siehe auch (Wachsmuth, 1998). Diese Art der Anbindung externer Rechenprozesse ist vor allem für sehr aufwendige Berechnungen ideal, deren Resultate nicht für den Echtzeitablauf der virtuellen Simulation erforderlich sind. In diesem Fall kann die Anforderung einer Berechnung über die Agentenkommunikation verschickt und das berechnete Ergebnis dann in das Programm eingebunden werden, sobald es vorliegt. Ein Beispiel hierfür ist der externe ACIS-Agent, der die aufwendige Berechnung der polygonalen Darstellung von skalierten CSG-Modellen vornimmt.

8.3 Weiterentwicklungen des Systems

Das hier vorgestellte Programmiersystem bildet eine Grundlage für komplexe Interaktionen in einer Virtuellen Umgebung. Einige Projekte haben schon von den Möglichkeiten des Systems profitiert und weitere sollen folgen.

Ein Hauptproblem für externe Benutzer des Programmiersystems war bisher die Unsicherheit durch viele Weiterentwicklungen und teilweise Re-Implementierung der Basisfunktionalitäten des Systems, um die jetzige Funktionalität und allgemeine Benutzbarkeit sicherzustellen. Auch die ursprüngliche Abhängigkeit von speziellen Softwarepaketen (AVANGO, OpenGL-Perfomer) beschränkte die Möglichkeiten eines umfassenderen Einsatzes.

Zum aktuellen Zeitpunkt hat die Basis des Systems einen sehr stabilen Stand erreicht und mit den generellen Programmierschnittstellen (USG- und Field-Event-Layer) können viele Erweiterungen demnächst außerhalb der Basisfunktionalität – ohne Abhängigkeiten von konkreten Szenegraphtools oder Eventsystemen – hinzugefügt

werden. Dadurch wird der Einsatz in externen Entwicklungen noch weiter vorangetrieben.

Eine Weiterentwicklung könnte die Auslagerung von etablierten Teilen der visuellen Programme auf ein reines (nicht visuelles) Datenflusskonzept betreffen. Während in der aktuellen Implementation fast alle Programmknoten ihre eigene Geometrie und Methoden für die Behandlung ihrer Visualisierung bereitstellen, können visuelle Programmteile, die eigentlich nicht mehr visuell verändert werden müssen, durch wesentlich einfachere Programmknoten ohne visuelle Unterstützung realisiert werden. Hierdurch wird die Graphiklast der Virtuellen Umgebung, aber auch der Speicherbedarf der Programme selber verringert.

Die aktuellen Erkenner für Handposturen oder Trajektorien sind über rein algorithmische, symbolische Programmstrukturen implementiert. In einigen Bereichen z.B. der Klassifikation kann der Einsatz von konnektionistischen Methoden, wie z.B. künstlichen Neuronalen Netzen, die Erkennung verbessern. Auch im Bereich der neuronalen Netze gibt es schon Ansätze für visuelle Programmierumgebungen (siehe z.B. Neo/NST (Ritter)). Hier kann eine reaktive virtuelle Programmierumgebung eine interessante Plattform für die Visualisierung der Abläufe in einem Neuronalen Netz oder der interaktiven Re-Konfiguration bieten. Erste Ansätze für die Einbindung von Neuronalen Netzen in das bestehende System sind bereits vorhanden.

Die Parallelisierung der Recheneinheiten, welche eine sehr gute Ausnutzung der aktuellen Hardware erlaubt, ist vorgesehen und lässt sich in die Datenflussarchitektur einfach einfügen, wurde aber bisher noch nicht realisiert. Gerade aktuelle Multiprozessor- und Hyperthreading-Systeme ermöglichen eine beschleunigte parallele Abarbeitung von Programmen. Hierbei kann ein Optimum gefunden werden, das – je nach den Möglichkeiten zur Parallelisierung der eingesetzten Hardware – zwischen einem linear abgearbeiteten Programm und einer fein granularen Aufteilung der Programme in mehrere parallele Threads liegt.

Für den Bereich der aktuell sehr primitiven graphischen Darstellung der Programme, können in Zukunft bessere Visualisierungen implementiert werden. Dazu gehört z.B. eine Verbesserung der Anordnung der Knoten im Programm und der Felder eines Programmknotens.

Die in Abschnitt 6.5.2 schon ausgeführte Einbindung der Integration von Sprache und Gestik über das ATN als Komponente in den visuellen Programmen kann in Zukunft die Ablauflogik der Applikation durch das Eventsystem steuern. Daraus resultiert eine sehr übersichtliche und visuell editierbare Darstellung der gesamten Abläufe in der Applikation.

Ein Vorteil der Implementation mittels AVANGO gegenüber der aktuellen, unabhängigen Realisierung des Programmiersystems ist die Einbindung einer Skripting-Sprache, mit der sich z.B. Programmknoten erzeugen, Felder verbinden und Feldwerte setzen können. Im Bereich des Einsatzes des aktuellen Systems innerhalb des AVANGO-Frameworks ist eine Skripting-Anbindung geschaffen worden. Eine Anbindung des Programmiersystems an ein allgemeineres Skriptingkonzept könnte die geschaffene Umgebung im Komfort noch verbessern.

9 Literatur

Abelson, H., Adams, N. I., Bartley, D. H., Brooks, G., Dybvig, R. K., Friedman, D. P., et al. (1991). Revised 4 Report on the Algorithmic Language Scheme. *Lisp Pointers, 4*(3), 1-55.

Abram, G., & Ternish, L. (1995). An Extended Data-Flow Architecture for Data Analysis and Visualization. *Computer Graphics, 29*(2), 17-21.

Ackerman, W. (1982). Data Flow Languages. *IEEE Computer, 15*(2), 15-25.

Apple Inc. (2006). Quartz Composer Programming Guide. Retrieved August, 2006, from http: //developer.apple.com/documentation/GraphicsImaging/Conceptual/QuartzComposer/

AVS. Advanced Visual Systems (AVS). 2006, from http://www.avs.com/products/index.html

Backus, J. (1978). Can Programming be Liberated from the Von Neumann Style? A Functional Style and Its Algrebra of Programs. *Communications of the ACM, 21*(8), 613-641.

Baraff, D. (1994). *Fast Contact Force Computation for Nonpenetrating Rigid Bodies.* Proceedings of Computer Graphics (SIGGRAPH 94), pp. 23-34.

Barahona, P., & Gurd, J. R. (1985, Sept.). *Simulated Performance of the Manchester Multi-Ring Dataflow Machine.* Proceedings of 2nd ICPC, pp. 419-424.

Baroth, E., Hartsough, C., & Burnett, M. (1995). *Visual Programming in the Real World.* Proceedings of Visual Object-Oriented Programming Concepts and Environments, pp. 21-42, Greenwich, CT.

Beck, R. J., & Keshav, P. M. (1991). From Control Flow to Dataflow. *Journal of Parallel and Distributed Computing, 12*, 118-129.

Becker, C., Prendinger, H., Ishizuka, M., & Wachsmuth, I. (2005). *Evaluating Affective Feedback of the 3D Agent Max in a Competitive Cards Game.* Proceedings of The First International Conference on Affective Computing and Intelligent Interaction (ACII-05), pp. 466-473, China, Beijing.

Bernini, M., & Mosconi, M. (1994). *VIPERS: A Data Flow Visual Programming Environment Based on the Tcl Language.* Proceedings of Workshop on Advanced Visual Interfaces., pp. 243-245.

Bichler, L., Radermacher, A., & Schürr, A. (2004). Integrating Data Flow Equations with Uml/Realtime. *Real-Time Systems, 26*(1), 107-125.

Biermann, P. (2001). *Interaktives VR-System zur halbautomatischen Generierung von Wissen über Verbindungsmerkmale CAD-basierter Bauteil-Modelle.* Master Thesis, University of Bielefeld, Bielefeld.

Biermann, P., & Jung, B. (2004). *Variant Design in Immersive Virtual Reality: A Markup Language for Scalable CSG Parts.* Proceedings of Articulated Motion and Deformable Objects (AMDO-2004), pp. 123-133, Palma de Mallorca, Spain.

Biermann, P., Jung, B., Latoschik, M., & Wachsmuth, I. (2002, June). *Virtuelle Werkstatt: A Platform for Multimodal Assembly in VR.* Proceedings of Fourth Virtual Reality International Conference (VRIC 2002), pp. 53-62., Laval, France.

Biermann, P., & Wachsmuth, I. (2003, May). *An Implemented Approach for a Visual Programming Environment in VR.* Proceedings of Fifth Virtual Reality International Conference (VRIC 2003), pp. 229-234, Laval, France.

Biermann, P., & Wachsmuth, I. (2004). *Non-Physical Simulation of Gears and Modifiable Connections in Virtual Reality.* Proceedings of Sixth Virtual Reality International Conference (VRIC 2004). Laval, France.

Borning, A., Freeman-Benson, B., & Wilson, M. (1992). Constraint Hierarchies. *Lisp and Symbolic Computation, 5*(3), 223-270.

Brooks, F. P. (1987). No Silver Bullet - Essence and Accident in Software Engineering. *The Computer Journal, 20*(4), 10-19.

Bryson, S. (1996). Virtual Reality in Scientific Visualization. *Communications of the ACM, 39*(5), 62-71.

Burnett, M. M. (1999). Visual Programming. In J. G. Webster (Ed.), *Encyclopedia of Electrical and Electronics Engineering*. New York: John Wiley & Sons Inc.

Burnett, M. M., Baker, M. J., Bohus, C., Carlson, P., Yang, S., & Zee, P. v. (1995). Scaling up Visual Programming Languages. *IEEE Computer, 28*(3), 45-54.

Burnett, M. M., Cook, C. R., & Rothermel, G. (2004). End-User Software Engineering. *Communications of the ACM, 47*(9), 53-58.

Burnett, M. M., Sheretov, A., Ren, B., & Rothermel, G. (2002). Testing Homogeneous Spreadsheet Grids with the "What You See Is What You Test" Methodology. *IEEE Trans. Software Eng., 28*(6), 576-594.

Carey, R., & Bell, G. (1997). *The Annotated VRML 2.0 Reference Manual*: Addison-Wesley.

Cox, P., Giles, F., & Pietrzykowski, T. (1989). *Prograph: A Step Towards Liberating Programming from Textual Conditioning.* Proceedings of IEEE Workshop on Visual Languages., pp. 150-156.

Cruz-Nera, C., Sandin, D., & DeFanti, T. (1993). *Surround-Screen Projection-Based Virtual Reality; The Design and Implementation of the CAVE.* Proceedings of Computer Graphics SIGGRAPH.

Daniel, D. H. (1991). Data Flow Visual Programming Languages. *Journal of Visual Languages and Computing, 2*(1), 1-32.

Davis, A. L., & Keller, R. M. (1982). Data Flow Program Graphs. *IEEE Computer, 15*(2), 26-41.

Davis, T., Neider, J., Shreiner, D., & Woo, M. (1999). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2.*: Addison-Wesley.

Dennis, J. B. (1974). *First Version of a Data Flow Procedure Language.* Proceedings of Symposium on Programming, pp. 241- 271., Paris, France.

Dennis, J. B., & Misunas, D. P. (1975). *A Preliminary Architecture for a Basic Data-Flow Processor.* Proceedings of Second Annual Symposium on Computer Architecture.

Deutsch Limit. *Wikipedia, The Free Encyclopedia* Retrieved August, 2006, from http://en.wikipedia.org/wiki/Deutsch_Limit

Döllner, J., & Hinrichs, K. (2000). *A Generalized Scene Graph API.* Proceedings of Vision, Modeling, Visualization 2000 (VMV 2000), pp. 247-254, Saarbrücken.

Dong, J., & Yang, S. (2003, October). *Visualizing Design Patterns with a Uml Profile.* Proceedings of IEEE Symposium on Visual/Multimedia Languages, Auckland, New Zealand.

Edelson, D. R., & Pohl, I. (1991, April). *Smart Pointers: They're Smart but They're Not Pointers.* Proceedings of Usenix C++ Conference.

Eriksson, H.-E., & Penker, M. (1998). *UML Toolkit*. New York: John Wiley & Sons.

Ermel, C., Hoelscher, K., Kuske, S., & Ziemann, P. (2005, September). *Animated Simulation of Integrated Uml Behavioral Models Based on Graph Transformation*. Proceedings of 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC'05), Dallas, Texas/USA.

Erwig, M. (1998). Abstract Syntax and Semantics of Visual Languages. *Journal of Visual Languages and Computing, 9*, 461-483.

Fernando, R., & Kilgard, M. J. (2003). *The Cg Tutorial. The Definitive Guide to Programmable Real-Time Graphics*: Addison-Wesley.

Fernando, T., Marcelino, L., Wimalaratne, P., & Tan, K. (2000, February). *Interactive Assembly Modelling within a CAVE Environment.* Proceedings of 9th EUROGRAPHICS Portuguese Chapter, pp. 43-49, Marinha Grande, Portugal.

Foulser, D. (1995). IRIS Explorer: A Framework for Investigation. *Computer Graphics, 29*(2), 13-16.

Geiger, C., Mueller, W., & Rosenbach, W. (1998). Sam - An Animated 3d Programming Language. *1998 IEEE Symposium on Visual Languages*.

Gelernter, D., & Carriero, N. (1992). Coordination Languages and their Significance. *Communications of the ACM, 35*(2), 97-107.

Goldfeather, J., Molnar, S., Turk, G., & Fuchs, H. (1989). Near Realtime CSG Rendering Using Tree Normalization and Geometric Pruning. *IEEE Computer Graphics and Applications, 9*(3), 20-28.

Golin, J. E., Reiss, & S, P. (1990). *The Specification of Visual Language Syntax.* Proceedings of IEEE Workshop on Visual Languages, pp. 105-110.

Green, T. R. G., & Petre, M. (1992). *When Visual Programs Are Harder to Read Than Textual Programs.* Proceedings of Sixth European Conference on Cognitive Ergonomics, pp. 167-180.

Green, T. R. G., & Petre, M. (1996). Usability Analysis of Visual Programming Environments: A `Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing., 7*(2), 131-174.

Grimsdale, G. (1991). *dVS - Distributed Virtual Environment System.* Proceedings of Computer Graphics (SIGGRAPH '91), London, UK.

Gurd, J. R., & Bohm, W. (1987, Aug.). *Implicit Parallel Processing: SISAL on the Manchester Dataflow Computer.* Proceedings of IBM-Europe Institute on Parallel Processing, Oberlech, Austria.

Heumer, G., Schilling, M., & Latoschik, M. E. (2005, March). *Automatic Data Exchange and Synchronization for Knowledge-Based Intelligent Virtual Environments.* Proceedings of IEEE VR2005, pp. 43-50, Bonn, Germany.

Hudson, S. E. (1991). Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update. *ACM Transactions on Programming Languages and Systems (TOPLAS), 13*(3), 315-341.

Hudson, T., Lin, M., Cohen, J., Gottschalk, S., & Manocha, D. (1997). *Vcollide: Accelerated Collision Detection for VRML.* Proceedings of VRML Conference, pp. 119-125.

Humphreys, G., Houston, M., Ng, Y., Frank, R., Ahern, S., Kirchner, P., et al. (2002). *Chromium: A Stream Processing Framework for Interactive Graphics on Clusters.* Proceedings of Computer Graphics (SIGGRAPH), San Antonio, Texas.

Hundhausen, C., & Brown, J. L. (2005, September). *What You See Is What You Code: A Radically Dynamic Algorithm Visualization Development Model for Novice*

Learners. Proceedings of 2005 IEEE Symposium on Visual Languages, Dallas, TX.

IBM Inc. (1998). VisualAge™ for Java. from http://www.software.ibm.com/ad/vajava/

Ibrahim, B., & Yoshizumi, H. (1999, September). *Solving the Spaghetti Plate Syndrome in a Control-Flow Language with a Vlsi-Like Solution.* Proceedings of 1999 IEEE Symposium on Visual Languages (VL'99), Tokyo, Japan.

Jagannathan, R. (1995a). *Coarse-Grain Dataflow Programming of Conventional Parallel Computers.* Proceedings of Advanced Topics in Dataflow Computing and Multithreading., pp. 113-129., Los Alamitos, CA.

Jagannathan, R. (1995b). Dataflow Models. In E. Y. Zomaya (Ed.), *Parallel and Distributed Computing Handbook*: McGrawHill.

Johnston, W., Hanna, J. R. P., & Millar, R. J. (2004). Advances in Dataflow Programming Languages. *ACM Computing Surveys, 36*(1), 1-34.

Jonathan Corney, & Lin., T. (2002). *3D Modeling with ACIS.*: Paul & Co Pub Consortium.

Jung, B., Latoschik, M., Biermann, P., & Wachsmuth, I. (2002). *Virtuelle Werkstatt.* Proceedings of 1. Paderborner Workshop Augmented Reality / Virtual Reality in der Produktentstehung, pp. 185-196, Paderborn: HNI.

Kahn, K. M., Saraswat, V. A., & Haarslev, V. (1991, Oktober). *Pictorial Janus: A Completely Visual Programming Language and its Environment.* Proceedings of GI-Fachgesprach Programmieren multimedialer Anwendungen der GI-Jahrestagung 1991, pp. 427-436, Darmstadt.

KDevelop Team. KDevelop. Retrieved July, 2006, from www.kdevelop.org

Kimura, T., & McLain, P. (1986). *Show and Tell User's Manual* (Technical Report No. WUCS-86-4.). St Louis, MO.

Kopp, S. (1998). *Ein wissensbasierter Ansatz zur Modellierung von Verbindungen für die virtuelle Montage.* Master Thesis, University of Bielefeld, Bielefeld.

Kopp, S., Jung, B., Lessmann, N., & Wachsmuth, I. (2003). Max - a Multimodal Assistant in Virtual Reality Construction. *KI-Künstliche Intelligenz, 3*(4), 11-17.

Kopp, S., Sowa, T., & Wachsmuth, I. (2004). *Imitation Games with an Artificial Agent: From Mimicking to Understanding Shape-Related Iconic Gestures.* Proceedings of International Gesture Workshop 2003, Gesture-based communication in human-computer interaction, pp. 436-447, Genua, Italy.

Kranstedt, A., & Wachsmuth, I. (2005, August). *Incremental Generation of Multimodal Deixis Referring to Objects.* Proceedings of 10th European Workshop on Natural Language Generation (ENLG 2005), pp. 75-82, Aberdeen, UK.

Kreylos, O., Bethel, W., Ligocki, T. J., & Hamann, B. (2001). *Virtual-Reality-Based Interactive Exploration of Mutiresolution Data*. Heidelberg: Springer Verlag.

Kruglinski, D. J. (1997). *Inside Visual C++* (Vol. fourth edition). Redmond, Washington: Microsoft Press.

Kuester, J. M., Heckel, R., & Engels, G. (2003, October). *Defining and Validating Transformations of Uml Models.* Proceedings of 2003 IEEE Symposium on Visual Languages and Formal Methods, Auckland, New Zealand.

Labview. (2000). *Lab View User Manual*. Austin, TX.: National Instruments.

Latoschik, M. E. (2001a, November). *A Gesture Processing Framework for Multimodal Interaction in Virtual Reality.* Proceedings of 1st International Conference on Computer Graphics, Virtual Reality and Visualization in Africa (Afrigraph 2001), pp. 95-100.

Latoschik, M. E. (2001b). *Multimodale Interaktion in Virtueller Realität am Beispiel der Virtuellen Konstruktion.* PhD. Dissertation, University of Bielefeld, Bielefeld.

Latoschik, M. E., Biermann, P., & Wachsmuth, I. (2005a, March). *High-Level Semantics Representation for Intelligent Simulative Environments.* Proceedings of IEEE VR2005, pp. 283-284, Bonn, Germany.

Latoschik, M. E., Biermann, P., & Wachsmuth, I. (2005b). *Knowledge in the Loop: Semantics Representation for Multimodal Simulative Environments.* Proceedings of 5th International Symposium on Smart Graphics 2005, pp. 25-39.

Latoschik, M. E., Fröhlich, M., Jung, B., & Wachsmuth, I. (1998). *Utilize Speech and Gestures to Realize Natural Interaction in a Virtual Environment.* Proceedings of IECON'98 - Proceedings of the 24th Annual Conference of the IEEE Industrial Electronics Society Vol. 4, IEEE, pp. 2028-2033.

Leßmann, N., Kranstedt, A., & Wachsmuth, I. (2004). *Towards a Cognitively Motivated Processing of Turn-Taking Signals for the Embodied Conversational Agent Max.* Proceedings of AAMAS '04, Workshop Embodied Conversational Agents: Balanced Perception and Action, pp. 57-64, New York.

Light, R., & Gossard, D. (1982). Modification of Geometric Models through Variational Geometry. *Computer-Aided Design (CAD), 14*(4), 209-214.

Lin, V. C., Gossard, D. C., & Light, R. A. (1981). Variational Geometry in Computer Aided Design. *ACM Computer Graphics (SIGGRAPH'81), 15*(3), 171-175.

Mencer, O., Hübert, H., Morf, M., & Flynn, M. J. (2000). StReAm: Object-Oriented Programming of Stream Architectures using PAM-Blox. *Field-Programmable Logic and Applications*, 595-604.

Mirtich, B., & Canny, J. (1995, April). *Impulse-Based Simulation of Rigid Bodies.* Proceedings of Workshop on Interactive 3D Graphics, pp. 181-188.

Musser, D. R., & Saini, A. (1996). *STL Tutorial and Reference Guide*: Addison-Wesley.

Myers, B. (1990). Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing, 1*(1), 97-123.

Najork, M. A. (1996). Programming in Three Dimensions. *Journal of Visual Languages and Computation, 7*, 219-242.

Najork, M. A., & Kaplan, S. M. (1991, October). *The Cube Language.* Proceedings of 1991 IEEE Workshop on Visual Languages, pp. 218-224, Kobe, Japan.

Object Technology International Inc. (2001, July). Eclipse Platform - A Universal Tool Platform. 2006, from http://www.eclipse.org/

Okamura, T., Shizuki, B., & Tanaka, J. (2004). *Execution Visualization and Debugging in Three-Dimensional Visual Programming.* Proceedings of Iv 2004, pp. 167-172.

Oshiba, T., & Tanaka, J. (1999). "*3d-Pp*": *Three-Dimensional Visual Programming System.* Proceedings of VL 1999, pp. 189-190.

Papadopoulos, G. M. (1988). *Implementation of a General Purpose Dataflow Multiprocessor* (Technical Report No. TR432.). Cambridge, MA.: Laboratory for Computer Science MIT.

Pfeiffer, T., Kranstedt, A., & Lücking, A. (2006). *Sprach-Gestik Experimente mit IADE, dem Interactive Augmented Data Explorer.* Proceedings of Dritter Workshop Virtuelle und Erweiterte Realität der GI-Fachgruppe VR/AR, Koblenz, (To Appear).

Reitmayr, G., & Schmalstieg, D. (2001). *Opentracker-an Open Software Architecture for Reconfigurable Tracking Based on Xml.* Proceedings of IEEE VR 2001, pp. 285-286.

Ritter, H. The Graphical Simulation Toolkit Neo/NST. Retrieved July, 2006, from http://www.techfak.uni-bielefeld.de/ags/ni/projects/neo/neo_e.html

Roth, K. (1994). *Konstruieren mit Konstruktionskatalogen* (2 ed. Vol. 1). Berlin: Springer Verlag.

Rothermel, G., Li, L., DuPuis, C., & Burnett, M. (1998, April). *What You See Is What You Test: A Methodology for Testing Form-Based Visual Programs.* Proceedings of 1998 International Conference on Software Engineering, pp. 198-207, Kyoto, Japan.

Sannella, M. (1993). *The SkyBlue Constraint Solver and Its Applications.* Proceedings of First Principles and Practice of Constraint Programming Workshop (PPCP'93), Newport, RI.

Sannella, M., Maloney, J., Freeman-Benson, B., & Borning, A. (1993). Multi-Way Versus One-Way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. *Software - Practice and Experience, 23*(5), 529-566.

Serrano, D., & Gossard, D. (1992). Tools and Techniques for Conceptual Design. *Artificial Intelligence in Engineering Design, 1*, 71-116.

Sherman, W. (1993, April). *Integrating Virtual Environments into the Dataflow Paradigm.* Proceedings of Fourth Eurographics Workshop on ViSC, Abington, UK.

Shizuki, B., Shibayama, E., & Toyoda, M. (2002). *Static Visualization of Dynamic Data Flow Visual Program Execution.* Proceedings of Iv 2002, pp. 713-718.

Silc, J., Robic, B., & Ungerer, T. (1998). Asynchrony in Parallel Computing: from Dataflow to Multithreading. *Parallel Distrib. Comput. Pract., 1*(1), 3-30.

Silicon Graphics Inc. (2002). *OpenGL Performer Programmer's Guide.* CA: Mountain View.

Sowa, T. (2006). *Understanding Coverbal Iconic Gestures in Shape Descriptions.* PhD. Dissertation, University of Bielefeld, Bielefeld.

Steed, A., & Slater, M. (1996). *A Dataflow Representation for Defining Interaction within Immersive Virtual Environments.* Proceedings of IEEE Virtual Reality Annual International Symposium 96, pp. 163-167.

Sterling, T., Kuehn, J., Thistle, M., & Anastasis, T. (1995). *Studies on Optimal Task Granularity and Random Mapping.* Proceedings of Advanced Topics in Dataflow Computing and Multithreading., pp. 349- 365., Los Alamitos, CA.

Stiles, R., & Pontecorvo, M. (1992, September). *Lingua Graphica: A Visual Language for Virtual Environments.* Proceedings of IEEE Workshop on Visual Languages, pp. 225 - 227, Seattle, WA.

Störrle, H. (2004). *Semantics of Control-Flow in Uml 2.0 Activities.* Proceedings of 2004 IEEE Symposium on Visual Languages and Human-Centric Computing, Rome, Italy, September.

Strauss, P., & Carey, R. (1992). *An Object-Oriented 3D Graphics Toolkit.* Proceedings of Computer Graphics (SIGGRAPH '92), pp. 341-449.

Strauss, P. S. (1993, September). *IRIS Inventor, A 3D Graphics Toolkit.* Proceedings of 8th Annual Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 192-200, Washington, DC, USA.

Tramberend, H. (1999). *A Distributed Virtual Reality Framework.* Proceedings of IEEE Virtual Reality 1999, pp. 14-21.

Turner, J. U., Subramanian, S., & Gupta, T. (1992). Constraint Representation and Reduction in Assembly Modeling and Analysis. *IEEE Transactions on Robotics and Automation, 8*(6), 707-722.

Ueda, K. (1985). *Guarded Horn Clauses* (ICOT Technical Report No. TR-103).

Verdoscia, L., & Vaccaro, R. (1998). A High-Level Dataflow System. *The Computer Journal, 60*(4), 285-305.

Vose, G. M., & Williams, G. (1986). *Labview: Laboratory Virtual Instrument Engineering Workbench*: Byte.

Wachsmuth, I. (1998). Experten- und Agentensystemtechniken für intuitivere Benutzungsschnittstellen. In J. Mester & J. Perl (Eds.), *Informatik im Sport: Bericht über das internationale Symposium 12.-14. Juni 1997 in Köln* (pp. 181-191). Köln: Sport und Buch Strauss.

Wachsmuth, I., & Jung, B. (1996). Dynamic Conceptualization in a Mechanical-Object Assembly Environment. *Artificial Intelligence Review, 10*(3-4), 345-368.

Wadge, W., & Ashcroft, E. A. (1985). Lucid, the Dataflow Programming Language. *APIC Studies in Data Processing*.

Weng, K. S. (1975). *Stream Oriented Computation in Recursive Data-Flow Schemas* (Technical Report No. 68). Cambridge, MA.: Laboratory for Computer Science MIT.

Wernecke, J. (1994). *The Inventor Mentor*: Addison-Wesley.

Wharton, S. M., & Singh, Y. P. (2001). Development of Solid Models and Multimedia Presentations of Kinematic Pairs. *American Society for Engineering Education*.

Wilcox, E. M., Atwood, J. W., Burnett, M. M., Cadiz, J. J., & Cook, C. R. (1997). *Does Continuous Visual Feedback Aid Debugging in Direct-Manipulation Programming Systems?* Proceedings of Chi 1997, pp. 258-265.

X3D Working Group. (2002). Extensible 3D International Draft Standards, iso/iec fcd 19775:200x., 2006, from http://www.web3d.org/technicalinfo/specifications/ISO_IEC_19775

Zachmann, G. (1998, March). *Rapid Collision Detection by Dynamically Aligned DOP-Trees.* Proceedings of IEEE Virtual Reality Annual International Symposium; VRAIS '98, pp. 90 - 97, Atlanta, Georgia.

Zachmann, G. (2000). *Virtual Reality in Assembly Simulation & Collision Detection, Simulation Algorithms Interaction Techniques.* PhD Dissertation, University of Technology, Darmstadt, Germany.

Zanden, B. (1988). *An Incremental Planning Algorithm for Ordering Equations in a Multilinear System of Constraints.* PhD thesis, Cornell University.

Zanden, B. (1996). An Incremental Algorithm for Satisfying Hierarchies of Multi-Way Dataflow Constraints. *ACM Transactions on Programming Languages and Systems (TOPLAS), 18*(1), 30-72.

Zhang, D.-Q., Zhang, K., & Cao, J. (2001). A Context-Sensitive Graph Grammar Formalism for the Specification of Visual Languages. *The Computer Journal, 44*(3), 186-200.