
i>PM 3D – Ein Prozessmodellierungswerkzeug für drei Dimensionen

Repräsentation von Prozessmodellen im dreidimensionalen Raum – Konzept und Implementierung

Tobias Stenzel

12. 04. 2012

Inhaltsverzeichnis

1 Einleitung	1
1.1 Motivation für die Modellierung von Prozessen in 3D	1
1.2 Zielsetzung und Aufbau dieser Arbeit	2
1.2.1 Visualisierung	2
1.2.2 Anpassbarkeit durch Metamodellierung	3
1.2.3 Modellanbindung	3
1.2.4 Rendering	3
1.3 Funktionale Anforderungen	4
2 Grundlagen	5
2.1 Prozessmodellierungssprachen	5
2.1.1 Visuelle Sprachen und deren Klassifikation	5
2.1.2 Perspektivenorientierte Prozessmodellierung	6
2.1.3 Grafische Modellierungswerkzeuge	7
2.2 Metamodellierung	8
2.2.1 Linguistic Meta Model	9
2.2.2 Model Designer Framework	12
2.3 Typ-Verwendungs-Konzept	13
3 Verwandte Arbeiten zur 3D-Visualisierung	15
3.1 3D-Softwarevisualisierung	15
3.1.1 Wilma: Ein 3D-Modellierungstool für UML-Diagramme	15
3.1.2 X3D-UML: Hierarchische 3D-UML-Zustandsdiagramme	17
3.1.3 3D-Visualisierung von großen, hierarchischen UML-Zustandsdiagrammen	18
3.1.4 Dreidimensionale Darstellung zur besseren Visualisierung von Beziehungen	19
3.1.5 Nutzung von Tiefeneindruck und Animation zur Visualisierung von UML-Diagrammen	19
3.1.6 Graphical Editing Framework 3D	21
3.2 3D-Prozessvisualisierung	22
3.2.1 3D-Repräsentation von Prozessmodellen	22
3.2.2 3D-Visualisierung für die Prozesssimulation	24
3.2.3 Modellierung von Prozessen in interaktiven, virtuellen 3D-Umgebungen	24
3.3 Verbesserung der dreidimensionalen Darstellung von Graphen	27
3.3.1 Nutzung von 3D-Effekten für einen verbesserten Tiefeneindruck	27

3.3.2	3D-Visualisierung von Graphen in voll immersiven virtuellen Umgebungen	27
3.4	Zusammenfassung und Bewertung	28
3.4.1	2,5D / 3D-Visualisierungen von Modellen	29
3.4.2	Integration von weiteren Informationen in 3D-Visualisierungen	30
3.4.3	Effizienz und Akzeptanz von 3D-Darstellungen?	31
3.4.4	Verwendbare Vorarbeiten und Schlussfolgerungen für die Arbeit	32
4	Verwendete Techniken und Software	33
4.1	Scala	33
4.1.1	Traits	33
4.1.2	Objects	34
4.1.3	Actors	34
4.1.4	Implizite Methoden	34
4.1.5	Parser-Kombinatoren	35
4.2	Simulator X	35
4.3	OpenGL / LWJGL	36
4.4	Sonstiges	37
4.4.1	StringTemplate	37
4.4.2	Simplex3D-Math	38
5	Einordnung in das Gesamtprojekt i>PM3D	39
5.1	Übersicht über die Projektbestandteile	39
5.1.1	Visualisierung	40
5.1.2	Modellanbindung	40
5.1.3	GUI	40
5.1.4	Eingabeauffbereitung und Editor	41
5.1.5	Neuartige Eingabegeräte	41
5.2	i>PM3D als Simulator X - Applikation	41
5.2.1	Modifikationen an Simulator X	41
5.2.2	Modellkomponente und Modell-Entitäten	42
5.2.3	Editor-Komponente und Eingabekonnektoren	42
6	Modellhierarchie	43
6.1	LMMLight	43
6.2	Editor-Model-Stack	44
6.2.1	Anpassbarkeit	44
6.2.2	Übersicht über die Editor-Model-Ebenen	45
6.3	Domain-Model-Stack	46
6.3.1	Domain-Meta-Model	47
6.3.2	Domain-Model	47
7	Spezifikation der Metamodelle	49
7.1	Scala-Mapping	49
7.2	Editor-Base-Level	49
7.2.1	Paket „types“	50
7.2.2	Paket „figures“	50
7.3	Editor-Definition-Level	53
7.4	Prozess-Metamodell	53
7.5	Anwendungsbeispiel: Hinzufügen eines neuen Modellelements	54
7.5.1	Änderungen am Prozess-Metamodell	54
7.5.2	Änderungen am Editor-Metamodell	55

8 3D-Visualisierung von Prozessen	57
8.1 Grundlegende Darstellung der grafischen Elemente	57
8.1.1 Knoten	57
8.1.2 Kanten	59
8.1.3 Szenenobjekte	60
8.2 Visualisierungsvarianten für interaktive Modelleditoren	60
8.2.1 Hervorhebung	61
8.2.2 Selektion	61
8.2.3 Deaktivierung	62
8.3 2D-Modellierungsflächen	62
8.4 Beleuchtung	62
8.5 Visualisierung eines Beispielsprozesses	64
8.6 Umsetzung von verschiedenen Nutzungsmöglichkeiten der dritten Dimension	65
8.6.1 Visualisierung von Attributen	65
8.6.2 Zeitliche Abläufe	65
8.6.3 Veranschaulichung von Beziehungen	66
8.6.4 Hierarchische Darstellung und mehrere Modelle	66
8.6.5 Einbettung in eine virtuelle Umgebung	68
8.7 Weitere Entwicklungsmöglichkeiten	68
8.7.1 Darstellung von Text	68
8.7.2 Konfigurierbarkeit	69
8.7.3 Räumliche Darstellung	69
8.7.4 Darstellung von Kanten	70
9 Modellanbindung	73
9.1 Modellkomponente	74
9.1.1 Commands	74
9.1.2 Übersicht	74
9.2 Modell-Persistenz	74
9.2.1 Speicherrepräsentation eines LMMLight-Modells	75
9.2.2 Vereinfachung des Umgangs mit Modellen	75
9.2.3 Laden von Metamodellen	77
9.2.4 Laden und Schließen von und Umgang mit mehreren Modellen	77
9.2.5 Speichern von Modellen	78
9.3 Modell-Entitäten	78
9.3.1 Aspekte	79
9.3.2 Setzen und Auslesen von Position, Ausrichtung und Größe	80
9.3.3 Modell-SVars	80
9.4 Übersicht über den Lebenszyklus von Model-Entitäten	82
10 Renderkomponente	85
10.1 RenderActor	85
10.2 OpenGL-Versionsproblematik	85
10.3 Projektspezifische Erweiterungen	86
10.3.1 ShapeFromFactory	86
10.3.2 Modellierungsflächen	86
10.3.3 2D-Grafikobjekte	86
10.3.4 User-Entity	86
11 Render-Bibliothek	87
11.1 Übersicht	88
11.2 Low-Level-API	89

11.3	Higher-Level-API	89
11.3.1	Drawable	89
11.3.2	RenderStage	91
11.3.3	Weitere Abstraktionen	92
11.4	COLLADA2Scala-Compiler	92
11.5	Spezielle Erweiterungen für i>PM3D	93
11.5.1	Unterstützung für deaktivierte, hevorgehobene und selektierte Elemente	93
11.5.2	Darstellung von Text	94
11.5.3	SVarSupport - Einbindung der Modell-Drawables in i>PM3D	94
11.6	Anwendungsbeispiel: Erstellen von neuen Modell-Figuren	96
11.6.1	Anmerkungen	96
12	Fazit und Ausblick	97
Literaturverzeichnis		99
Literatur		99
WWW-Referenzen		103
A	Verwendete Metamodelle	105
A.1	Editor-Metamodell	105
A.1.1	Programming-Language-Mapping	105
A.1.2	Editor-Base-Level	105
A.1.3	Editor-Definition-Level (Auszug)	108
A.2	Prozess-Metamodell	109
B	Systemanforderungen und Inhalt der DVD	111
B.1	Systemanforderungen von i>PM3D	111
B.1.1	Hardware	111
B.1.2	Software	111
B.2	DVD	111
B.2.1	Ausführbare Dateien	111
B.2.2	Eclipse-Projekte	112
B.2.3	Beilagen	112

Abbildungsverzeichnis

2.1	Graph- und geometriebasierter Ansatz im Vergleich	6
2.2	Perspektivenorientierte Prozessmodellierung aus [Rot11]	7
2.3	Prozessmodellierungswerkzeug i>PM2 aus [Rot11]	8
2.4	Hierarchie der LMM-Elemente aus [Vol11]	9
2.5	Vergleich von objektorientierter Modellierung (links) und Metamodellierung mit Clabjects	10
2.6	Instanz-Spezialisierung ausgehend von ConceptD	11
2.7	Modellhierarchie von MDF mit Domain-Model- und Designer-Stack aus [Rot11]	12
2.8	Prozess in i>PM2 aus [Vol11] (Bezeichner A hinzugefügt)	13
2.9	Prozess mit angepasster Verwendung aus [Vol11] (B hinzugefügt)	14
3.1	3D-UML-Klassendiagramm aus [Dwy01]	16
3.2	Hierarchisch aufgebautes 3D-UML-Zustandsdiagramm aus [MHS08]	17
3.3	3D-Zustandsdiagramm aus [KN09]	18
3.4	Zustandsdiagramm mit ausgeblendeten Diagrammteilen (dargestellt durch blaue Würfel) aus [KN09]	18
3.5	3D-UML-Sequenzdiagramm; Ausschnitt aus [GK98]	19
3.6	3D-UML-Klassendiagramm aus [GRR99]	20
3.7	Diagramm mit nach hinten verschobenen, „unwichtigen“ Klassen aus [GRR99]	20
3.8	Kombination mehrerer 2D-Editoren in einer 3D-Ansicht von [www:gef3d]	21
3.9	3D-Ecore-Editor von [www:gef3ddevblog]	22
3.10	Darstellung von Beziehungen zwischen Prozess- und Organisationsmodell aus [Bet+08]	23
3.11	Visualisierung von Ähnlichkeiten zwischen Prozessmodellen aus [Bet+08]	23
3.12	Vier Verfeinerungsstufen eines Prozessmodells aus [Bet+08]	24
3.13	Prozessgraph mit „Datenflusskugeln“ aus [SBE00]	25
3.14	Darstellung eines Prozesses mit assoziierten Daten (3D-Histogramme) aus [SBE00]	25
3.15	BPMN-Prozessgraph in virtueller Welt aus [Bro10]	26
3.16	Benutzer-Avatar vor 3D-BPMN-Elementen aus [Bro10]	26
3.17	Kommentarwände und Multimedia-Inhalte in der virtuellen Welt aus [Bro10]	27
3.18	Visualisierung von semantischen Netzwerken aus [Hal+08]	28
3.19	Visualisierung eines BPMN-Prozesses in einer virtuellen Umgebung	31
4.1	Zustandsvariablen-Konzept aus [LT11]	36

5.1	Übersicht über die Bestandteile von i>PM3D	39
5.2	Architektur von i>PM3D, aufbauend auf Simulator X	42
6.1	Modellhierarchie in i>PM3D (angelehnt an MDF [Rot11])	44
6.2	Assoziation zwischen abstraktem Modellelement und konkreter Repräsentation	45
7.1	Hierarchie des <code>figures</code> -Pakets	51
7.2	Perspektiven-Hierarchie im Prozess-Meta-Modell	53
7.3	Die Kanten ControlFlow, NodeDataConnection und deren Assoziationen	54
8.1	Zwei Prozessknoten; links im Ursprungszustand, rechts als angepasste Verwendung (Screenshot aus i>PM3D)	58
8.2	AND-Connector (Screenshot aus i>PM3D)	58
8.3	Schriftdarstellung bei direkter Sicht auf eine Ecke (links) und bei günstigerer Perspektive (Screenshot aus i>PM3D)	59
8.4	Gerichtete Kontrollflusskante von „Backen“ nach „Verpacken“ (Screenshot aus i>PM3D) . .	60
8.5	Datenknoten, normal (links) und hervorgehoben (Screenshot aus i>PM3D)	61
8.6	Entscheidungsknoten und Prozess im selektierten Zustand (Screenshot aus i>PM3D) . . .	61
8.7	Deaktivierter (vorne, durchsichtig) und aktiver Prozess (Screenshot aus i>PM3D)	62
8.8	Zwei leere Modellierungsflächen (Screenshot aus i>PM3D)	63
8.9	Beispiel für einen Prozess mit deaktivierten Datenfluss-Elementen (Screenshot aus i>PM3D)	64
8.10	Darstellung von zwei Prozessen, Fokus auf gelbem Modell (Screenshot aus i>PM3D) . . .	67
8.11	Darstellung von zwei Prozessen, Fokus auf blauem Modell (Screenshot aus i>PM3D) . .	67
8.12	Visualisierung von Beziehungen mit gekrümmten 3D-Röhren aus [BD04]	71
9.1	Die Modellanbindung im Kontext von i>PM3D	73
9.2	Unterkomponenten der <code>ModelComponent</code> (vereinfacht)	75
9.3	Speicherrepräsentation eines Beispiel-Prozessmodells (Ausschnitt)	76
9.4	Sequenzdiagramm <code>LoadMetaModels</code> (vereinfacht).	78
9.5	<code>EntityDescription</code> für einen Knoten (nur ausgewählte und vereinfachte Attribute)	79
9.6	Ablauf der Erstellung einer <code>ModelEntity</code> durch die <code>Editorkomponente</code>	83
11.1	Zusammensetzung eines farbigen Würfels aus den Basis-Traits	90
11.2	Zusammengesetzter <code>PhongMaterialEffect</code>	91
11.3	<code>RenderStage</code> mit eingemischten Plugin-Traits	92
11.4	Ablauf bei Änderung der Hintergrundfarbe eines Prozesses durch den Benutzer	95

Einleitung

In der Computergrafik zeigte sich in den letzten Jahrzehnten eine bemerkenswerte Entwicklung der Möglichkeiten von Hardware und Software. Interaktive 3D-Grafikanwendungen, die früher nur mit sehr großem Aufwand und finanziellem Einsatz realisierbar waren, sind heute für handelsübliche PCs verfügbar. Besonders deutlich wird diese Entwicklung im Bereich der Computerspiele, die es heute dem Spieler erlauben, in eine nahezu fotorealistische Darstellung der realen Welt einzutauchen und mit 3D-Objekten zu interagieren [Por04].

Neben den seit Jahrzehnten verbreiteten Eingabegeräten Tastatur und Maus lassen sich Spiele mittlerweile auch mit neuartigen Eingabegeräten – wie der Nintendo Wii oder Microsoft Kinect – bedienen [Sun11]. So lassen sich Bewegungen des Benutzers direkt in den dreidimensionalen Raum übertragen, wodurch eine natürliche Interaktion mit der virtuellen Welt ermöglicht wird.

Auch in „ernsthaften“ Anwendungen werden die Möglichkeiten der 3D-Computergrafik genutzt. Beispielsweise werden in der Bioinformatik Werkzeuge wie *PyMol* [Sch10] eingesetzt, welche eine Visualisierung von dreidimensionalen Strukturen ermöglichen, welche entscheidend für die Funktion von Biomolekülen sind. Durch Interaktion per Tastatur und Maus lässt sich das Molekül aus verschiedenen Perspektiven betrachten. Das vollständige Eintauchen in eine 3D-Szene und die Interaktion durch Körperbewegungen oder auch Sprache wird von Werkzeugen zur virtuellen Konstruktion gezeigt („virtuelle Werkstatt“) [FLW09]. So lässt sich beispielsweise ein rein virtuelles Auto in einer realistischen Darstellung betrachten und sogar „anfassen“. Solche Systeme bieten teilweise auch die Möglichkeit, dass mehrere Benutzer gleichzeitig mit dem System interagieren können. In den genannten Anwendungsfällen sind die Vorteile einer dreidimensionalen Darstellung offensichtlich, da hier Objekte aus der realen Welt abgebildet werden, die grundsätzlich dreidimensional ist.

1.1 Motivation für die Modellierung von Prozessen in 3D

Die vorliegende Arbeit beschäftigt sich jedoch vorrangig mit der Modellierung von Prozessen, welche die Aufgabe hat, (Geschäfts-)Abläufe und zugehörige Informationen in einer abstrahierten Form darzustellen. Hierbei ist es weniger leicht festzustellen, inwieweit eine dreidimensionale Visualisierung sinnvoll wäre und wie diese Darstellung überhaupt aussehen könnte.

Prozessmodelle dienen unter anderem der Kommunikation zwischen den an der Entwicklung oder Ausführung eines Prozesses beteiligten Personen („Stakeholder“), für welche die Darstellung leicht verständlich und informativ sein sollte [Bro10]. Prozessmodellierungswerkzeuge, wie beispielsweise *ARIS* (*Express*) [SN00] oder das später in dieser Arbeit gezeigte *i>PM²* [Rot11], nutzen bisher ausschließlich 2D-Darstellungen. Das Bedienkonzept dieser Anwendungen folgt den Standards der seit zwei bis drei

Jahrzehnten üblichen Desktopprogramme, welche eine zweidimensionale grafische Benutzeroberfläche anbieten und mit Tastatur und Maus bedient werden („WIMP“-Oberflächen¹).

Der Einsatz der dritten Dimension für die Repräsentation von Prozessen wurde jedoch schon vereinzelt untersucht. Beispielsweise wird von [Bet+08] gezeigt (Unterabschnitt 3.2.1), wie sich dies für die Visualisierung von Beziehungen zwischen mehreren Modellen oder zur Darstellung von hierarchischen Modellen sinnvoll nutzen lässt. So ergeben sich Vorteile zu einer 2D-Darstellung, welche unter anderem weniger Möglichkeiten bietet, verschiedene Arten von Beziehungen zwischen Modellelementen in leicht verständlicher Form zu visualisieren [GK98]. Arbeiten auf dem Gebiet der Softwaremodellierung, welche in dieser Arbeit vorgestellt werden, zeigen weitere Nutzungsmöglichkeiten, die sich auch auf die Prozessmodellierung übertragen lassen.

Prozessmodelle enthalten oft auch Konzepte, die Entitäten aus der realen Welt vertreten, beispielsweise die in einem Prozessschritt verwendete Maschine oder eine ausführende Person. Es kann sinnvoll sein, diese Objekte in ihrem realen Erscheinungsbild neben dem Prozessmodell darzustellen, um das abstrakte Modell für Benutzer anschaulicher zu machen oder weitere Informationen bereitzustellen, wie von [Bro10] vorgeschlagen wird (siehe Unterabschnitt 3.2.3 und Unterabschnitt 3.4.2).

Ein Prozessmodellierungswerkzeug, welches die Möglichkeiten der modernen 3D-Computergrafik ausnutzt oder gar neuartige (3D-)Eingabegeräte unterstützt, existiert bisher nicht [Bro10]. Um die Effizienz von 3D-Visualisierungen für die Prozessmodellierung zu beurteilen und verschiedene Darstellungsformen zu vergleichen wäre allerdings ein solches System vonnöten.

1.2 Zielsetzung und Aufbau dieser Arbeit

Da es kaum Möglichkeiten gibt, die Effizienz von 3D-Prozessvisualisierungen – besonders in interaktiven Anwendungen – zu evaluieren, wurde mit dem i>PM3D-Projekt ein Prototyp eines 3D-Prozessmodellierungswerkzeugs entwickelt, welches auch neuartige (3D-)Eingabegeräte nutzt und die Anbindung von weiteren Eingabemöglichkeiten einfach macht. Das Projekt basiert auf *Simulator X* (Abschnitt 4.2), einer Plattform für eine modulare, komponentenbasierte Realisierung von Anwendungen aus dem Bereich der 3D-Computergrafik. Ein detaillierter Überblick über das Gesamtprojekt wird später in [dieser Arbeit](#) (Kapitel 5) gegeben.

Die vorliegende Arbeit beschäftigt sich im Rahmen des Projekts mit der Konzeption und Realisierung der **Repräsentation** der Prozessmodelle im Modellierungswerkzeug. Repräsentation bezieht sich hier sowohl auf die Visualisierung der Prozessmodelle als auch auf die interne Darstellung der Modelle und deren physische Speicherung (auf Datenträgern).

1.2.1 Visualisierung

Da es kaum möglich war, auf schon vorhandene Implementierungsarbeiten zurückzugreifen, liegt der Fokus dieser Arbeit eher auf der Bereitstellung von technischen Grundlagen, die zur Realisierung einer flexiblen 3D-Prozessvisualisierung im Prototypen nötig waren. Dennoch werden in Kapitel 3 Arbeiten vorgestellt, die einen Überblick darüber geben sollen, wie die dritte Visualisierungsdimension genutzt werden kann und welche Vorteile sich aus 3D-Darstellungen ergeben.

Die Implementierung konzentriert sich nicht auf eine bestimmte Nutzungsmöglichkeit, sondern ist möglichst allgemein gehalten. So werden Modelle in i>PM3D als 3D-Graph dargestellt, dessen Knoten sich frei im Raum platzieren lassen. Der Benutzer selbst kann sich in der 3D-Szene bewegen und so den Graphen aus verschiedenen Perspektiven betrachten. Zusätzlich zu den Modellelementen können beliebige 3D-Objekte in die Szene eingefügt werden, um reale Objekte abzubilden. Inwieweit sich die

¹WIMP steht für „Windows, Icons, Menus, Pointer“. Grafische Benutzeroberflächen, die auf die Nutzung mit anderen Eingabegeräten als Tastatur und Maus ausgelegt sind, werden auch als „Post-WIMP-Interfaces“ bezeichnet. [Dam97]

vorgestellten Nutzungsmöglichkeiten mit dem Prototypen realisieren lassen und welche Erweiterungen dafür sinnvoll wären, wird in Kapitel 8 näher ausgeführt.

1.2.2 Anpassbarkeit durch Metamodellierung

Um die Anpassung der in einem Modell verwendeten Konstrukte zu ermöglichen – wie es für die Prozessmodellierung sinnvoll ist ([siehe Abschnitt 2.2](#)) – werden abstrakte Syntax der Modellierungssprache und deren konkrete grafische Repräsentation in getrennten **Metamodellen** beschrieben, wie es schon durch das in [Rot11] entwickelte *Model Designer Framework* (Unterabschnitt 2.2.2) für 2D-Modelleeditoren umgesetzt wird. So lassen sich auch gänzlich neue Elemente und dazugehörige grafische Objekte hinzufügen. Ebenso macht dies ein Experimentieren mit der Visualisierung einfach. Eine Übersicht über die in i>PM3D verwendeten (Meta-)Modelle und deren Hierarchie wird in Kapitel 6 gegeben.

Prinzipiell lässt sich i>PM3D durch diese Anpassbarkeit nicht nur für die Modellierung von Prozessen, sondern auch für ähnliche Anwendungsdomänen einsetzen. Der Fokus liegt hier allerdings speziell auf der Modellierung nach dem Prinzip der *perspektivenorientierten Prozessmodellierung* (Unterabschnitt 2.1.2) und dem damit assoziierten *Typ-Verwendungs-Konzept* (Abschnitt 2.3). So wird jeweils ein Metamodell für diese Domäne und deren Visualisierung nach einem graphbasierten Ansatz bereitgestellt (Kapitel 7). Zusammen beschreiben diese Metamodelle einen **Prozessmodell-Editor**, der den Konzepten von vergleichbaren 2D-Modellierungswerkzeugen und der daraus bekannten Visualisierung folgt.

1.2.3 Modellanbindung

Für den Zugriff auf die interne Repräsentation der Modelle muss eine Schnittstelle bereitgestellt werden, über die andere Komponenten der Anwendung Parameter zur Laufzeit verändern können, welche die grafische Repräsentation oder das Prozessmodellelement selbst (bspw. die Funktion eines Prozessknotens) betreffen. Ebenfalls werden für ein Modellierungswerkzeug übliche Funktionen wie das Neuerstellen, Laden und Speichern von Modellen (aus einer textuellen Repräsentation) angeboten. Diese sog. *Modellanbindung* (Kapitel 9) nutzt hierfür die von Simulator X bereitgestellten Möglichkeiten zur Kommunikation zwischen den Komponenten der Anwendung.

1.2.4 Rendering

Für die Implementierung der 3D-Visualisierung, insbesondere für das leichte Hinzufügen von neuen grafischen Modellobjekten und die Realisierung von speziell für einen Modelleeditor benötigten *grafischen Effekten* (Kapitel 8) stand keine geeignete Plattform zur Verfügung. In *Modellierungswerkzeugen* (Unterabschnitt 2.1.3) ist es üblich, Informationen aus dem (Prozess-)Modell auf den grafischen Elementen durch Text oder andere Symbole zu visualisieren. Außerdem sollen *Interaktionszustände der Modellelemente* (Abschnitt 8.2) (selektiert, hervorgehoben, deaktiviert) geeignet visualisiert werden. „Deaktiviert“ bedeutet in diesem Zusammenhang, dass das Objekt transparent dargestellt wird, um den Blick auf dahinterliegende Grafikobjekte zu ermöglichen.

Um diese Anforderungen mit ausreichender Darstellungsqualität und -geschwindigkeit umsetzen zu können, wurde auf Basis von („modernem“) OpenGL eine *Render-Bibliothek* (Kapitel 11) und eine darauf aufbauende *Renderkomponente* (Kapitel 10) für Simulator X erstellt, die auf die Anforderungen des i>PM 3D-Projekts zugeschnitten, aber möglichst allgemein gehalten und erweiterbar sind.

1.3 Funktionale Anforderungen

Zusammengefasst werden in dieser Arbeit folgende funktionale Anforderungen an den i>PM3D Prototypen realisiert:

- (a) Modellierung von Prozessen mit einer grafischen Modellierungssprache nach einem allgemeinen, graphbasierten Ansatz in einer 3D-Darstellung
- (b) Möglichkeit, beliebige grafische Objekte – zusätzlich zu den Modellelementen – in der 3D-Szene anzuzeigen
- (c) Beschreibung der verwendeten Modellierungssprache durch Metamodelle
- (d) Möglichkeit, bestehende Modellkonstrukte und deren Visualisierung zu verändern sowie neue Modellelemente hinzuzufügen
- (e) Anbindung der Modelle an die Simulator X-Anwendung und Bereitstellung von Manipulationsmöglichkeiten an Modellelementen und deren Visualisierung
- (f) Erstellen, Laden und Speichern von Modellen in textueller Form
- (g) Bereitstellung von Grafikeffekten für einen Modelleditor: Darstellung von Text und 2D-Grafiken auf Modellfiguren; Visualisierung von selektierten, hervorgehobenen und deaktivierten Modellelementen
- (h) Anzeige von textuellen Attributen aus dem Prozessmodell auf den grafischen Objekten

Grundlagen

Dieses Kapitel beschäftigt sich mit Grundlagen, die für das Verständnis der Modellierungsaspekte dieser Arbeit wichtig sind. Zuerst wird eine Einführung in die Beschreibung von Prozessmodellen durch Sprachen gegeben und das Konzept der perspektivenorientierten Prozessmodellierung [JB96] gezeigt. Anschließend wird die Metamodellierung, insbesondere das Linguistic Meta Model [Vol11] vorgestellt, welches die Basis für die Anpassbarkeit des hier entwickelten Modellierungswerkzeugs darstellt.

2.1 Prozessmodellierungssprachen

Modellierung hat im Rahmen des Prozessmanagements die Aufgabe, komplexe (Geschäfts-)Abläufe aus der Realität in einer abstrahierten, das heißt vereinfachten, aber dennoch korrekten Form darzustellen¹. Einerseits werden Prozessmodelle erstellt, um Zusammenhänge besser zu erkennen und Optimierungsmöglichkeiten für den realen Prozess aufzuzeigen. Andererseits können abstrakt modellierte Prozesse von einem Softwaresystem automatisch ausgeführt bzw. simuliert werden [Hof+03].

Um Modelle formulieren zu können, bedarf es einer passenden Modellierungssprache. Zu einer Sprache gehört eine **abstrakte Syntax**, die allgemein Elemente einer Sprache und deren Beziehungen beschreibt, wohingegen die **konkrete Syntax** das „Aussehen“ der Sprache festlegt [CSW08]. Grundsätzlich lassen sich textuelle und grafische Notationen für Sprachen unterscheiden.

2.1.1 Visuelle Sprachen und deren Klassifikation

Wohl die leistungsfähigste „Schnittstelle“ des Menschen ist das visuelle System [War04], welches auf die Erkennung von Mustern und Strukturen ausgelegt ist. Daher eignen sich besonders grafische Darstellungen dafür, einen Überblick über komplexe Modelle zu geben und Zusammenhänge zwischen einzelnen Modellelementen aufzuzeigen. So spielen visuelle Sprachen auch eine große Rolle in der Prozessmodellierung [Hof+03] [JG08].

Die konkrete Syntax einer grafischen (oder „visuellen“) Sprache umfasst eine Ansammlung von grafischen Objekten (auch „Formen“ oder „Figuren“ genannt), die Sprachelemente repräsentieren. Elemente können auf verschiedene Arten miteinander in Beziehung gesetzt werden. Grafische Sprachen lassen sich nach diesem Kriterium prinzipiell in zwei Klassen, welche in Abbildung 2.1 gezeigt werden, und Mischformen einteilen [Cos+02].

¹Allgemein zum Modellbegriff und den Eigenschaften von Modellen: [Sta73]

So können visuelle Sprachen einem **graphbasierten** Ansatz folgen. Graphen bestehen aus Knoten und Kanten. Kanten drücken dabei eine Relation zwischen bestimmten Knoten aus, mit denen sie „verbunden“ sind. Für die Bedeutung ist das „Verbundensein“ über die Kante und nicht die Positionierung im Raum entscheidend.

Im Gegensatz dazu steht eine **geometriebasierte** Darstellung, welche die relative Positionierung von Modellelementen für die Darstellung von Beziehungen nutzt. So kann ein Objekt an einem anderen anhaften, oder in diesem enthalten sein.

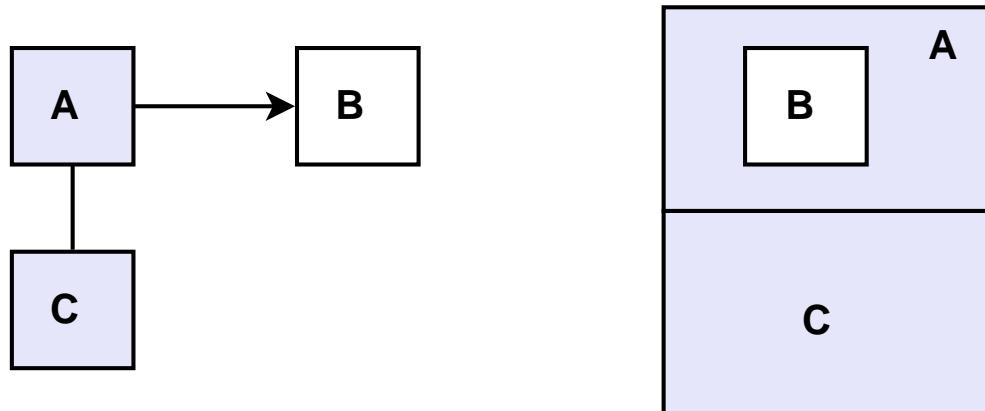


Abbildung 2.1: Graph- und geometriebasierter Ansatz im Vergleich

Aus den beiden Ansätzen können Mischformen („Hybride“) gebildet werden, die so eine größere Auswahl an Möglichkeiten zur Visualisierung von Beziehungen bieten können. In der Praxis sind daher solche Ansätze in der UML [BRJ99] und auch in der Prozessmodellierung zu finden, wie an den Beispielen in den folgenden Abschnitten zu sehen ist.

2.1.2 Perspektivenorientierte Prozessmodellierung

In einem Prozessmodell wird oft eine Vielzahl von Informationen dargestellt, die verschiedenste Bereiche der Prozessausführung beschreiben. Nach dem Konzept der perspektivenorientierten Prozessmodellierung (POPM) werden die „Informationsbestandteile“ eines Prozesses in sog. „Perspektiven“ (oder auch „Aspekte“ genannt) eingeteilt [JB96] [JG08].

Es wurden folgende fünf wichtigen Perspektiven identifiziert, die auch in [Vol11] (S.251f) beschrieben werden:

Funktionale Perspektive Dies umfasst alle funktionalen Einheiten eines Prozesses. Hier sind Ablaufschritte, Entscheidungen oder Konnektoren (AND, OR) eingeschlossen. Ablaufschritte werden wieder als „Prozess“ bezeichnet. Dies drückt aus, dass ein Prozessschritt selbst aus mehreren Schritten bestehen kann. Ein solcher Prozess(schritt) wird als „komposit“ bezeichnet. So ergibt sich eine Hierarchie von Prozessverfeinerungen.

Verhaltensorientierte Perspektive Dies wird auch als „Kontrollfluss“ bezeichnet und gibt die zeitlichen bzw. logischen Abhängigkeiten zwischen Elementen der funktionalen Perspektive an. Durch diese Perspektive wird also die Ausführungsreihenfolge festgelegt.

Organisationale Perspektive Einem Prozess lässt sich eine Entität zuordnen, die für die Ausführung verantwortlich ist, beispielsweise eine abstrakte Rolle oder eine konkrete Person.

Datenbezogene Perspektive Prozesse sind ohne Daten, die im Ablauf erstellt, modifiziert und ausgetauscht werden nahezu undenkbar. Datenflüsse legen oft auch die Abhängigkeiten zwischen Prozessschritten fest.

Operationale Perspektive Zur Ausführung von Prozessen sind verschiedene Betriebsmittel wie Maschinen, Werkzeuge oder Rechnerressourcen erforderlich, welche in dieser Perspektive abgebildet werden.

Dies soll explizit keine vollständige Aufzählung sein, sondern nur eine Zusammenfassung sehr häufig vorkommender Bestandteile. So kann es nötig sein, für einen Anwendungsfall weitere Perspektiven hinzuzufügen oder Perspektiven um neue Elemente zu erweitern. Daraus ergibt sich, dass (grafische) Modellierungssprachen, die POPM unterstützen möglichst erweiterbar sein sollten.

Abbildung 2.2² zeigt einen Prozess nach der perspektivenorientierten Prozessmodellierung.

Die funktionale Perspektive wird hier durch drei Prozesse sowie einen Entscheidungsknoten vertreten. Kontrollflüsse, die mit grauen Pfeilen visualisiert werden bilden die verhaltensorientierte Perspektive. Am Entscheidungsknoten kann sich der Kontrollfluss je nach Ausgang des Kriteriums (Einschreiben / Paket?) verzweigen. Mit dem blau eingekreisten Prozess sind Daten assoziiert, die in einem an den Prozess angehängten grauen Rechteck benannt werden.

Die drei bisher genannten Perspektiven werden, wie zu sehen ist, nach einem graphbasierten Ansatz visualisiert. Im Gegensatz dazu werden durch an die Prozessknoten „angeklebte“ Zeichenketten die organisationale (unten) und operationale (oben) Perspektive visualisiert. Dies entspricht dem geometriebasierten Ansatz.

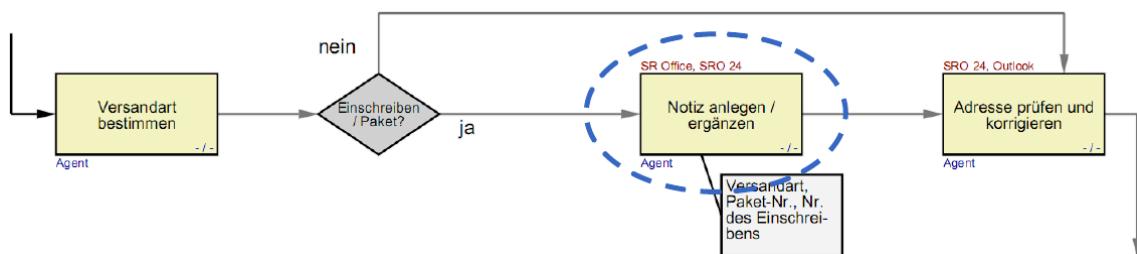


Abbildung 2.2: Perspektivenorientierte Prozessmodellierung aus [Rot11]

2.1.3 Grafische Modellierungswerzeuge

Für die Erstellung von grafischen Prozessmodellen am Rechner wird eine Unterstützung durch Softwarewerkzeuge benötigt. Prinzipiell können „Modelle“ einfach mit Hilfe von 2D-Zeichenwerkzeugen wie *Dia* [[www.dia](http://www.dia-project.org)] oder *MS Visio* [[www.visio](http://www.visio.com)] erstellt werden. Solche Programme bieten oft schon passende Formen und Verbindungen, beispielsweise für BPMN³ an. Ein Benutzer macht die Bedeutung eines solchen Diagramms an den erkennbaren grafischen Formen und deren Aussehen fest; insofern wäre dies für Menschen durchaus ausreichend.

Durch ein Zeichenprogramm wird das Diagramm intern nur als eine Ansammlung von Bildpunkten oder geometrischen Primitiven dargestellt und auch entsprechend gespeichert („persistiert“). Für ein solches Programm hat die Semantik der grafischen Konstrukte keinerlei Bedeutung. So ergibt sich ein Problem, wenn der modellierte Prozess automatisch ausgeführt oder verändert werden soll. Wie soll den grafischen Elementen eine Bedeutung zugeordnet werden?

Daher sind eher Werkzeuge gefragt, die auch intern eine „Vorstellung“ von Modellierungskonzepten haben [Vol11]. Solche Werkzeuge werden – auch in dieser Arbeit – „Modellierungswerzeuge“ genannt.

Ein solches grafisches Werkzeug bietet die Möglichkeit, Modelle zu erstellen, diese in formaler Weise zu persistieren und wieder aus einer physischen Repräsentation – beispielsweise einer Datei – zu laden.

²Das gezeigte Diagramm stammt aus dem Prozessmodellierungswerzeug i>PM [[ipm](http://ipm-project.org)].

³Business Process Modeling and Notation; umfasst eine standardisierte, (grafische) Prozessmodellierungssprache. Siehe [[bpmn](http://bpmn.org)].

Dem Benutzer wird üblicherweise eine Palette an Modellelementen angeboten, die in einem konkreten Prozessmodell eingesetzt werden können. Ein Anwender „baut“ ein Modell, indem er Instanzen der grafischen Objekte miteinander auf einer „Zeichenfläche“ kombiniert. Zu den konkreten, grafischen Elementen wird automatisch eine abstrakte Instanz angelegt, welche die Bedeutung des Modellelements festlegt.

Änderungen am abstrakten Modellelement können die Darstellung der grafischen Elemente beeinflussen. So kann beispielsweise in einem Prozessknoten dessen Funktion angegeben sein, welche in der grafischen Repräsentation als Text angezeigt wird. Bei einer Änderung der Funktion im Modellelement wird auch der Text auf dem Grafikobjekt angepasst. Genauso können Eigenschaften oder der Typ eines Modellelements durch Symbole auf den Grafikobjekten visualisiert werden.

Ein Modellierungswerkzeug für die perspektivenorientierte Prozessmodellierung wird in Abbildung 2.3 gezeigt. Auf der linken Seite lässt sich die Palette mit den Modellelementen erkennen, die in verschiedene „Gruppen“ eingeteilt sind.

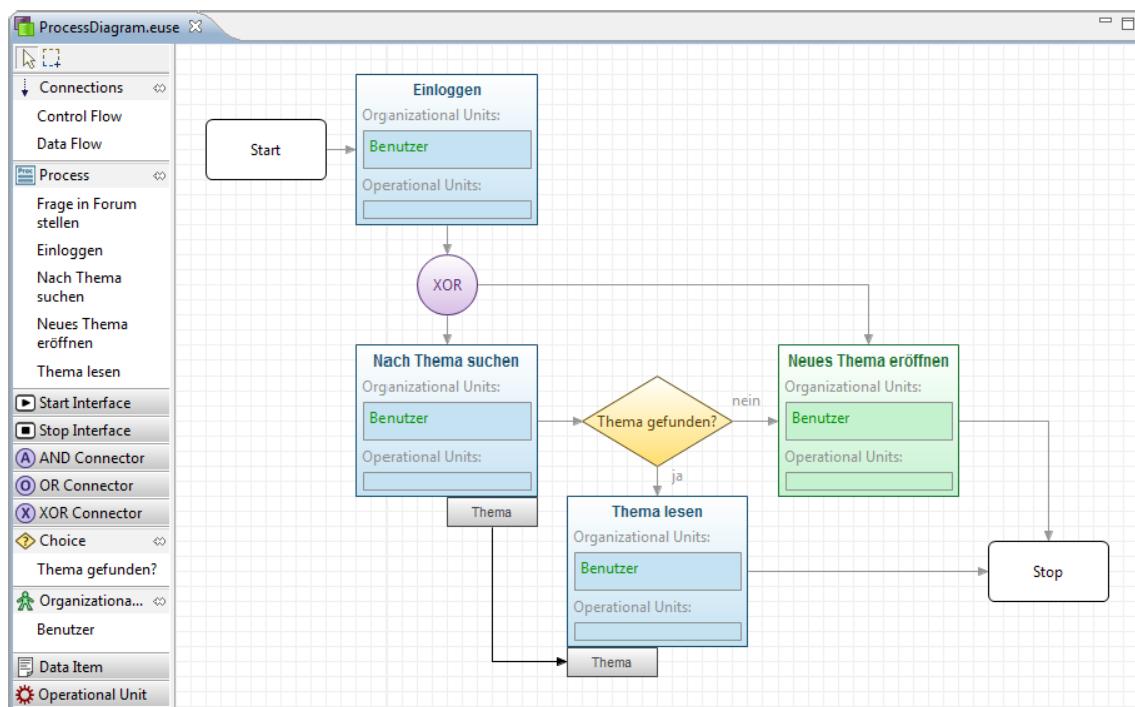


Abbildung 2.3: Prozessmodellierungswerkzeug i>PM2 aus [Rot11]

2.2 Metamodellierung

In der Prozessmodellierung kann es sinnvoll sein, die Modellierungssprache selbst zu verändern, um diese an spezielle Anforderungen anzupassen. So lassen sich Sachverhalte verständlicher und direkter als mit allgemeinen, fest vordefinierten Sprachen darstellen, indem spezialisierte Sprachelemente verwendet werden. An ein bestimmtes Einsatzgebiet angepasste Sprachen werden als **domänenspezifische Sprachen** (DSL) bezeichnet [CSW08].

Zur Beschreibung von Modellierungssprachen lässt sich das Konzept der **Metamodellierung** einsetzen [WS08] [Vol11]. Ein Metamodell lässt sich als ein Modell für eine Menge („Klasse“) von Modellen charakterisieren [Sei03].

Durch die Anpassung eines Metamodells, welches die abstrakte Syntax beschreibt, können neue Modellelemente hinzugefügt und bestehende angepasst oder entfernt werden. Andererseits lässt sich die

konkrete Syntax, im Falle einer grafischen Sprache also die grafische Repräsentation der Modellelemente ebenfalls durch ein Metamodell spezifizieren. So ist es möglich, zu einer abstrakten Syntax mehrere grafische Repräsentationen zu erstellen, die auf spezielle Anforderungen zugeschnitten sein können [JG08].

Um Metamodelle zu „erstellen“ ist es notwendig, diese auf eine wohldefinierte Weise beschreiben zu können. Dies leistet das im Folgenden vorgestellte Linguistic Meta Model (LMM), welches im Rahmen des Open Meta Modelling Environment (OMME), einer Metamodellierungsumgebung, entstanden ist [Vol11].

2.2.1 Linguistic Meta Model

LMM stellt eine Sprache bereit, welche zur Definition von Metamodellen dient. Abbildung 2.4 zeigt die grundlegenden LMM-Elemente und deren Hierarchie.

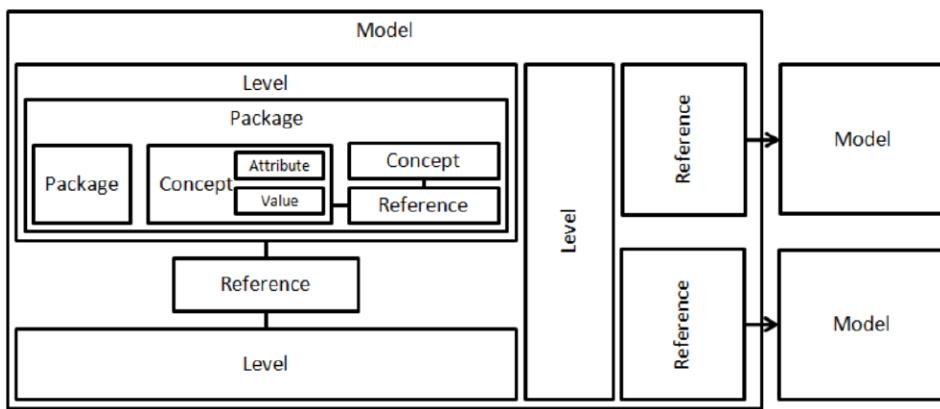


Abbildung 2.4: Hierarchie der LMM-Elemente aus [Vol11]

Das zentrale Element im LMM ist das **Concept**.

Ein Concept kombiniert Eigenschaften einer Klasse und eines Objekts, wie sie aus objektorientierten Programmiersprachen⁴ bekannt sind. So kann ein Concept – wie eine Klasse – Attribute definieren. Gleichzeitig kann ein Concept – wie ein Objekt – Wertzuweisungen enthalten. Anders ausgedrückt können Concepts sowohl eine „Typ-Facette“, die Attribute definiert als auch eine „Instanz-Facette“, die Zuweisungen vornimmt, beinhalten [AK00]. Dieses Prinzip wird mit dem Begriff **Clabject** (Class and Object) umschrieben.

Klassen stellen im objektorientierten System Typen dar; Objekte sind Instanzen von Klassen, welche Werte an die Attribute der Klasse zuweisen. Im Gegensatz zu der von Klasse und Objekt vorgegebenen Hierarchie aus zwei „Ebenen“ lassen sich mit Concepts Hierarchien mit beliebig vielen Ebenen realisieren. Dazu können Concepts gleichzeitig den Typ für Concepts auf der darunterliegenden Ebene und eine Instanz eines Concepts (`instanceOf`) auf der nächsthöheren Ebene darstellen. Ebenso gibt es die Möglichkeit für Concepts, andere Concepts analog zu Klassen zu „erweitern“ (`extends`), also einen Subtyp zu bilden.

Ein Vergleich zwischen Klasse-Objekt-Beziehungen und Concept-Concept-Beziehungen ist in Abbildung 2.5 zu sehen. In der Abbildung besitzt ConceptC eine Instanz-Facette, welche den Attributen aus ConceptA und ConceptB Werte zuweist. Die Typ-Facette von ConceptC stellt das Attribut c bereit, welches von ConceptD mit dem Wert 5.5 belegt wird.

⁴Dies deckt natürlich nicht alle objektorientierten Programmiersprachen ab. „Objektorientierung“ kann durchaus auf anderem Wege umgesetzt werden.

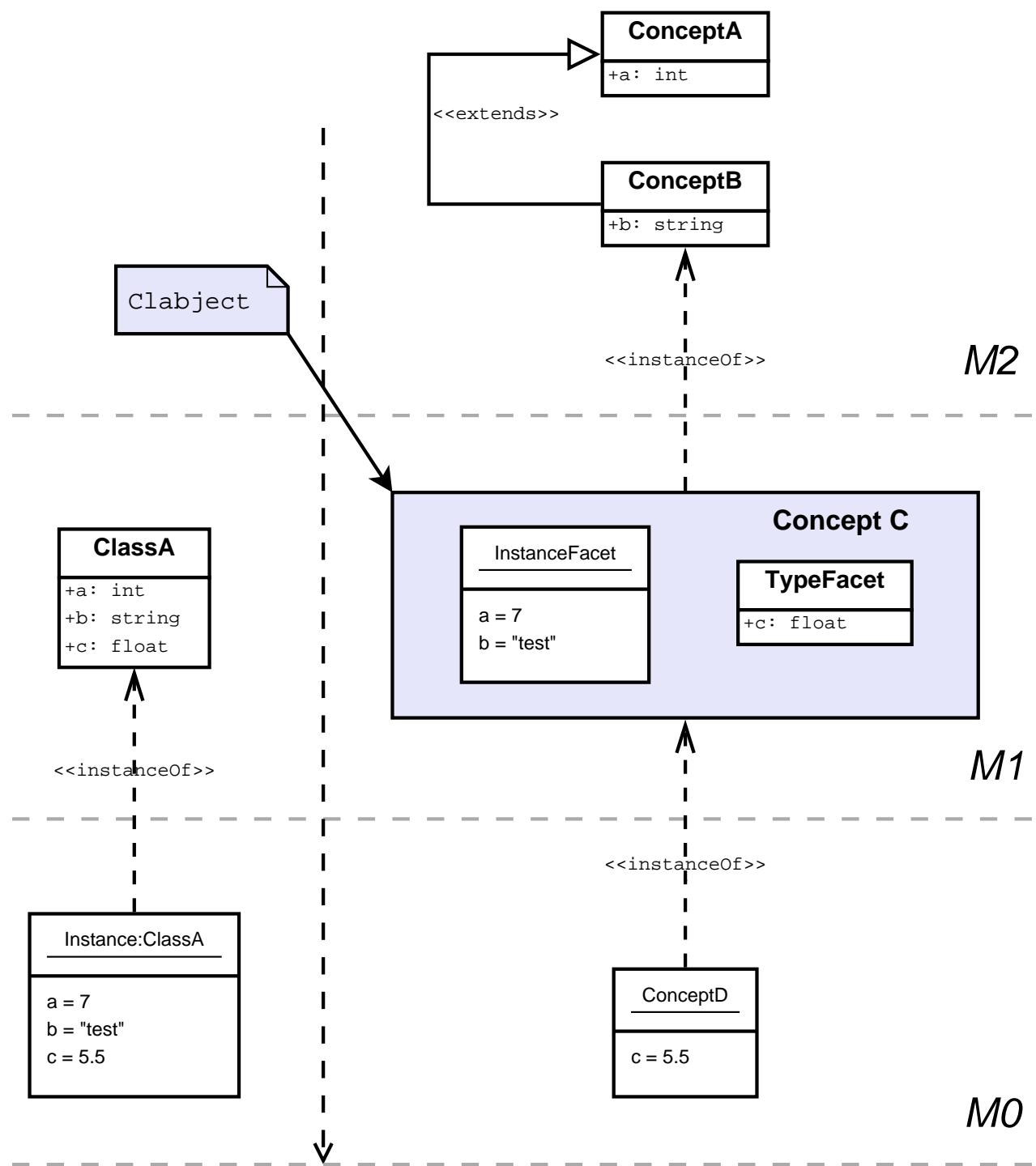


Abbildung 2.5: Vergleich von objektorientierter Modellierung (links) und Metamodellierung mit Clabjects

Concepts werden, wie in Abbildung 2.4 gezeigt, in **Packages** eingeordnet. Packages bilden zusammen einen **Level**, welcher eine Ebene in der Metamodellierungshierarchie repräsentiert. Mehrere Levels stellen zusammen das vollständige **Model** dar, wobei auch Modelle mit nur einer Ebene erlaubt sind.

Levels können ebenfalls zueinander in einer Instanzbeziehung (`instanceOf`) stehen. Wenn alle in einem Level *MA* definierten Concepts Instanzen von jeweils genau einem Concept in Level *MB* sind, ist *MA* eine Instanz von *MB*,

Die genannten Beziehungen wie `instanceOf` zwischen Levels bzw. Concepts werden in Abbildung 2.4 als „Reference“ dargestellt. In Concepts können sowohl **Literatyp-Attribute** (bspw. string, real, integer) als auch **Referenz-Attribute**, welche auf andere Concepts verweisen, angegeben werden.

Neben der schon erwähnten Instanziierung und Subtypbildung werden von LMM zusätzliche Modellierungsmuster unterstützt. Von diesen ist für die vorliegende Arbeit die sog. **Spezialisierung von Instanzen** bedeutend, deren Vorteile für die Modellierung von [Vol11] beschrieben werden. Dieses Muster wird in Abbildung 2.6 veranschaulicht.

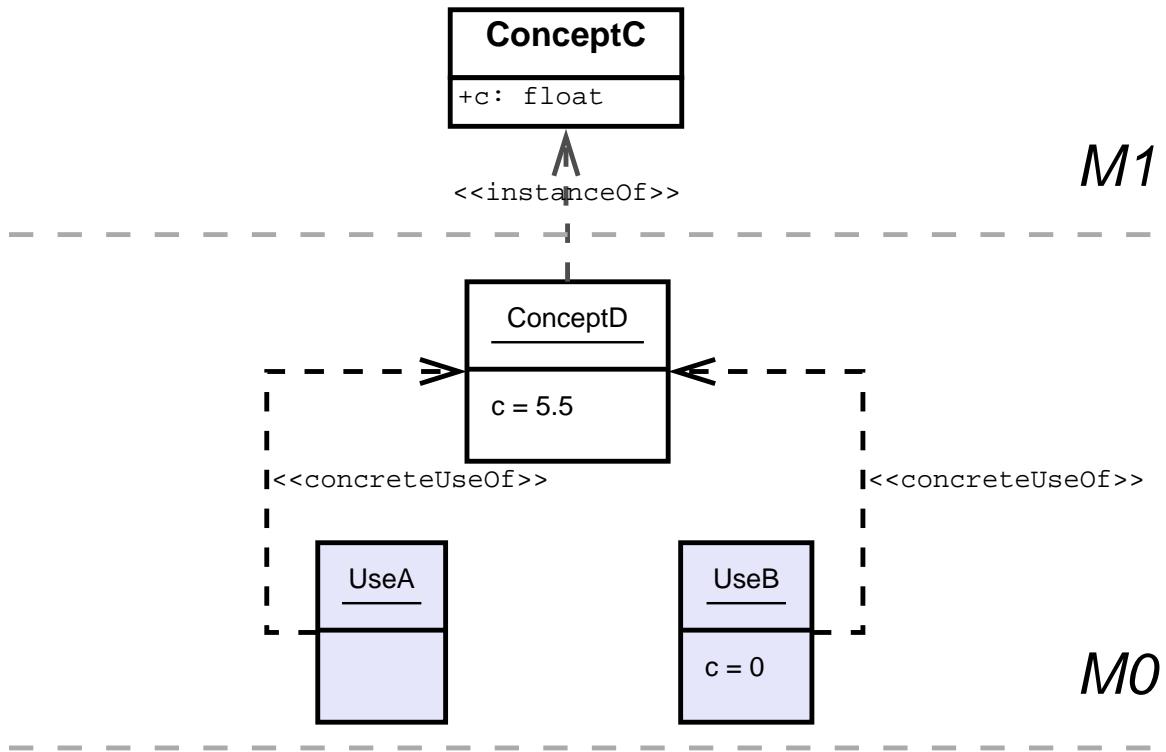


Abbildung 2.6: Instanz-Spezialisierung ausgehend von ConceptD

In der Abbildung spezialisiert **UseA** **ConceptD** (`concreteUseOf`). **UseA** übernimmt dabei alle Zuweisungen von **ConceptD**; damit hat das Attribut in **UseA** ebenfalls den Wert 5.5. **UseB** dagegen setzt wiederum einen Wert für das Attribut `c`. Das heißt, dass in **UseB** die bisherige Zuweisung „überschrieben“ wird und damit den Wert 0 hat. Für **ConceptD** ändert sich dabei nichts; die Überschreibung wirkt sich nur in **UseB** aus. In LMM lässt sich für Attribute festlegen, inwieweit das Setzen von Werten in Spezialisierungen zulässig ist und welche Bedeutung dies hat.

LMM-(Meta-)Modelle lassen sich mit der Sprache Linguistic Meta Language (LML) [Vol11] (S.159ff) in einer textuellen Form beschreiben. Die Syntax ist an bekannte Programmiersprachen wie C++ oder C# angelehnt und kann daher als „menschenlesbar“ angesehen werden. Gleichzeitig ist es damit möglich, LMM automatisch zu verarbeiten oder es sogar für die Beschreibung von Software zu nutzen, wie im Folgenden am Beispiel des MDF gezeigt wird.

Beispielsweise sieht ein Concept mit einer Zuweisung und einer Attributdefinition in LML wie folgt aus:

```

concept ConceptC instanceOf ConceptB {
    a = 7;
    real c;
}

```

Zur einfachen Bearbeitung von LMM-Modellen wird von OMME ein textueller Editor auf Basis von Xtext [[www:xtext](#)] bereitgestellt.

2.2.2 Model Designer Framework

Ebenfalls als Teil der Metamodellierungsumgebung OMME ist das Model Designer Framework (MDF) von Roth [Rot11] entwickelt worden. Dieses erlaubt es, Modell-Editoren mit Hilfe von LMM-Metamodellen zu spezifizieren. So lassen sich grafische Modellierungswerkzeuge („Editoren“) auf Basis von MDF für beliebige (domänenspezifische) Modellierungssprachen erstellen.

Abbildung 2.7 zeigt die in MDF verwendeten Modelle. Hier sollen nur kurz die für die vorliegende Arbeit wichtigsten Aspekte verdeutlicht werden. Details können bei Roth in Kapitel 5, Modellhierarchie nachgelesen werden.

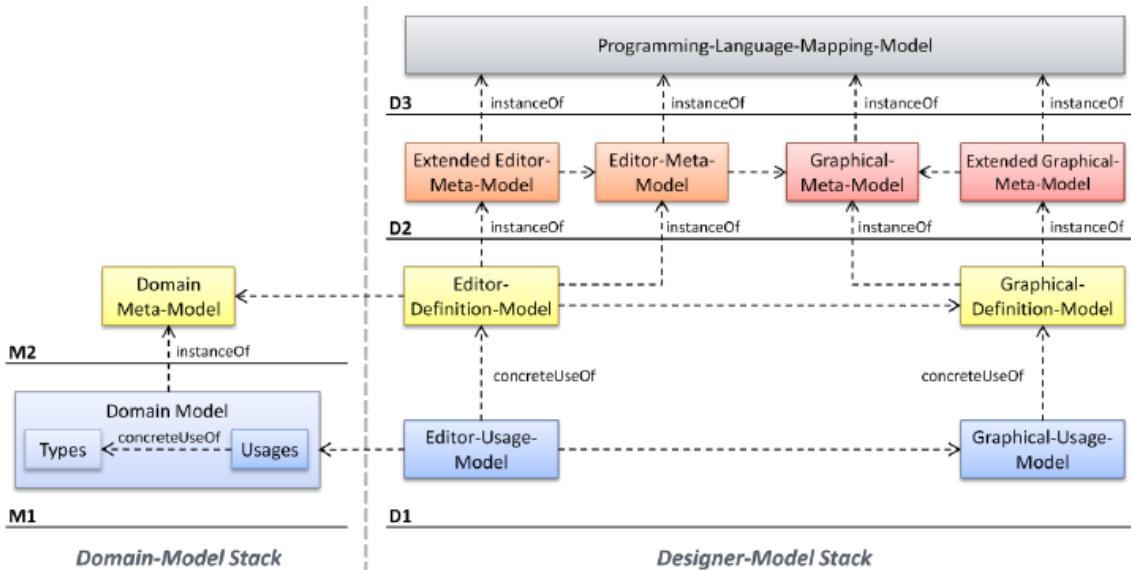


Abbildung 2.7: Modellhierarchie von MDF mit Domain-Model- und Designer-Stack aus [Rot11]

Der *Domain-Model-Stack* (links) enthält alle Modelle, die für die Domäne relevant sind. Das *Domain-Metamodel* legt die Elemente der domänenspezifischen Sprache fest, welche im *Domain-Model* genutzt wird, um ein Modell zu beschreiben.

Rechts wird der *Designer-Model-Stack* gezeigt, der den Editor für die Domäne spezifiziert. Das *Graphical-Definition-Model* beschreibt Figuren, die sich für die Visualisierung der Domäne einsetzen lassen. Figuren werden über das *Editor-Definition-Model* mit den Domänenmodellelementen verbunden. So wird die grafische Repräsentation der Modellelemente im Editor festgelegt. Bemerkenswert ist, dass LMM sowohl für die Beschreibung des Modellierungswerkzeugs als auch für die persistente Speicherung und interne Darstellung der mit dem Werkzeug erstellten Modelle genutzt wird.

Abbildung 2.9 zeigt einen Prozess, der in einem mit MDF definierten Editor (*i>PM²*) für die *POPM* (Unterabschnitt 2.1.2) erstellt wurde.

2.3 Typ-Verwendungs-Konzept

An Abbildung 2.8 und Abbildung 2.9 lässt sich das „Typ-Verwendungs-Konzept“, welches von i>PM² umgesetzt wird, zeigen.

Das Grundprinzip des Typ-Verwendungs-Konzeptes ist es, einmal erstellte Objekte in unterschiedlichen Zusammenhängen zu verwenden. Dieses Konzept lässt sich durch die in LMM (Unterabschnitt 2.2.1) eingeführte Spezialisierung von Instanzen leicht realisieren⁵. Die Spezialisierung von Instanzen, deren Einsatz für das Typ-Verwendungs-Konzept und das im Folgenden gezeigte Beispiel werden auch in der Arbeit von Volz [Vol11] (S.56ff) beschrieben.

Abbildung 2.8 zeigt den Prozess „Notiz aufnehmen“ (A). Nun wird eine sehr ähnliche Funktionalität für einen anderen Prozess benötigt, der in Abbildung 2.9 gezeigt ist. Hier ist der Prozess „Notiz erstellen / ergänzen“ (B) zu sehen. Um diesen Prozess zu definieren, könnte nun ein komplett neues „Objekt“ erstellt werden. Es ist allerdings schon ein „Objekt“ mit nahezu gleichen Eigenschaften vorhanden, nämlich der vorher genannte Prozess A. Wie in der Informatik üblich wäre es wünschenswert, solche Redundanzen zu vermeiden und die „Wiederverwendbarkeit“ zu erhöhen.

Dazu kann ein „Typ“ definiert werden, vom dem mehrere „Verwendungen“ erstellt werden, die dann in mehreren Kontexten eingesetzt werden können. Hier könnte beispielsweise der Typ T angelegt werden, welcher einen Prozess repräsentiert. T legt fest, dass die Funktion des Prozesses „Notiz aufnehmen“ (der auf der Figur angezeigte Text) ist und „OneNote“ und „Agent“ mit ihm assoziiert sind. Prozess A kann als Verwendung von T gesehen werden; A übernimmt alle Eigenschaften von T.

Um den Prozess B darzustellen, müssen jedoch zwei Änderungen vorgenommen werden. Das ist möglich, da eine Verwendung Werte des Typs überschreiben kann. So wird in der Verwendung für B die vordefinierte Funktion durch „Notiz erstellen / ergänzen“ ersetzt und „Outlook“ zu den operationalen Einheiten hinzugefügt.

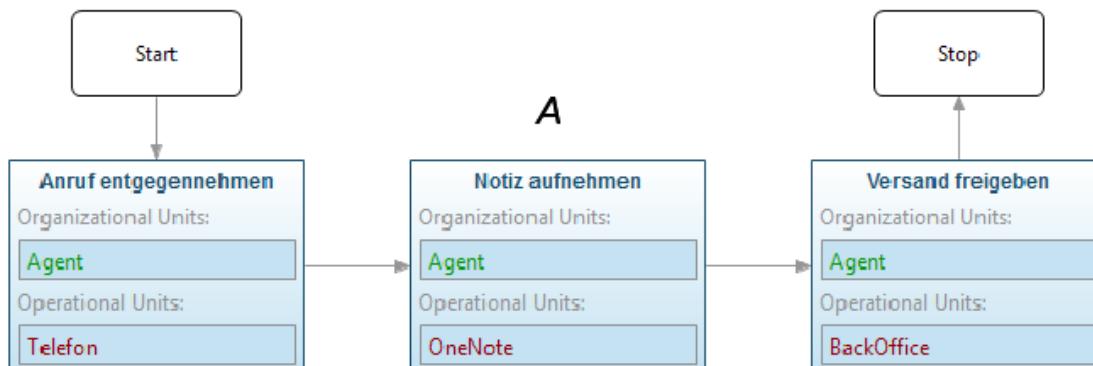


Abbildung 2.8: Prozess in i>PM2 aus [Vol11] (Bezeichner A hinzugefügt)

Das Typ-Verwendungs-Konzept ist auch in i>PM² (Abbildung 2.3) zu erkennen. Die Palette (links) zeigt unter „Process“ die davon instanzierten „Typen“, wovon für die Zeichenfläche „Verwendungen“ erstellt werden. Rechts auf der Zeichenfläche ist eine Verwendung vom Typ „Neues Thema eröffnen“ mit geänderter Grundfarbe zu sehen.

⁵Nach der Terminologie des Typ-Verwendungs-Konzepts ist in der Abbildung 2.6 ConceptD ein „Typ“, UseA und UseB sind „Verwendungen“ davon.

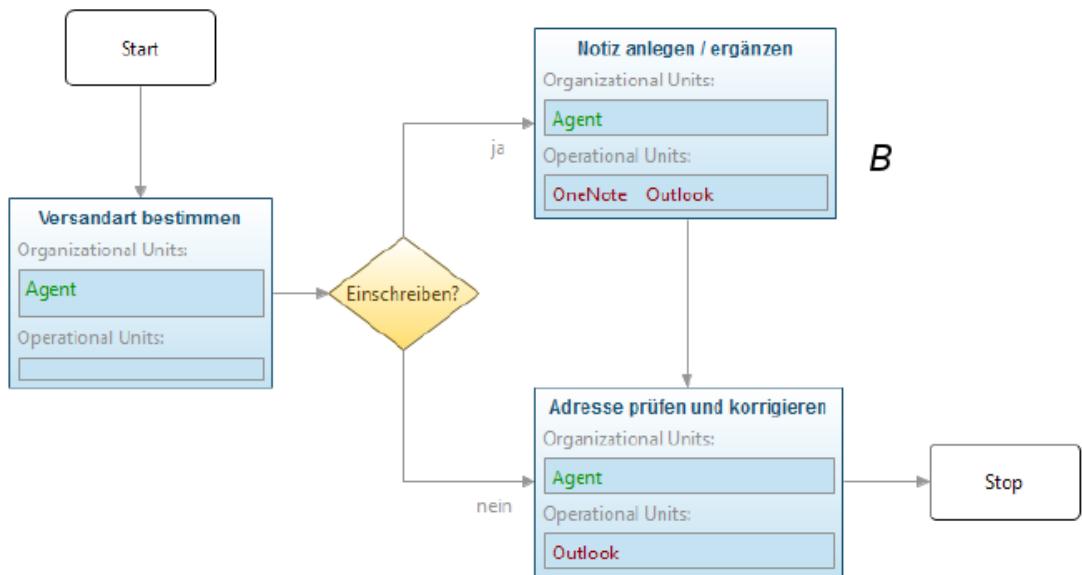


Abbildung 2.9: Prozess mit angepasster Verwendung aus [Vol11] (B hinzugefügt)

Verwandte Arbeiten zur 3D-Visualisierung

Neben den (wenigen) Arbeiten, die sich explizit mit der dreidimensionalen Visualisierung und Modellierung von Prozessen beschäftigen, sollen hier auch solche vorgestellt werden, die sich allgemein oder in verwandten Gebieten wie der Softwaremodellierung mit 3D-Visualisierungen beschäftigen.

Die hier gezeigten Arbeiten sollen als Anregung oder Motivation für die in dieser Arbeit dargestellte 3D-Visualisierung von Prozessen dienen. Außerdem sollen Ideen für zukünftige Erweiterungen des vorliegenden Projekts gesammelt und prinzipielle Vor- und Nachteile von 3D-Visualisierungen beleuchtet werden.

3.1 3D-Softwarevisualisierung

Nahe verwandt mit der Prozessmodellierung ist die Modellierung von Software. Nicht selten sind Softwaresysteme überaus umfangreich und es muss daher nach Möglichkeiten gesucht werden, eine Vielzahl von Informationen übersichtlich und klar zu visualisieren ohne den Betrachter zu überfordern.

Bisher nutzen Werkzeuge zur Softwaremodellierung, die häufig auf der Unified Modeling Language (UML) aufsetzen nahezu ausschließlich 2D-Visualisierungen. Die UML lässt prinzipiell aber auch 3D-Repräsentationen zu [BRJ99]. Einen umfassenden Überblick über Arbeiten, die sich mit 3D-Softwarevisualisierung befassen, gibt [TC09].

3.1.1 Wilma: Ein 3D-Modellierungstool für UML-Diagramme

[Dwy01] weist auf die Probleme von Softwarevisualisierungstechniken hin, große und insbesondere hierarchisch aufgebaute Diagramme darzustellen. 3D-Darstellungen hätten hier Vorteile durch die Möglichkeit, Hierarchieebenen des Diagramms als Flächen im 3D-Raum zu zeigen.

Die Platzierung von UML-Elementen per Hand sei eine zeitraubende Aufgabe, die besonders im dreidimensionalen Raum wegen der schlechten Verfügbarkeit von 3D-Eingabegeräten zum Problem werde. Daher wird eine Anordnung der Diagrammelemente im 3D-Raum mit Hilfe eines automatischen, kräftebasierten Layout-Algorithmus vorgeschlagen.

Es wurde ein Prototyp („Wilma“)¹ realisiert, der Graphen in einer 3D-Darstellung zeichnen kann und

¹ Quellcode und ausführbare Dateien des (weiterentwickelten) Prototyps „WilmaScope“ (auf Basis von Java3D) können unter <http://wilma.sourceforge.net/> heruntergeladen werden.

Interaktionsmöglichkeiten mit den Graphelementen bietet. Ein damit erstelltes 3D-Klassendiagramm ist in Abbildung 3.1 zu sehen.

So werden Klassen durch 3D-Quader dargestellt, deren Seiten beschriftet werden können. Um die Lesbarkeit sicherzustellen, wird immer eine Seite des Würfels auf den Betrachter ausgerichtet. Pakete werden durch transluzente Kugeln und Verbindungen zwischen Klassen durch gestreckte 3D-Zylinder dargestellt.

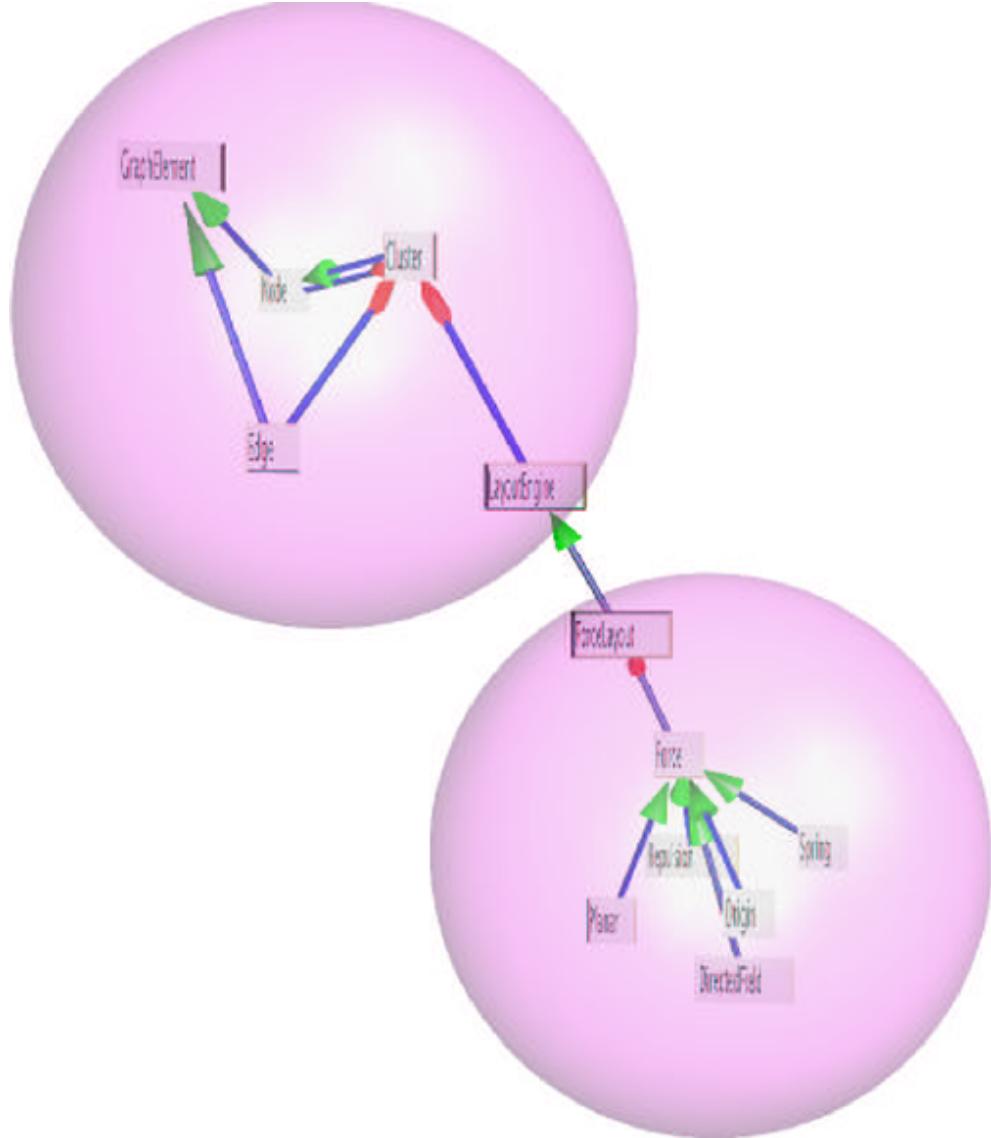


Abbildung 3.1: 3D-UML-Klassendiagramm aus [Dwy01]

In einer Studie zur Benutzbarkeit seien die 3D-Diagramme von Beteiligten als nützlich für das Verständnis des Modells eingestuft worden. Es wird ein Benutzer zitiert, der die Möglichkeit, das Diagramm aus verschiedenen Richtungen betrachten zu können besonders positiv kommentiert.

Auch seien Benutzer gebeten worden, selbst ein 3D-Diagramm nach einer textuellen Vorlage zu modellieren, wobei die meisten wenig Probleme mit der Aufgabe gehabt hätten. Es wird jedoch vermutet, dass die 3D-Darstellung bei einigen Benutzern eine gewisse Eingewöhnungszeit voraussetzen könnte. Probanden mit vorheriger Erfahrung aus 3D-Computerspielen hätten im Versuch die wenigsten Schwierigkeiten mit der Navigation im 3D-Raum gehabt.

3.1.2 X3D-UML: Hierarchische 3D-UML-Zustandsdiagramme

In [MHS08] werden Zustandsdiagramme („state diagrams“) der UML in den 3D-Raum übertragen.

Zu Beginn seien von vier Unternehmen erhaltene Zustandsdiagramme untersucht worden, die mit dem Modellierungswerkzeug IBM Rational Rose erstellt wurden. Daraus habe sich ergeben, dass die Modelle oft hierarchisch aus Unterzuständen aufgebaut seien. In RationalRose würden diese Unterdiagramme jedoch in separaten Tabs dargestellt, was dazu führe, dass Betrachter ständig zwischen einzelnen Diagrammen hin- und herwechseln müssten. Das erschwere das Erkennen von Zusammenhängen und groben Strukturen, was von befragten Benutzern bemängelt worden sei. Die Einschränkungen durch die Tab-Ansicht würden auf verschiedenem Wege „umgangen“, etwa indem separate Handskizzen angefertigt würden. Andere Benutzer würden „in die Luft starren“, um sich die Zusammenhänge und Auswirkungen von Änderungen besser vorstellen zu können.

Daher sei es die wichtigste Anforderung an eine 3D-Repräsentation, hier Abhilfe zu schaffen und hierarchische Zustandsdiagramme besser abzubilden. Es wird eine Darstellung vorgeschlagen, welche die Zustandsdiagramme selbst immer noch zweidimensional zeichnet, diese jedoch auf ebenen Flächen im 3D-Raum platziert. So würden sich Beziehungen zwischen mehreren Diagrammen gut grafisch darstellen lassen. Wie sich in Abbildung 3.2 erkennen lässt, werden Beziehungen zwischen Super- und Subzuständen durch transluzente, graues Dreiecke dargestellt.

Solche Diagramme seien Benutzern mit Erfahrung in Rational Rose vorgelegt worden, welche sich insgesamt positiv zur Nützlichkeit jener 3D-Diagramme geäußert hätten. Von den Benutzern seien verschiedene Erweiterungen vorgeschlagen worden; unter anderem eine Filtermöglichkeit, mit der sich uninteressante Details verbergen lassen, Einschränkungen der Navigation um ungünstige Sichten auf das Modell zu vermeiden (bspw. direkt von der Seite, so dass die 2D-Elemente nicht erkennbar sind) sowie Funktionen, um schnell zwischen verschiedenen Betrachtungsperspektiven wechseln zu können.

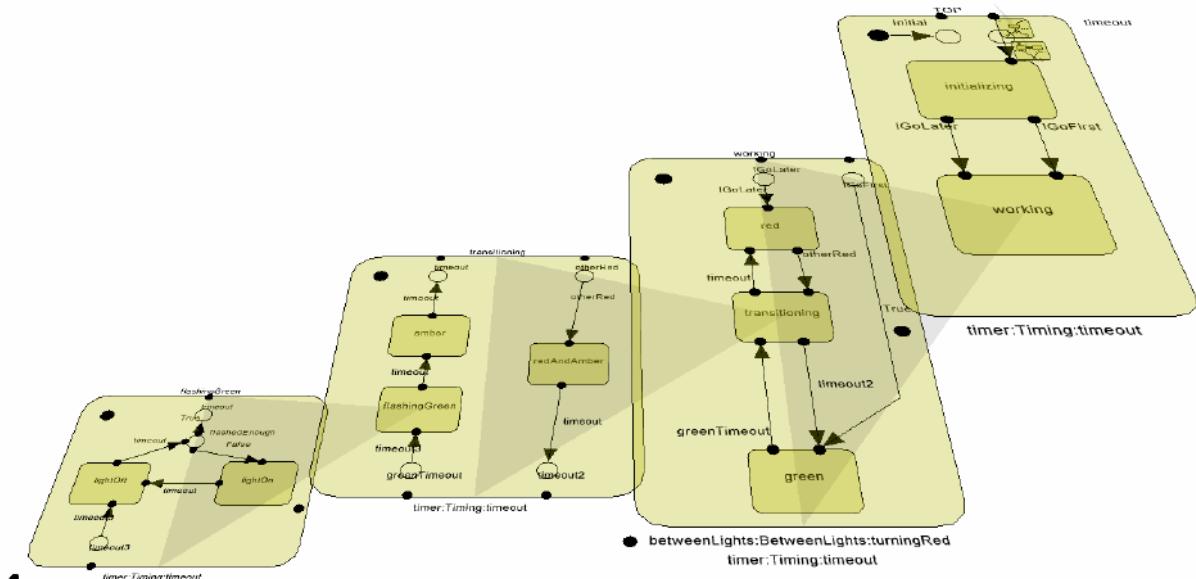


Abbildung 3.2: Hierarchisch aufgebautes 3D-UML-Zustandsdiagramm aus [MHS08]

3.1.3 3D-Visualisierung von großen, hierarchischen UML-Zustandsdiagrammen

3D-Visualisierungen von (großen) UML-Zustandsdiagrammen werden auch von [KN09] und, darauf aufbauend, [AG09] untersucht. Zustandsdiagramme werden, wie in [MHS08] auf Flächen im 3D-Raum gezeichnet, wobei hier die Zustände selbst als 3D-Objekte dargestellt sind, um den visuellen Eindruck zu verbessern, wie in Abbildung 3.3 zu sehen ist.

Abbildung 3.4 zeigt, wie in komplexen Diagrammen komplettte Diagrammteile ausgeblendet werden können, um momentan unwichtige Details zu verbergen und die Übersichtlichkeit zu erhöhen.

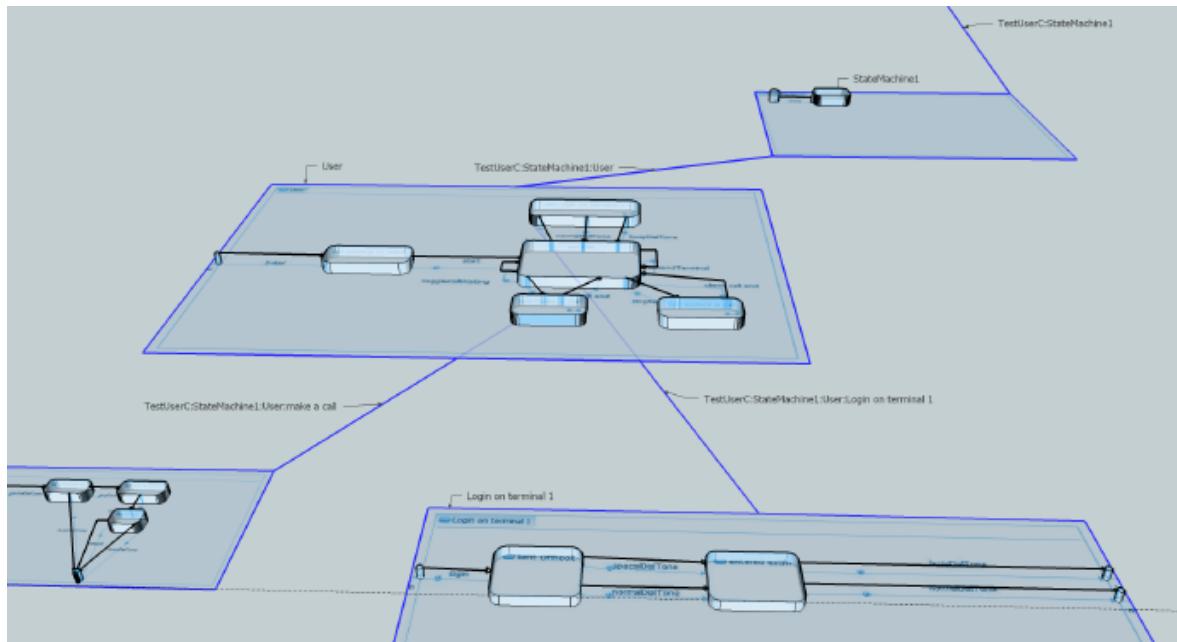


Abbildung 3.3: 3D-Zustandsdiagramm aus [KN09]

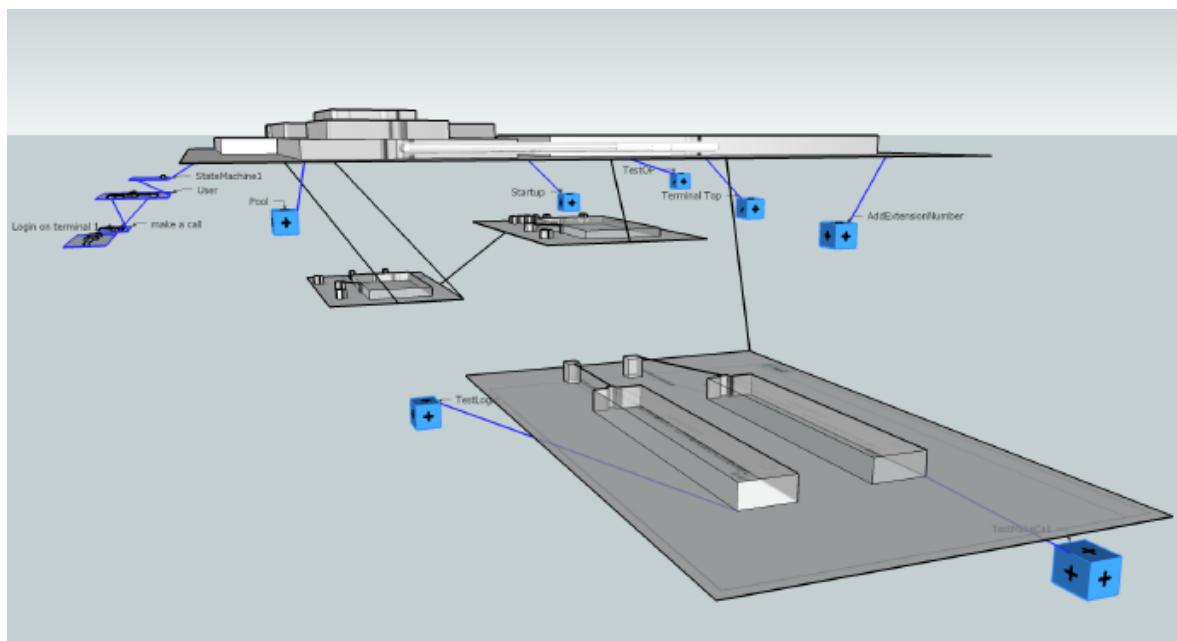


Abbildung 3.4: Zustandsdiagramm mit ausgeblendeten Diagrammteilen (dargestellt durch blaue Würfel) aus [KN09]

3.1.4 Dreidimensionale Darstellung zur besseren Visualisierung von Beziehungen

[GK98] merkt an, dass durch 3D-Visualisierungen die Ausdrucksstärke von (graphbasierten) grafischen Notationen deutlich erhöht werden können. Besonders vorteilhaft seien 3D-Visualisierungen von Graphen, wenn es darum ginge, eine Vielzahl von unterschiedlichen Beziehungs- bzw. Verbindungstypen darzustellen. Im 2D-Bereich habe man nur relativ eingeschränkte Möglichkeiten, unterschiedliche Verbindungstypen durch Farbe, unterschiedliche Linientypen oder durch Konnektoren – Symbole an den Enden der Linien – voneinander abzugrenzen. Um diese Probleme im 2D-Raum zu umgehen würden oft unterschiedliche Graphen bzw. Diagrammtypen genutzt. Dabei besäßen Knoten in unterschiedlichen Diagrammtypen oft die gleiche Bedeutung während Verbindungen eine komplett andere Semantik hätten. Problematisch sei die Repräsentation von Zusammenhängen zwischen unterschiedlichen Diagrammtypen, was allgemein einen großen Schwachpunkt von Modellierungssprachen darstelle.

Zur Visualisierung von Verbindungen lasse sich die dritte Dimension, also die z-Richtung sinnvoll nutzen. Verbindungen in der x-y-Ebene hätten eine andere Bedeutung als die, die aus der Ebene heraus in z-Richtung verlaufen. So würden sich mehrere Diagrammtypen in eine Darstellung integrieren lassen.

Die dritte Dimension ließe sich auch als Zeitachse interpretieren. So sei es möglich, in 3D-Sequenzdiagrammen die Zustände des Systems zu bestimmten Zeitpunkten auf parallelen Flächen darzustellen, zu denen die Zeitachse senkrecht steht wie in Abbildung 3.5 gezeigt wird.

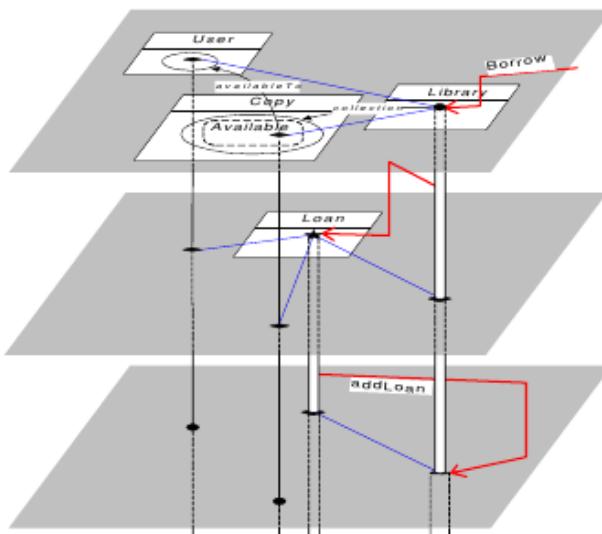


Abbildung 3.5: 3D-UML-Sequenzdiagramm; Ausschnitt aus [GK98]

3.1.5 Nutzung von Tiefeneindruck und Animation zur Visualisierung von UML-Diagrammen

In [GRR99] wird ebenfalls die 3D-Darstellung von UML-Diagrammen, speziell Klassen-, Objekt- und Sequenzdiagrammen untersucht. Die dritte Dimension könne beispielsweise dafür genutzt werden, als „uninteressant“ eingestufte Elemente in den Hintergrund zu schieben und damit Elemente im Vordergrund besonders hervorzuheben.

In Abbildung 3.6 und Abbildung 3.7 wird das Prinzip am Beispiel eines Klassendiagramms verdeutlicht. Bei letzterer Abbildung ist zu sehen, dass bei Klassen, die nah am Betrachter sind, mehr Information dargestellt wird als bei den in größerer Entfernung dargestellten Klassen, bei denen nur der Name als Text zu erkennen ist. Zusätzlich wird die Nutzung von Animationen vorgeschlagen, um Übergänge zwischen verschiedenen Visualisierungsperspektiven – wie zwischen den beiden gezeigten Abbildungen – anschaulicher zu machen.

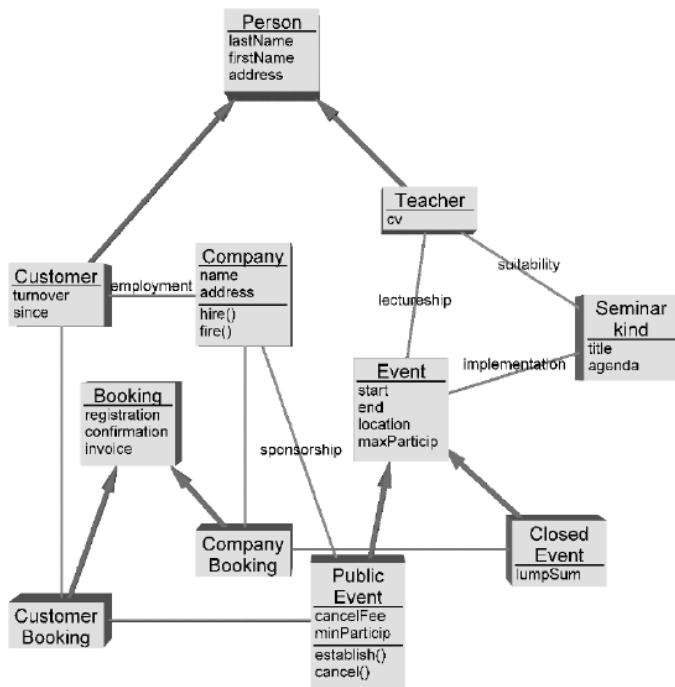


Abbildung 3.6: 3D-UML-Klassendiagramm aus [GRR99]

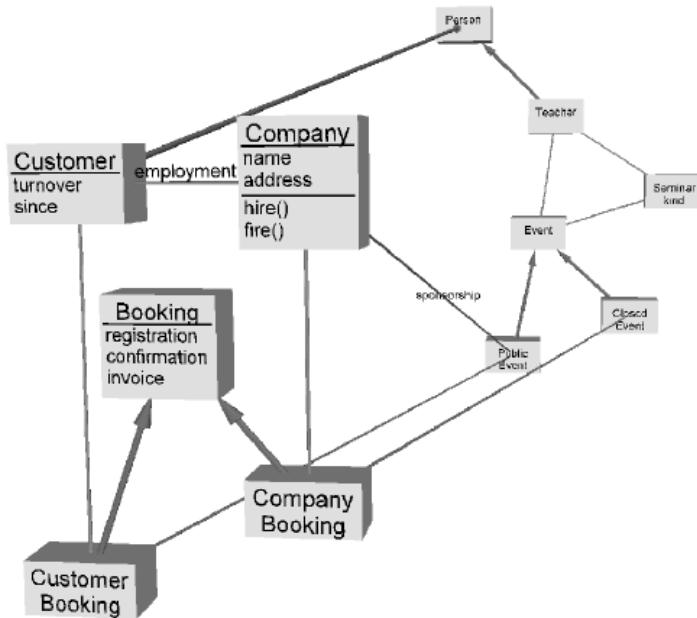


Abbildung 3.7: Diagramm mit nach hinten verschobenen, „unwichtigen“ Klassen aus [GRR99]

3.1.6 Graphical Editing Framework 3D

Bei GEF3D handelt es sich um ein Framework zur Erstellung von Modell-Editoren [PD08]. Das Projekt basiert auf den Konzepten des Grafical Editing Framework der Eclipse Plattform und überträgt diese in den dreidimensionalen Raum.

Mit GEF3D ist es möglich, 3D-Editoren für Eclipse zu erstellen und schon vorhandene, GEF-basierte 2D-Editoren darin einzubetten indem 2D-Elemente auf Flächen im dreidimensionalen Raum gezeichnet würden. Abbildung 3.8 zeigt ein Beispiel für die Darstellung von mehreren Diagrammtypen in einer Ansicht und Verbindungen zwischen Elementen verschiedener Diagramme.

In Abbildung 3.9 ist ein mit GEF3D implementierter Ecore-Editor zu sehen. Diese Darstellungsform mit 2D-Elementen, die im 3D-Raum platziert werden können wird als „2.5D“-Darstellung bezeichnet. Elemente könnten, wie in der Abbildung zu sehen ist, auf Flächen oder auch frei im 3D-Raum platziert werden [www.gef3ddevblog].

Die grafische Ausgabe von GEF3D baut direkt auf OpenGL auf; um 2D-Grafiken und Text zu zeichnen werde Vektorgrafik genutzt, was zu einer besseren Darstellungsqualität im Vergleich zu texturbasiertem 2D-Rendering führe².

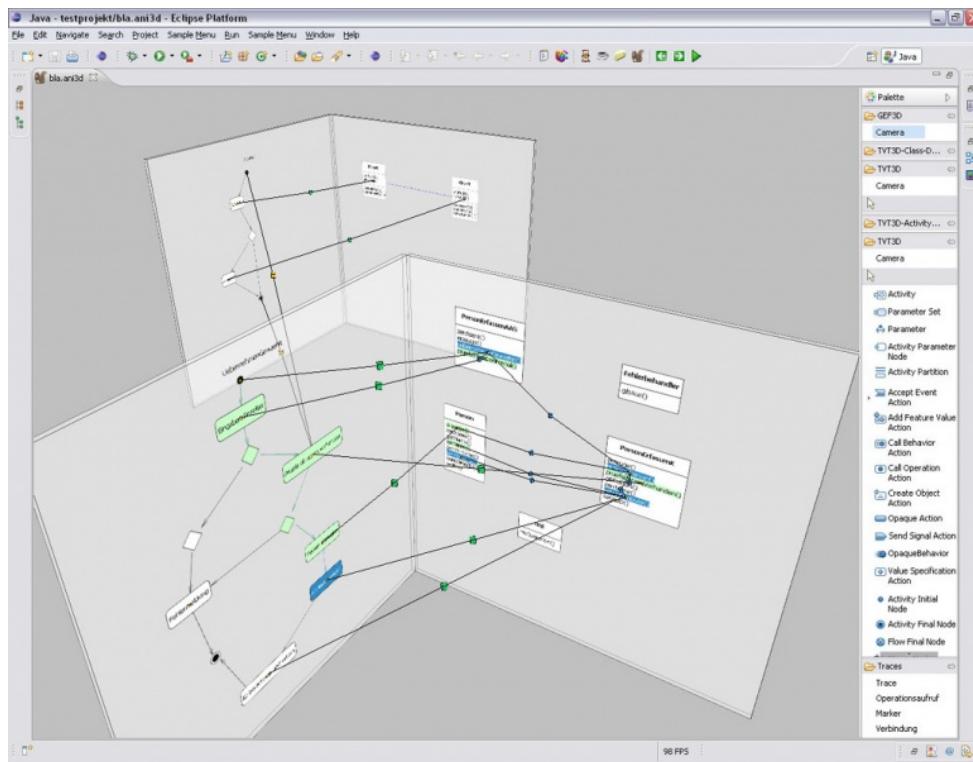


Abbildung 3.8: Kombination mehrerer 2D-Editoren in einer 3D-Ansicht von [www.gef3d]

²Ein verbreiteter Ansatz, um 2D-Grafiken und Text in OpenGL darzustellen ist es, diese erst in eine Textur zu zeichnen und diese auf 3D-Objekte aufzubringen. Dies wird auch in dieser Arbeit verwendet.

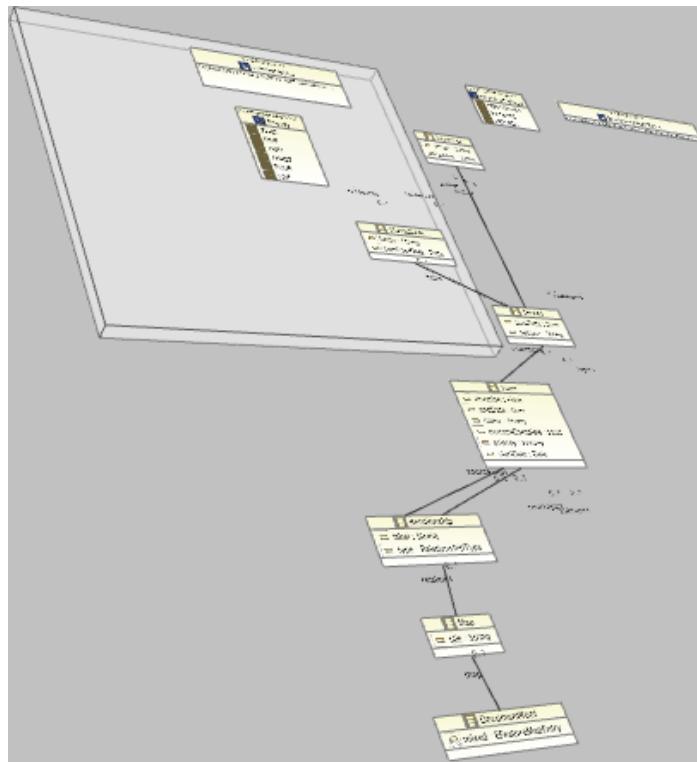


Abbildung 3.9: 3D-Ecore-Editor von [www:gef3ddevblog]

3.2 3D-Prozessvisualisierung

Arbeiten, die sich speziell mit 3D-Visualisierungen im Kontext des Prozessmanagements (Prozessmodellierung, -simulation) beschäftigen, werden hier vorgestellt. Außerdem wird gezeigt, wie sich abstrakte Modelle in eine virtuelle Umgebung integrieren lassen.

3.2.1 3D-Repräsentation von Prozessmodellen

Von [Bet+08] wird die Visualisierung von Prozessen mittels dreidimensional dargestellter Petrinetze vorgestellt. Es werden verschiedene Szenarien gezeigt, in denen 3D-Visualisierungen gewinnbringend genutzt werden könnten.

Es wird das Problem angesprochen, dass für die Modellierung von Prozessen oft verschiedene Diagrammtypen nötig seien, zwischen denen in üblichen 2D-Werkzeugen zeitraubend gewechselt werden müsse. Mehrere Diagrammtypen in eine 3D-Ansicht zu integrieren könnte hier Abhilfe schaffen.

Als Beispiel wird Abbildung 3.10 eine Kombination eines Organisationsmodells mit einem Prozessmodell gezeigt. Neben den Beziehungen zwischen Aktivitäten im Prozessmodell und den Rollen des Organisationsmodells sei es gleichzeitig möglich, Beziehungen im Organisationsmodell, wie die Generalisierung von Rollen oder die Zuordnung von Akteuren zu Rollen zu visualisieren.

Ein weiteres Anwendungsszenario für 3D-Visualisierungen sei es, Ähnlichkeiten zwischen verschiedenen Prozessmodellen aufzuzeigen.

Im 3D-Raum sei es einfach möglich, zu vergleichende Prozesse nebeneinander auf parallelen Ebenen im Raum zu platzieren. Verbindungen zwischen Modellelementen der gegenüber gestellten Prozessmodelle könnten dafür genutzt werden, mit verschiedenen Metriken berechnete Ähnlichkeitswerte anzuzeigen. Wie in Abbildung 3.11 zu sehen ist werden die Werte sowohl durch die Beschriftung als auch durch die Dicke der Verbindungslien visualisiert.

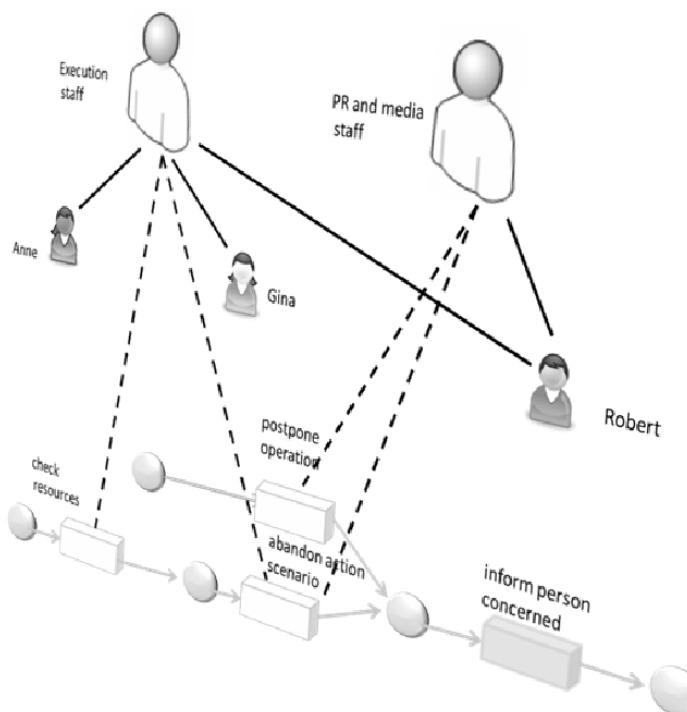


Abbildung 3.10: Darstellung von Beziehungen zwischen Prozess- und Organisationsmodell aus [Bet+08]

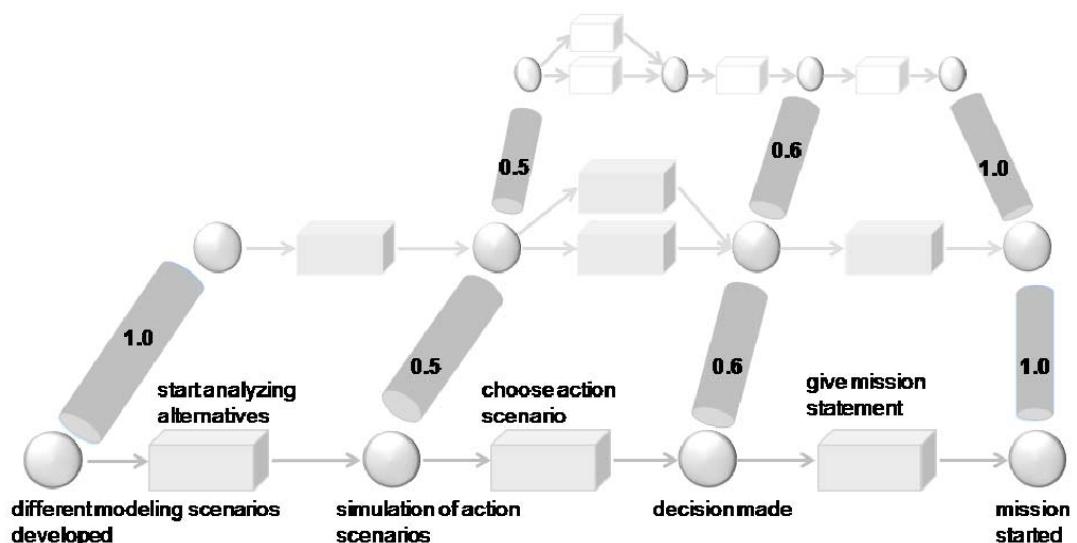


Abbildung 3.11: Visualisierung von Ähnlichkeiten zwischen Prozessmodellen aus [Bet+08]

Außerdem könnten hierarchische Prozessdiagramme gut im dreidimensionalen Raum dargestellt werden. Der Benutzer könnte mehrere Verfeinerungsstufen des Modells in einer Ansicht sehen, wie in Abbildung 3.12 gezeigt wird.

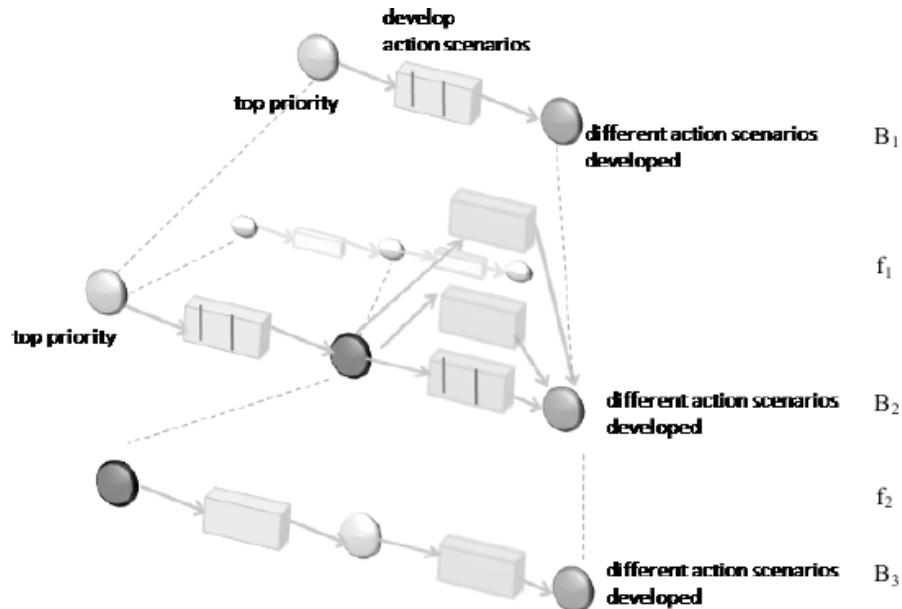


Abbildung 3.12: Vier Verfeinerungsstufen eines Prozessmodells aus [Bet+08]

3.2.2 3D-Visualisierung für die Prozesssimulation

In [SBE00] wird ein Prototyp einer interaktiven 3D-Umgebung vorgestellt, der dafür genutzt werden könnte, Simulationen von Prozessen zu kontrollieren und dabei anfallende Daten zu visualisieren. Der Prozess selbst wird, wie in Abbildung 3.13 gezeigt, als 3D-Graph dargestellt, wobei Subgraphen durch den Benutzer nach Bedarf auf- und zugeklappt werden könnten. Datenflüsse würden durch animierte Kugeln angezeigt, die sich entlang der Kanten von einem Aktivitätsknoten zum nächsten bewegen würden. Der Anwender könnte durch die Auswahl von Knoten und dem Drücken einer „drill-down-Schaltfläche“ eine Visualisierung zugehöriger Prozessdaten öffnen – hier im Beispiel 3D-Histogramme – wie in Abbildung 3.14 zu sehen ist. Im Beispiel zeigen die 3D-Histogramme eine Häufigkeitsverteilung (horizontal) von „Wartezeiten“ im Laufe von vier Wochen. Es sei möglich, Ansichten auf den Prozessgraphen zu speichern, um später wieder schnell zu diesen zurückzuspringen zu können.

3.2.3 Modellierung von Prozessen in interaktiven, virtuellen 3D-Umgebungen

In [Bro10] wird ein Prototyp eines BPMN-Editors vorgestellt, der Prozesse innerhalb einer virtuellen 3D-Umgebung darstellt. Besonderer Wert sei auf die Zusammenarbeit zwischen mehreren Modellierern und die Prozesskommunikation – auch unter Beteiligung von Personen, die keine Modellierungsexperten sind – gelegt worden. „Naive stakeholders“ hätten oft Probleme, die abstrakte Welt der konzeptuellen Modellierung zu verstehen, weil der Bezug zu realen Gegenständen fehle. Unter Zuhilfenahme einer „virtuellen Welt“ (virtual reality), in welche abstrakte Prozessmodelle eingebettet sind, solle dies abgemildert werden.

In dieser Umgebung könnten Abbilder von realen Entitäten, die mit dem Prozess in Beziehung stehen oder mit diesem interagieren – beispielsweise verwendete Betriebsmittel oder ausführende Personen – dargestellt werden. Dies könnte auch dazu dienen, den Ort und die räumliche Anordnung von Prozessschritten, beispielsweise durch die Einbettung in ein virtuelles Gebäude, zu visualisieren. Möglich sei auch eine Simulation der Prozessausführung in der virtuellen Welt. Dadurch solle es den Beteiligten

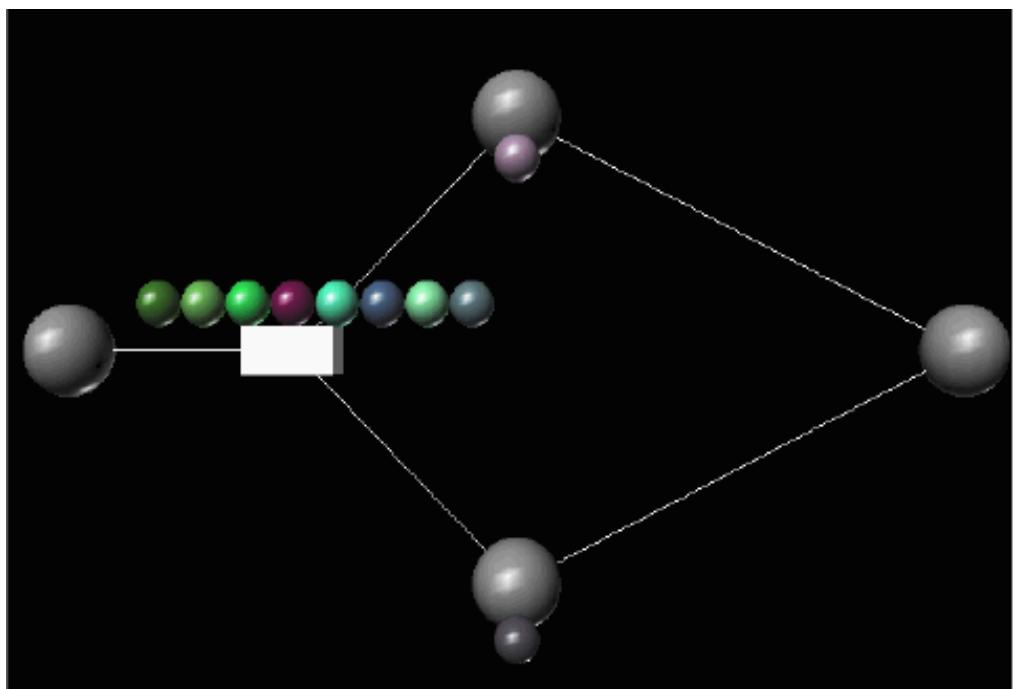


Abbildung 3.13: Prozessgraph mit „Datenflusskugeln“ aus [SBE00]

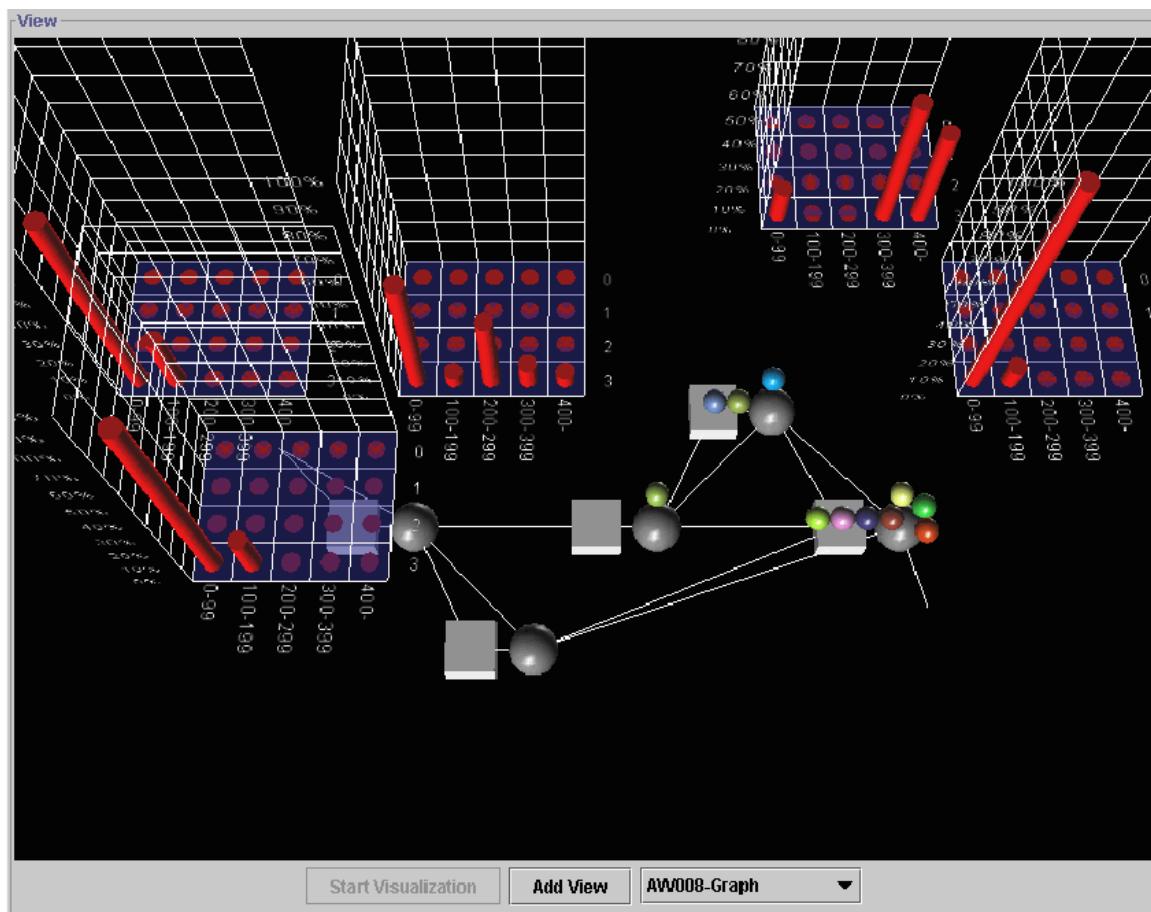


Abbildung 3.14: Darstellung eines Prozesses mit assozierten Daten (3D-Histogramme) aus [SBE00]

leichter möglich sein, festzustellen, ob das Modell die Realität richtig abbilde und ob eventuell Probleme bei der Umsetzung des Prozesses in der Realität auftreten könnten.

Wie in Abbildung 3.15 zu sehen ist, werden Prozesse als 3D-Graph dargestellt, wobei als Knoten auf 3D-Objekte übertragene BPMN-Modellelemente genutzt werden. Auf den Knoten können Informationen durch Texte oder statische Grafiken vermittelt werden.

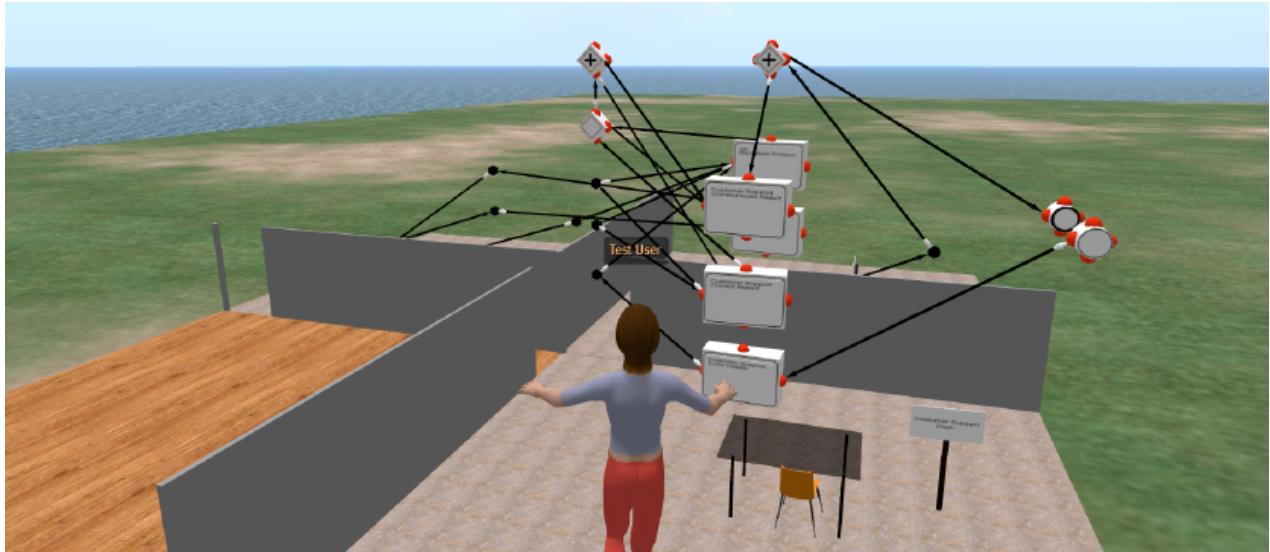


Abbildung 3.15: BPMN-Prozessgraph in virtueller Welt aus [Bro10]

Informationen auf den Objekten scheinen nur auf einer Seite dargestellt zu sein. Das ist problematisch, falls Modellelemente gedreht und Bewegungen um den Prozessgraphen herum ausgeführt werden. Je nach Perspektive wäre es möglich, dass die Texte bzw. die Symbole nicht mehr sichtbar sind. Abbildung 3.15 zeigt auch, dass die gegenseitige Verdeckung von Modellelementen ebenfalls zu Schwierigkeiten bei der Lesbarkeit der Informationen führt. Die Benutzer selbst werden, wie in Abbildung 3.16 zu sehen ist, als Avatar gezeigt, welcher die Interaktion der Benutzer mit dem Modell für andere Teilnehmer zeigen soll.



Abbildung 3.16: Benutzer-Avatar vor 3D-BPMN-Elementen aus [Bro10]

Es gebe die Möglichkeit, „Kommentarwände“ zur Anzeige von Texten für die Kommunikation zwischen den Beteiligten aufzustellen. Daneben könnten auch andere Multimedia-Inhalte wie Videos, Tonaufnahmen oder Statistiken zur Prozessausführung (über Web-Services) eingebettet werden. Dies ist in Abbildung 3.17 zu sehen.

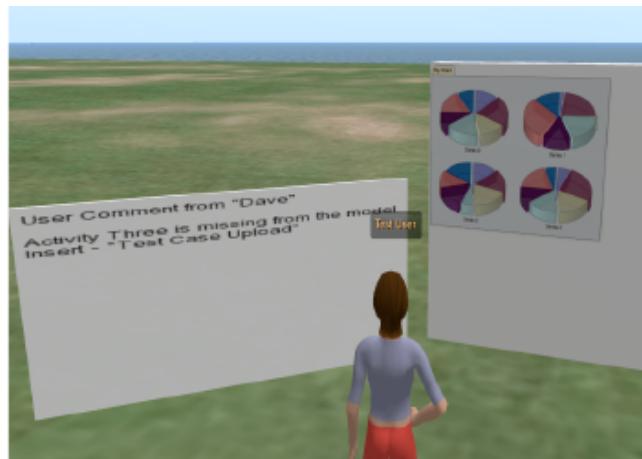


Abbildung 3.17: Kommentarwände und Multimedia-Inhalte in der virtuellen Welt aus [Bro10]

3.3 Verbesserung der dreidimensionalen Darstellung von Graphen

3D-Szenen werden auf üblichen Arbeitsplatz-Bildschirmen zu einer zweidimensionalen Projektion reduziert [AHH08]. Dies bedeutet, dass die Vorteile einer dreidimensionalen Darstellung, welche sich aus der Tiefenwirkung ergeben, nicht vollständig zur Geltung kommen. Um dies zu umgehen lassen sich Techniken wie die Stereoskopie, Bewegungsparallaxe³ oder voll immersive virtuelle Umgebungen (oft als CAVE bezeichnet⁴) einsetzen.

3.3.1 Nutzung von 3D-Effekten für einen verbesserten Tiefeneindruck

In [WM08] wird an Probanden untersucht, wie groß die Vorteile einer stereoskopischen 3D-Darstellung von umfangreichen Graphen im Vergleich zu einer 2D-Darstellung sind. Als Maß für die „Lesbarkeit“ wird hier das Abschneiden bei der Aufgabe, die Pfadlänge zwischen zwei markierten Knoten zu erkennen, genutzt.

Stereoskopische 3D-Darstellung sei besonders hilfreich, um dem Betrachter einen realistischen Tiefeneindruck zu vermitteln und damit das Erkennen von Verbindungen zu erleichtern. Eine weitere Maßnahme, um den Tiefeneindruck zu verbessern, sei es, den Graphen ständig zu rotieren und damit die Bewegungsparallaxe zu nutzen². Es zeigte sich, dass die Probanden – bei gleicher Fehlerrate – Verbindungen in 3D-Graphen erkennen hätten können, welche um eine Größenordnung größer gewesen seien als die entsprechenden 2D-Graphen.

Dabei sei eine Anzeige mit einer sehr hohen Auflösung verwendet worden, die nahe an das Auflösungsvermögen des menschlichen Sehsystems herankomme. Eine frühere Untersuchung mit ähnlicher Konzeption [WF96] zeigte deutlich kleinere Vorteile für die stereoskopische 3D-Darstellung. Dies wird in der späteren Arbeit auf den Umstand zurückgeführt, dass hierbei Anzeigen mit einer viel niedrigeren Auflösung verwendet worden seien.

3.3.2 3D-Visualisierung von Graphen in voll immersiven virtuellen Umgebungen

Neben der Anzeige von 3D-Graphvisualisierungen auf handelsüblichen Arbeitsplatz-Rechnern könnten dafür auch immersive 3D-Umgebungen (fully immersive virtual reality) genutzt werden.

³Näheres zu Wahrnehmung von Tiefe siehe [WTS89], [wp:bewegungsparallaxe] oder [wp:stereoskopie].

⁴Näheres zu CAVE-Systemen siehe [CSD93] oder [wpe:cave].

So zeigt [Hal+08] die Visualisierung von sozialen Netzwerken in einer CAVE-Umgebung. Benutzer könnten so direkt mit der Graphdarstellung der Daten in einer natürlichen Art und Weise interagieren und einen realitätsnahen räumlichen Eindruck von der virtuellen Welt bekommen.

Der Graph würde zu Beginn in einer „2D-Darstellung“ in einer Ebene vor dem Benutzer angezeigt, wie in der Abbildung 3.18 (unten) zu sehen ist. Links ist zu sehen, wie durch das „Berühren“ mit einem virtuellen Werkzeug (grauer Quader) die mit dem Knoten assoziierten Daten angezeigt werden können.

Wenn sich ein Benutzer speziell für die Verbindungen eines bestimmten Knoten interessiere, sei es möglich aus dieser Darstellung den gewünschten Knoten zu „extrudieren“, also zu sich heranzuziehen. Wie in Abbildung 3.18 (rechts) zu sehen ist werden dadurch die Verbindungen des Knotens hervorgehoben.

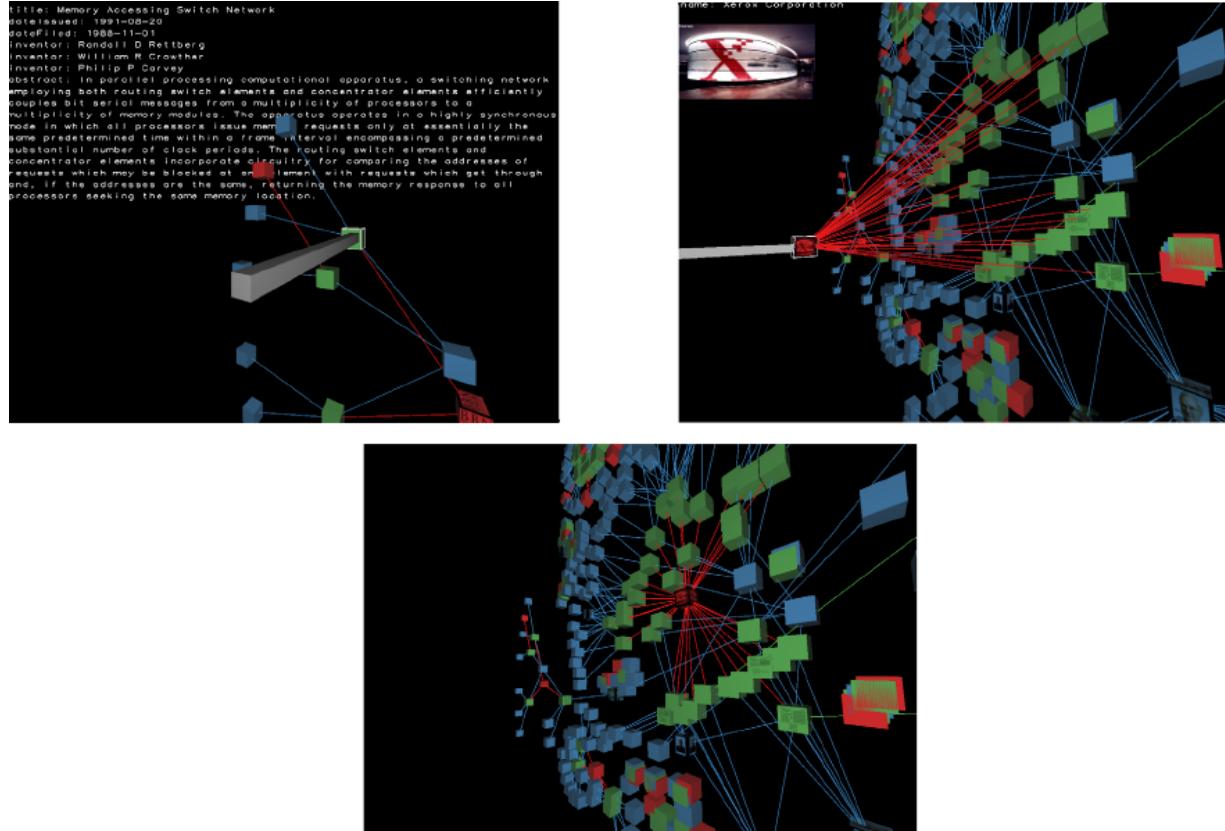


Abbildung 3.18: Visualisierung von semantischen Netzwerken aus [Hal+08]

3.4 Zusammenfassung und Bewertung

Es wurden verschiedene Ansätze gezeigt, zu einer 3D-Visualisierung von Informationen zu gelangen und deren Vorteile zu nutzen. So lässt sich häufig der Ansatz beobachten, von einer bekannten 2D-Visualisierung auszugehen und diese in den 3D-Raum zu übertragen. Dies war besonders bei den verschiedenen Arbeiten zu sehen, die sich mit 3D-UML beschäftigen.

Der vorliegende Abschnitt fasst die wichtigsten Nutzungsmöglichkeiten der dritten Dimension zusammen, die in den gezeigten Arbeiten vorgeschlagen wurden. Außerdem wird versucht, eine Einschätzung zu geben, wie vorteilhaft die gezeigte (3D-)Visualisierung im Vergleich zu einer reinen 2D-Darstellung ist. Die folgende Tabelle gibt einen Überblick über die vorgestellten Verwendungsmöglichkeiten und deren möglichen Nutzen.

Name	Ansatz	Domäne	3. Dimension verwendet für	Nutzen
McIntosh	2,5D	UML(Zustand)	Visualisierung von Modellhierarchien	o
Gil	2,5D	UML(Sequenz)	Zeitachse, unterschiedl. Beziehungstypen	+
Gogalla	3D	UML(Klassen)	„Wichtigkeit“ von Knoten	+
Schönhage	3D	Prozesssimulation	Darstellung von Ausführungsdaten	o
Betz	3D	Prozesse(Petrinetz)	mehrere Modelle/-typen, hierarchische Modelle	+
Brown	3D	Prozesse(BPMN)	Einbettung in virtuelle Umgebung	++

3.4.1 2,5D / 3D-Visualisierungen von Modellen

Eine naheliegende Möglichkeit ist es, schon bekannte 2D-Modellierungssprachen wieder zweidimensional auf Flächen im 3D-Raum zu platzieren. Dies wurde von *McIntosh* (Unterabschnitt 3.1.2) für UML-Zustandsdiagramme, von *Gil und Kent* (Unterabschnitt 3.1.4) oder bei *GEF3D* (Unterabschnitt 3.1.6) gezeigt. In der Tabelle werden solche Darstellungsformen als 2,5D-Ansatz bezeichnet. Für die Implementierung bedeutet das, dass sich schon vorhandene 2D-Bibliotheken nutzen lassen, deren Grafikausgabe direkt auf die Flächen gezeichnet wird. Für den Benutzer hat die Darstellung den Vorteil, dass sich die Darstellung der Modellelemente selbst nicht ändert und sich mehrere Modelle gleichzeitig darstellen lassen, indem die Ebenen zueinander versetzt werden.

Wie von *Gil und Kent* (Unterabschnitt 3.1.4) erwähnt, lassen sich durch eine dreidimensionale Anordnung verschiedene Beziehungstypen besser unterscheiden als in reinen 2D-Ansichten. Modellhierarchien und Beziehungen zwischen verschiedenen Modellen lassen sich gut darstellen, indem beispielsweise Linien zwischen assoziierten Elementen oder zu Unterdiagrammen gezeichnet werden. Diese Beziehungen lassen sich schon durch die Anordnung optisch leicht von denjenigen unterscheiden, die innerhalb eines (Teil-)Modells bestehen und in einer Ebene mit den Modellelementen liegen. In reinen 2D-Darstellungen ist diese Unterscheidung deutlich schwieriger und es muss üblicherweise auf unterschiedliche Farben oder Konnektoren zurückgegriffen werden. Vor allem bei einer großen Anzahl von Elementen kann dies leicht zu verwirrenden Darstellungen führen.

Problematisch ist dagegen bei 2,5D-Ansätzen, dass es bei „schrägen“ Betrachtungswinkeln schwierig wird, Informationen abzulesen, was sich besonders bei Schriftdarstellung bemerkbar machen wird. Außerdem ist bei den bisher genannten Visualisierungsformen die Möglichkeit eingeschränkt, die dritte Dimension zur Vermittlung von zusätzlichen Informationen zu nutzen, da Elemente immer auf festen Ebenen platziert werden müssen. Kontinuierliche Attribute der Modellelemente lassen sich so nicht darstellen.

Als Weiterentwicklung lässt sich die von *Krolovitsch und Nilsson* (Unterabschnitt 3.1.3) vorgestellte Visualisierung von Zustandsdiagrammen betrachten, die ebenfalls 2D-Flächen nutzt, jedoch die Elemente aus der Ebene herausragen lässt. So wirkt die Darstellung etwas „plastischer“ und Strukturen lassen sich besser erkennen. Die Höhe der Elemente lässt sich hier nutzen, um Werte von kontinuierlichen Modellattributen direkt zu visualisieren.

Interessant ist die dort gezeigte Möglichkeit, Subdiagramme temporär auszublenden und durch ein einzelnes Symbol zu ersetzen. Dies wäre auch in der Prozessmodellierung hilfreich für die Darstellung von kompositen Prozessen. So kann beispielsweise durch einen Doppelklick auf einen Prozessknoten ein weiteres Modell in der 3D-Szene angezeigt werden ohne ein neues Fenster zu öffnen, wie es in 2D-Werkzeugen praktiziert wird.

Betz et al. (Unterabschnitt 3.2.1) zeigten für den Bereich der Prozessmodellierung die schon für die Softwaremodellierung genannten Nutzungsmöglichkeiten des 3D-Raums, also die hierarchische Darstellung von Prozessdiagrammen und die Visualisierung von Beziehungen zwischen unterschiedlichen Modellarten.

Von *Gogolla et al.* (Unterabschnitt 3.1.5) wurden UML-Diagramme mit „echten“, frei plazierbaren 3D-Objekten gezeigt. Die dritte Dimension („Tiefe“) lässt sich dazu nutzen, schon durch Verbindungen festgelegte Zusammenhänge zu verdeutlichen oder um Attribute der Modellelemente zu visualisieren.

3D-Objekte wie Quader haben den Vorteil, dass sich Information — oft in Textform — auf mehreren Seiten darstellen lässt. Wie von *Dwyer* (Unterabschnitt 3.1.1) vorgeschlagen, gibt es die Möglichkeit, diese Objekte so zu drehen, dass dem Benutzer immer eine Seite zugewandt und damit gut lesbar ist. Damit lassen sich solche Modelle besser aus unterschiedlichen Perspektiven betrachten als die vorgenannten 2,5D-Darstellungen. Der Wechsel der Perspektive kann hilfreich sein, um unterschiedliche Aspekte der 3D-Szene gezielt betrachten zu können. Beispielsweise können so Beziehungen zwischen Elementen auf parallelen Ebenen herausgestellt werden, indem die Szene „von der Seite“ betrachtet wird.

3.4.2 Integration von weiteren Informationen in 3D-Visualisierungen

Neben dem abstrakten Prozessmodell an sich lassen sich auch weitere Informationen dreidimensional darstellen. So zeigten *Schönhage et. al.* (Unterabschnitt 3.2.2), wie sich aus einer Simulation des Prozesses gewonnene Daten neben dem Prozessmodell anzeigen lassen. Dies ist aber prinzipiell mit reinen 2D-Darstellungen ebenfalls möglich. Um hier einen klaren Vorteil der 3D-Visualisierung erkennen zu können, müssen sich die Daten selbst sinnvoll dreidimensional darstellen lassen. Ein Beispiel dafür sind die 3D-Histogramme, wie sie von Schönhage gezeigt wurden.

Brown (Unterabschnitt 3.2.3) bettet die abstrahierte Darstellung des Prozesses in eine virtuelle Umgebung ein, welche den tatsächlichen Ausführungsort eines Prozesses räumlich abbilden kann. Dies lässt sich als deutlicher Vorteil für die Nutzung von 3D-Visualisierungen im Vergleich zu 2D-Darstellungen festhalten.

Beispielsweise lassen sich Laufwege von am Prozess beteiligten Personen oder andere Vorgänge wie der Transport von Werkstücken (animiert) darstellen, um mögliche Probleme bei der Ausführung und Optimierungsmöglichkeiten aufzuzeigen. So kann festgestellt werden, ob sich gewisse Wege verkürzen oder vermeiden lassen, indem Reihenfolge oder der Ausführungsort von Prozessschritten verändert werden. Durch die Integration von Abbildern realer Objekte in die virtuelle Welt können abstrakte Konzepte des Prozessmodells veranschaulicht oder um weitere Informationen ergänzt werden. Sinnvoll ist dies beispielsweise, um Veränderungen an einem Werkstück im Laufe eines Produktionsprozesses zu visualisieren, indem die Zwischenstufen dreidimensional neben den Prozessschritten abgebildet werden.

Eine andere denkbare Anwendung wäre eine Visualisierung der Platzverhältnisse in einer Ausführungs-umgebung. Wenn die Umgebung sowie sich darin befindliche Objekte relativ zueinander im richtigen Größenverhältnis und in der tatsächlichen Form dargestellt sind, könnte schon bei einer Betrachtung des Prozesses in der virtuellen Welt bemerkt werden, dass vorgesehene Ablageplätze in einem Lager oder Transportbehälter für ein Werkstück zu klein dimensioniert sind.

Abbildung 3.19⁵ zeigt einen BPMN-Prozess in einer 3D-Umgebung, die mit Hilfe des von Brown vorgestellten Editors erstellt wurde und einen Flughafen darstellen soll. Dies ließe sich prinzipiell auch mit einer 2D-Darstellung realisieren, indem die Szene von oben gezeigt wird. Dadurch wird aber die Übersichtlichkeit eingeschränkt; die Möglichkeit, den Prozessgraphen im dreidimensionalen Raum in einer Ebene über dem Boden zu zeigen, erweist sich hier als Vorteil. Ebenso bietet es sich an, im 3D-Raum die Betrachtungsperspektive nach Bedarf zu verändern. Ist die Blickrichtung des Betrachters annähernd parallel zum Untergrund wie in der Abbildung, lassen sich auch weit entfernte „Stationen“ des Prozesses erkennen. Ein Blick von oben auf die Szene aus größerer Entfernung gibt dagegen einen groben Überblick über die gesamte Prozessstruktur.

⁵Quelle: www.youtube.com/watch?v=aUBmvykDhB0. Das Video ist auch auf der beigelegten *DVD* (Anhang B) zu finden.



Abbildung 3.19: Visualisierung eines BPMN-Prozesses in einer virtuellen Umgebung

3.4.3 Effizienz und Akzeptanz von 3D-Darstellungen?

Die wichtige Frage, ob und in welchen Situationen 3D-Visualisierungen Vorteile gegenüber 2D-Darstellungen haben, die über eine reine Verbesserung des Erscheinungsbilds hinausgehen, kann von den gezeigten Arbeiten sicher nicht vollständig beantwortet werden. Es wurden immerhin einige Hinweise zur Effizienz gegeben, indem beispielsweise Benutzerstudien durchgeführt wurden, welche Vorteile für 3D-Darstellungen in der Softwaremodellierung andeuten, jedoch auch Probleme aufzeigen [Dwy01] [MHS08] [Hal+08]. Untersuchungen zur Effizienz, die sich speziell auf die Prozessmodellierung beziehen, ließen sich nicht finden.

Inwieweit 3D-Darstellungen für die Prozessmodellierung nützlich sind, hängt sicher auch davon ab, wie komplex die Modelle sind. Einfache Modelle lassen sich auch mit den Hilfsmitteln der 2D-Visualisierung adäquat darstellen. Prinzipiell ist eine 2D-Darstellung von Modellen wohl für die meisten Benutzer etwas intuitiver zu verstehen, da diese schon länger verbreitet und beispielsweise aus UML- oder BPMN-Werkzeugen bekannt ist. Wie erwähnt, werden die Vorteile von 3D-Darstellungen dann deutlich, wenn es darum geht, viele Verbindungstypen darzustellen oder hierarchische Modelle in einer Szene darzustellen. Ob 3D-Visualisierungen effizient sind und ob sich deren höhere Komplexität – sowohl für den Benutzer als auch für die Implementierung – auszahlt muss daher sicher abhängig vom konkreten Anwendungsfall bewertet werden.

Bei der Betrachtung der Effizienz muss auch berücksichtigt werden, dass die Erfahrung der Benutzer mit 3D-Darstellungen aus anderen Bereichen – beispielsweise aus Computerspielen oder 3D-CAD-Werkzeugen – eine Rolle spielt [Dwy01] [WM08] [SBE00]. Es stellt sich die Frage, wie unerfahrene Benutzer an 3D-Werkzeuge für die Prozessmodellierung herangeführt werden könnten, um diese effizient nutzen zu können.

In eine ähnliche Richtung geht die Fragestellung, inwieweit 3D-Werkzeuge überhaupt von Benutzern akzeptiert werden. [SBE00] bemerkte, dass 3D-Visualisierungen oft als reines „Spielzeug“ angesehen würden, die keinen wirklichen Nutzen gegenüber 2D-Darstellungen brächten, sondern bestenfalls nur „schöner“ aussähen. Daher ist es wichtig, 3D-Visualisierungen für den Benutzer möglichst hilfreich und intuitiv zu gestalten, so dass deren Vorteile klar erkennbar sind. Um eine hohe Akzeptanz zu

erreichen müssten aber auch technische Probleme wie die schlechte Verfügbarkeit von 3D-tauglichen Eingabegeräten oder zu langsame Hardware⁶ gelöst werden. So ergeben sich große Herausforderungen, sowohl auf der konzeptuellen Ebene als auch auf der Seite der Implementierung von 3D-Visualisierungen und -Modellierungswerkzeugen.

3.4.4 Verwendbare Vorarbeiten und Schlussfolgerungen für die Arbeit

In den vorgestellten Arbeiten wurden einige Prototypen für 3D-Modellierungswerkzeuge entwickelt. Allerdings war nur von [Dwy01] eine freie Version im Internet auffindbar. Diese ist allerdings technisch auf einem ziemlich alten Stand und lässt in Sachen Bedienung eher zu wünschen übrig.

Frei verfügbare Softwareprojekte, die schon ein flexibles 3D-Prozessmodellierungswerkzeug realisieren ließen sich nicht finden. Als Grundlage für ein solches Werkzeug könnte möglicherweise GEF3D dienen, was jedoch nicht weiter verfolgt wurde. Negativ könnte bei GEF3D gesehen werden, dass in letzter Zeit relativ wenige Änderungen an der Codebasis erfolgten und insgesamt eher wenig Aktivität festzustellen ist.

Ein Blick in den Quellcode zeigte, dass das Projekt noch auf „alter“ OpenGL-Funktionalität aufbaut und damit die Möglichkeiten moderne Grafikhardware nicht nutzt. Bei der vorliegenden Arbeit stand es aber im Vordergrund, eine möglichst flexible und „zukunftsorientierte“ Grundlage für ein (anpassbares) Prozessmodellierungswerkzeug zu legen, wozu auch eine Grafikausgabe auf dem aktuellen Stand der Technik gehört.

Aus den hier vorgestellten Arbeiten ließen sich jedoch einige Konzepte ableiten, die in i>PM 3D realisiert wurden. Es ist für die Arbeit sinnvoll, einen möglichst allgemeinen Visualisierungsansatz zu wählen, der es erlaubt, verschiedene Nutzungsmöglichkeiten der dritten Dimension umzusetzen und diese für verschiedene Anwendungsfälle im Prototypen zu evaluieren. Daher soll der Prototyp grundsätzlich eine 3D-Graphvisualisierung von Prozessen unterstützen, wie es beispielsweise von *Brown* (Unterabschnitt 3.2.3) gezeigt wurde. (*Anforderung (a)* (Abschnitt 1.3)). Die Knoten des Graphen werden selbst dreidimensional dargestellt und sind frei in der 3D-Szene platzierbar. Dies stellt prinzipiell den flexibelsten Ansatz dar. Eine 2,5D oder gar 2D-Darstellung lässt sich als Sonderfall einer 3D-Darstellung betrachten. So kann die Darstellung oder die Platzierbarkeit von Elementen – ausgehend vom gewählten 3D-Ansatz – wieder eingeschränkt werden, falls sich dies für eine Anwendung als vorteilhaft herausstellen sollte.

Abstrakte Prozessmodelle in eine virtuelle Welt einzubetten und so räumliche Information zu nutzen stellt einen vielversprechenden Ansatz dar, aus dreidimensionalen Darstellungen einen großen Vorteil zu ziehen. Dadurch wird *Anforderung (b)* motiviert, beliebige 3D-Objekte in die Szene einzufügen zu können, um damit abstrakte Modellelemente zu illustrieren oder virtuelle Ausführungsumgebungen visualisieren zu können.

⁶Die besagte Arbeit ist 2000 entstanden. Sicherlich ist die Geschwindigkeit heutzutage ein kleineres Problem, aber es lässt sich nicht vernachlässigen. Gerade bei aufwändigen Grafikeffekten oder der Verarbeitung von komplexen Eingabedaten kann man leicht an die Grenzen der Rechenleistung stoßen.

Verwendete Techniken und Software

4.1 Scala

Die Implementierung des i>PM3D-Projekts erfolgte zum größten Teil in der Programmiersprache Scala [OSV11] [[www:scala](http://www.scala-lang.org)]. Die Verwendung von Scala ergab sich aus der Entscheidung, die in Scala implementierte Simulations-Middleware *Simulator X* (Abschnitt 4.2) als Basis für den Prototypen zu verwenden.

Scala wird als „objektfunktionale“ Programmiersprache charakterisiert. „Objektfunktional“ soll die Bestrebungen ausdrücken, Aspekte aus funktionalen und objektorientierten Programmiersprachen zu einer flexiblen und effektiven Programmiersprache zu kombinieren.

Scala wird zur Zeit vorwiegend auf der Java VM genutzt, wobei der Compiler auch in der Lage ist, CIL-Code für die .NET-Runtime zu erzeugen. i>PM3D läuft wegen einiger Abhängigkeiten von Java-Bibliotheken bisher ausschließlich auf der Java VM.

Die „objektorientierte Facette“ Scalas orientiert sich an den Konzepten von Java, bietet aber einige Erweiterungen. Hier werden nur Features kurz vorgestellt, die für die Implementierung besonders hilfreich waren und in späteren Kapiteln erwähnt werden.

4.1.1 Traits

Als Erweiterung zu Java unterstützt Scala eine (eingeschränkte) Mehrfachvererbung von Implementierungscode über sog. *Traits*. Traits kann man sich als ein Java-Interface vorstellen, in dem Methoden schon vorimplementiert sein können. Zur Vereinfachung dürfen Traits keinen Konstruktor definieren.

Neben der Verwendung als „Interface“ – wie in Java – werden diese oft genutzt, um wieder verwendbare Code-Einheiten zu realisieren, die sich in verschiedenen Klassen einsetzen lassen. Traits werden daher oft als **Mixin** bezeichnet. Wie ein Trait zu einer Klasse hinzugefügt wird – auch **einmischen** genannt – zeigt das folgende Scala-Codebeispiel:

```
class Example extends BaseClass with MixinTrait
```

Example wird hiermit von BaseClass abgeleitet und MixinTrait eingemischt.

In dieser Arbeit werden Traits in UML-Diagrammen als Klasse mit dem Stereotyp `<<trait>>` dargestellt und Assoziationen zu einmischenden Klassen mit `<<mixin>>` versehen.

4.1.2 Objects

Anstelle der aus Java bekannten statischen Klassenmethoden oder Singleton-Klassen wird in Scala das *object*-Konstrukt genutzt. *Objects* können Klassen „erweitern“, das bedeutet, dass das *Object* als Instanz der Klasse betrachtet werden kann. Als Beispiel sei hier ein ausführbares Scala-Programm gezeigt, welches eine (im Sinne von Java statische) Methode definiert und aufruft:

```
object Main extends App {  
    def hello() {  
        println("Hello World")  
    }  
    hello()  
}
```

4.1.3 Actors

Ein sinnvoller Einsatzbereich von Scala ist unter anderem die Erstellung von parallelen und verteilten Anwendungen. Dazu kommt oft das **Actor-Modell** [HO09] zum Einsatz, das früher schon in der Programmiersprache Erlang [[www:erlang](http://www.erlang.org)] realisiert wurde.

Grundlage für das Actor-Modell ist das **message passing**, welches eine asynchrone Kommunikation zwischen den beteiligten Actors ermöglicht. Berechnungen innerhalb einzelner Actors können so prinzipiell parallel erfolgen. Informationen werden ausschließlich über Nachrichten ausgetauscht. Es ist nicht erlaubt, auf gemeinsame, veränderliche Datenstrukturen zuzugreifen. Actors können auf unterschiedlichen (Java-)Threads ausgeführt werden und somit auch mehrere Prozessorkerne nutzen, ohne den Programmierer mit der manuellen Verwaltung und Synchronisation von Threads zu belasten.

In Scala wird eine Nachricht oft durch ein Objekt einer **case class** dargestellt¹. Diese Klassen werden dafür genutzt, Daten zu unveränderlichen Objekten zusammenzufassen, wie im folgenden Code gezeigt wird:

```
case class Message(data: String, number: Int)  
receivingActor ! Message("hello!", 42)
```

In der zweiten Zeile wird ein Objekt der Klasse `Message` erzeugt und an `receivingActor` gesendet.

4.1.4 Implizite Methoden

Es ist möglich, sog. „implizite Methoden“ zu definieren, welche vom Compiler automatisch eingesetzt werden können, wenn diese benötigt werden². Besonders praktisch sind diese Methoden für die Realisierung von „transparenten“ Adaptern, wie sie im vorliegenden Projekt genutzt werden. Diese werden auch **implizite Wrapper** genannt.

```
implicit def conceptToAdapter(m: MConcept) = new MConceptAdapter(m)
```

Mit dieser Definition lassen sich nun Methoden, die für `MConceptAdapter` definiert sind auch auf Objekten des Typs `MConcept` aufrufen als wären sie Teil von `MConcept`.

¹Das *case class*-Konstrukt erzeugt eine Klasse, in der gewisse Methoden vorimplementiert sind, die bspw. einen inhaltlichen Vergleich mit dem `==`-Operator oder einen Einsatz im *pattern matching* erlauben. Siehe [OSV11].

²Welche Bedingungen dafür erfüllt sein müssen, kann bspw. in [OSV11] nachgelesen werden.

4.1.5 Parser-Kombinatoren

Die Scala-Standardbibliothek bietet eine einfache Möglichkeit, Parser mit Hilfe von Parser-Kombinatoren [OSV11] zu erstellen. Dies wird in dieser Arbeit für die Laden von Modellen in einer textuellen Repräsentation eingesetzt.

Einfache Parser werden von Parser-Kombinatoren zu komplexeren Parsing-Ausdrücken zusammengesetzt. Parser sind als Funktionen definiert, die einen String auf eine beliebige Ausgabe abbilden. Parser-Kombinatoren sind Funktionen höherer Ordnung, die Parser als Eingabe erwarten und als Ausgabe wiederum eine Parser-Funktion liefern.

In Scala werden die Bestandteile der textuellen Eingabe oft in Objekte von *case classes* übersetzt, die zusammen einen Syntaxbaum der Eingabe ergeben.

Folgende Parser-Funktion

```
def stringAssignment = ident ~ ("=" ~> stringLits <~ ";") ^^ {  
    case id ~ stringLits => LiteralTypeAssignment(id, stringLits)  
}
```

würde beispielsweise die LML-String-Zuweisung

```
functions = "a", "test";
```

erkennen und in ein Scala-Objekt des Typs `LiteralTypeAssignment` übersetzen. Dieser Typ könnte wie folgt definiert sein:

```
case class LiteralTypeAssignment(id: String, stringLiterals: List[String])
```

4.2 Simulator X

Simulator X [LT11] [Fis+11] ist ein Prototyp einer neuartigen Simulationsplattform, welche die Realisierung von interaktiven Echtzeit-Anwendungen besonders einfach machen soll. Der Fokus liegt dabei auf Anwendungen aus der Computergrafik, besonders in Verbindung mit multimodalen Bedienschnittstellen, welche neuartige Eingabemethoden wie Gesten- und Sprachsteuerung nutzen.

Simulator X setzt auf dem (*Scala-Actor-Modell*) (Unterabschnitt 4.1.3) auf und bietet daher dessen Eigenschaften wie die Ausnutzung mehrerer Prozessorkerne und eine asynchrone Kommunikation zwischen Actors. Sog. **Komponenten** (Component) sind die grundlegenden Bestandteile einer Simulator X-Anwendung. Komponenten sind als Actor realisiert und stellen eine wohldefinierte Funktionalität für das System zur Verfügung.

Von Simulator X wird eine Reihe von Komponenten bereitgestellt, beispielsweise die für i>PM3D verwendete Gestenerkennung und eine *Physikkomponente* für physikalische Simulationen (bspw. für Gravitation und Erkennung von Kollisionen zwischen Simulationsobjekten). Andere, typischerweise für Simulator X-Anwendungen verwendete Komponenten umfassen Renderkomponenten für die Realisierung der Grafikausgabe oder Komponenten zur Beeinflussung der Simulationsobjekte durch künstliche Intelligenz.

Simulator X bietet – ebenfalls auf Basis des Actor-Modells – ein Event-System und eine Abstraktion globaler Zustandsvariablen an. Komponenten können sich beim Event-System für bestimmte Ereignisse, die von anderen Komponenten ausgelöst werden, registrieren und so automatisch benachrichtigt werden.

Globale Zustandsvariablen, **SVars** genannt, vereinfachen für den Programmierer den Umgang mit verteilten Daten. Ein bestimmtes Datum wird von genau einem Actor, dem „Besitzer“ verwaltet. Andere Actors besitzen nur eine spezielle Referenz auf den Wert und müssen mit dem Besitzer kommunizieren,

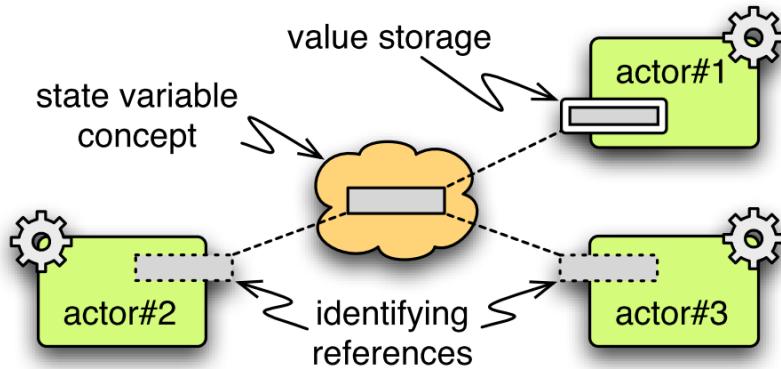


Abbildung 4.1: Zustandsvariablen-Konzept aus [LT11]

um den Wert auszulesen oder zu manipulieren. Diese Kommunikation wird von Simulator X automatisch und transparent für den Programmierer durchgeführt. Abbildung 4.1 zeigt ein Beispiel, in welchem actor#1 der Besitzer der SVar ist und die beiden anderen Actors nur Referenzen auf diese SVar halten.

Eine zugeordnete SVarDescription benennt die SVar, gibt ihr einen Scala-Datentyp und definiert deren Semantik in einer Anwendung. So lässt sich beispielsweise definieren, dass der Wert einer SVar eine Farbe darstellt, welche durch eine Klasse Vec4 repräsentiert wird.

Eine Menge von SVars ergibt zusammen eine **Entität**, die genau ein Simulationsobjekt repräsentiert³. So kann durch die Manipulation der Color-SVar einer bestimmten Entität deren Farbe festgelegt werden.

Entities werden durch den Programmierer mittels einer EntityDescription beschrieben, die aus mehreren Aspect-Definitionen aufgebaut sein kann [WL12]. **Aspects** beschreiben eine wohldefinierte Facette der Entität und sind einer bestimmten Komponente zugeordnet. So gibt es beispielsweise Grafik- oder Physik-Aspects.

Ein Aspect legt fest, wie eine Komponente mit einem bestimmten Simulationsobjekt umgehen soll. Über die Aspect-Definition können Werte durch den Benutzer vorgegeben werden, die das Verhalten der Komponente im Bezug auf die zugehörige Entität festlegen. So lässt sich beispielsweise über einen Physik-Aspect festlegen, welche physikalische Repräsentation für die Entität genutzt werden soll, beispielsweise ein Quader mit bestimmten Abmessungen und einer Masse. Für eine Renderkomponente kann der Pfad zu einer 3D-Objektdatei angegeben werden, welche als grafische Repräsentation für die Simulationsobjekt auf dem Bildschirm angezeigt werden soll.

Simulator X befindet sich gerade in der Entwicklung. Für das vorliegende Projekt wird eine Version von August 2011 genutzt.

4.3 OpenGL / LWJGL

Um die Grafikausgabe von i>PM3D zu realisieren, wird die plattformunabhängige 3D-Schnittstelle OpenGL [[opengl](#)] genutzt.

Zur Anbindung an OpenGL wird die Java-Bibliothek LWJGL (Lightweight Java Gaming Library) [[www.lwjgl](#)] in der Version 2.8.2 eingesetzt. Zusätzlich stellt LWJGL eine Schnittstelle für den Zugriff auf Tastatur- und Mausdaten zur Verfügung.

Hier sollen nur einige wenige Hinweise zu „modernen“ OpenGL (ab Version 3.0) und den in späteren Kapiteln benutzten Begriffen gegeben werden. Näheres kann in [[Wri+10](#)] oder unter [[opengl](#)] nachgelesen werden. Allgemeines zu Begriffen aus der 3D-Computergrafik findet sich bei [[AHH08](#)].

³Dies könnte im Prozesseditor beispielsweise ein Modellelement wie ein Prozess oder eine Kontrollflusskante sein.

In älteren OpenGL-Versionen (1.x) wurden von OpenGL viele, fest eingebaute Funktionen wie die Berechnung der Beleuchtung und Texturierung bereitgestellt, die nur aktiviert und konfiguriert werden mussten. Deshalb wird „altes“ OpenGL oft mit dem Begriff *fixed-function-Pipeline* [AHH08] in Verbindung gebracht.

Mit Version 3.0 wurden viele dieser Funktionen aus dem Kern von OpenGL entfernt. In neueren Versionen müssen die Berechnungen durch den Programmierer selbst in *Shadern* implementiert werden. Das neue Konzept gibt jedoch dem Programmierer die Freiheit, auch völlig neue Grafikeffekte zu implementieren, die mit der *fixed-function-Pipeline* nicht oder nur schwer umsetzbar gewesen wären. Diese Möglichkeit wurde in der vorliegenden Arbeit für einige „Spezialeffekte“ genutzt, die sich auf diesem Weg einfach realisieren ließen.

Bei **Shadern** handelt es sich um kleine Programme, die in der Programmiersprache GLSL (OpenGL Shading Language) geschrieben und die direkt auf dem Grafikprozessor von sog. *Shader-Einheiten* ausgeführt werden. Code kann in GLSL in Funktionen ausgelagert und so in mehreren Shadern genutzt werden. Shader erfüllen verschiedene Aufgaben an von OpenGL festgelegten Positionen innerhalb der Render-Pipeline [www.glpipe] [AHH08].

In OpenGL 4 werden folgende Typen unterstützt:

Vertex-Shader arbeiten auf einzelnen Vertices eines 3D-Objekts ⁴ und sind beispielsweise für die Transformation von 3D-Modellkoordinaten in das von OpenGL benutzte Koordinatensystem zuständig.

Geometry-Shader können aus den gegebenen Vertices neue Zwischen-Vertices erzeugen.

Fragment-Shader werden einmal pro Fragment aufgerufen⁵ und implementieren beispielsweise Texturierung und Beleuchtung.

Tesselation-Shader (ab OpenGL 4) können komplett neue Geometrien erzeugen.

Mit **Vertex-Attributen** lassen sich beliebige Daten pro Vertex an die Shaderprogramme übertragen; häufig sind das Vertexkoordinaten⁶, Normalen⁷ und Texturkoordinaten⁸. Vertex-Attribute werden vom Shader aus Puffern im Grafikspeicher ausgelesen, welche als Vertex Buffer Objects (VBO) bezeichnet werden.

Uniforms übermitteln Werte an Shaderprogramme, die üblicherweise für ein ganzes Grafikobjekt gelten. Dies können beispielsweise Lichtparameter oder Farbwerte sein.

4.4 Sonstiges

4.4.1 StringTemplate

Um Prozessmodelle in einer textuellen Form speichern zu können, wird die Template-Bibliothek *StringTemplate* (ST) in der Version 4.0.4 verwendet. [Par09] ST folgt dem Prinzip, einen Text mit „Platzhaltern“ (Attributen) zu definieren. Die Attribute werden aus dem Anwendungsprogramm heraus gesetzt und so das Template mit Inhalt gefüllt. Diese Schicht sorgt unter anderem dafür, dass beliebige Scala-Objekte als Java-Bean an ST weitergegeben werden können, auch wenn sie selbst nicht der Java-Bean-Konvention entsprechen. In folgendem Beispiel wird ein Template erstellt, welches die LMM-Zuweisung `function = "test"` produziert:

⁴Ein Vertex ist ein „Eckpunkt“ eines 3D-Objekts, welches in OpenGL üblicherweise als ein aus Dreiecken aufgebautes Gitter beschrieben wird.

⁵Ein Fragment entspricht – vereinfacht gesagt – einem Pixel auf dem Bildschirm.

⁶Vertexkoordinaten sind die Koordinaten des Punkts im 3D-Raum. OpenGL „rendert“ ein 3D-Objekt, indem eine Liste von Vertices der Reihe nach gezeichnet wird.

⁷Normalen werden vor allem für die Berechnung der Beleuchtung benötigt.

⁸Texturkoordinaten sind häufig zweidimensional und werden vor allem dazu genutzt, 2D-Grafiken auf 3D-Objekten zu positionieren.

```
val assignTemplate = "<attribName> = \"<value>\""
val assignST = ST(assignTemplate)
assignST.addAll(
    "attribName" -> "function",
    "value" -> "test")
val output = assignST.render
```

4.4.2 Simplex3D-Math

Im i>PM3D-Projekt wird die in Scala implementierte Mathematikbibliothek *Simplex3D-Math* in der Version 1.3 [[www.simplex3d](http://www.simplex3d.com)] genutzt. Durch die Bibliothek werden Matrizen, Vektoren und dazugehörige Utility-Funktionen bereitgestellt. Deren API orientiert sich weitgehend an der OpenGL Shading Language.

Einordnung in das Gesamtprojekt

i>PM3D

Diese Arbeit und die dazugehörige Implementierung sind im Rahmen des i>PM3D-Projekts entstanden. Das Ziel des Projekts ist es, einen Prototypen eines grafischen 3D-Prozessmodellierungswerkzeugs zu erstellen, der prinzipielle Vor- und Nachteile von 3D-Editoren zeigen und als Grundlage für weitere Arbeiten in dieser Richtung dienen soll. Das vorliegende Kapitel gibt eine Übersicht über das i>PM3D-Projekt¹.

5.1 Übersicht über die Projektbestandteile

In der Übersichtsgrafik Abbildung 5.1 wird der konzeptionelle Aufbau des Projektes veranschaulicht. Bestandteile, mit denen sich die vorliegende Arbeit befasst, sind darin farbig hervorgehoben (rechts). In das i>PM 3D-Projekt sind ebenfalls die Bachelorarbeiten von Sebastian Buchholz [Buc12] und Uli Holtmann [Hol12] einzuordnen, die die übrigen Bestandteile von i>PM3D beschreiben.

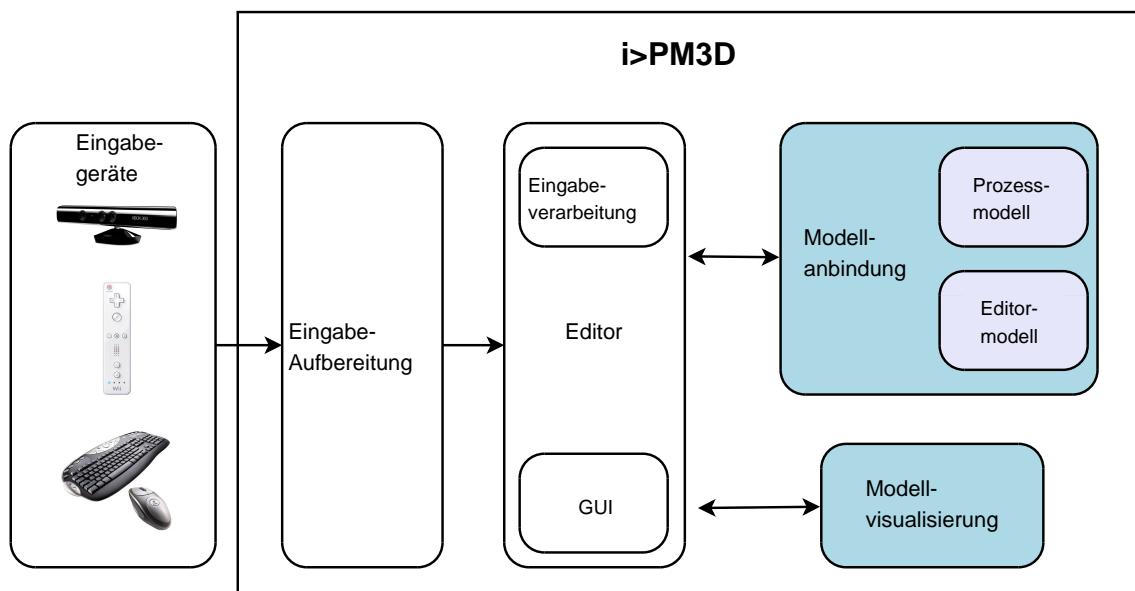


Abbildung 5.1: Übersicht über die Bestandteile von i>PM3D

¹Dieses Kapitel ist in Zusammenarbeit mit [Buc12] und [Hol12] entstanden und in jenen Arbeiten in ähnlicher Form zu finden.

Im Folgenden werden die einzelnen Projektbestandteile kurz vorgestellt und miteinander in Beziehung gesetzt.

5.1.1 Visualisierung

Eine zentrale Fragestellung bei der Realisierung eines grafischen Prozessmodellierungswerkzeugs ist es, auf welche Weise Prozessmodelle visualisiert werden sollen.

Elemente aus dem Prozessmodell sollen in einer für den Benutzer leicht verständlichen Art und Weise angezeigt werden, die die Möglichkeiten des dreidimensionalen Raums nutzt. Die Darstellung soll dabei an die aus 2D-Prozessmodellierungswerkzeugen bekannten grafischen Notationen angelehnt sein. Prozessmodelle in i>PM3D werden in einer graphbasierten Form, also durch Knoten und damit verbundenen Kanten dargestellt. Zusätzlich zu den eigentlichen Prozessmodellelementen gibt es die Möglichkeit, beliebige 3D-Objekte anzuzeigen. Dies kann beispielsweise dafür genutzt werden, um abstrakte Konzepte mit Abbildern von realen Objekten zu illustrieren.

Wie in der Computergrafik üblich wird das Prinzip einer virtuellen Kamera benutzt, durch die der Benutzer die Szene beobachtet („Egoperspektive“). Durch Verschieben und Rotieren der Kamera kann sich der Benutzer in der virtuellen Umgebung „bewegen“ und die dargestellten Prozessdiagramme aus verschiedenen Perspektiven betrachten. Knoten und Szenenobjekte sind frei drehbar um alle drei Achsen und unter Beibehaltung der Seitenverhältnisse skalierbar. Sie können prinzipiell frei in der 3D-Szene platziert werden. Die Visualisierung von Modellen in i>PM3D wird in der vorliegenden Arbeit näher vorgestellt (Kapitel 8).

5.1.2 Modellanbindung

Ein Modellierungswerkzeug muss die Fähigkeit haben, bestehende Modelle aus einer physischen Repräsentation zu laden, das Modell beziehungsweise dessen Elemente zu bearbeiten und wieder zu speichern. Außerdem sollen neue Modelle erstellt werden können. In i>PM 3D kann die grafische Modellierungssprache (in einem gewissen Rahmen) ohne Änderungen am Programmcode modifiziert werden, da die abstrakten Modellelemente und deren (visuelle) Repräsentation jeweils durch ein zur Laufzeit geladenes Metamodell beschrieben werden. Die in einem Prozessmodell verwendeten abstrakten Modellelemente (bspw. ein „Prozessknoten“) sowie deren aktuelle Repräsentation im Modelleditor (bspw. das „Aussehen“ und die Position eines Prozessknotens) werden in separaten Modellen abgelegt. In der Abbildung 5.1 werden diese Modelle als „Prozess-Modell“ bzw. als „Editor-Modell“ bezeichnet.

Der grundsätzliche Aufbau und die Anpassbarkeit der Modell-Hierarchie wird in dieser Arbeit (Kapitel 6) besprochen. Außerdem werden die verwendeten Metamodelle im Detail beschrieben (Kapitel 7) und ein Beispiel gezeigt, wie sich neue Modellelemente ergänzen lassen.

Es ist ebenfalls Gegenstand dieser Arbeit, die Einbindung der Modell-Funktionen in den Prototypen zu realisieren. In der Übersichtsgrafik Abbildung 5.1 wird dieser Teil des Projekts als „Modellanbindung“ (Kapitel 9) bezeichnet.

5.1.3 GUI

Dem Benutzer wird das 3D-Prozessdiagramm in einer interaktiven Umgebung präsentiert, die das Erstellen, Bearbeiten und Löschen von Elementen erlaubt.

Die verschiedenen Funktionen des Prozessmodellierungswerkzeugs wie das Erstellen von Modellelementen und das Laden von Modellen lassen sich durch grafische Menüs aktivieren, die über der 3D-Szene gezeichnet werden und die an das Bedienkonzept verbreiteter 2D-Anwendungen mit grafischer Oberfläche angelehnt sind. Für das Erstellen von neuen Knoten und Szenenobjekten wird ein Menü –

auch als „Palette“ bezeichnet – bereitgestellt, über welches die zur Verfügung stehenden Objekte durch einen Klick auf eine Schaltfläche erzeugt werden können. Attribute der Modellelemente, die entweder die Visualisierung selbst oder das damit verbundene Element des Prozessmodells betreffen, können in einem in einem Menü angezeigt und bearbeitet werden. Die Menüs werden in der Übersichtsgrafik Abbildung 5.1 als GUI zusammengefasst.

5.1.4 Eingabeaufbereitung und Editor

Eine wichtige Anforderung an den Prototypen ist, dass verschiedene Arten von Eingabegeräten unterstützt, neue Geräte einfach angebunden und – soweit sinnvoll – nebeneinander benutzt werden können. Die von den Eingabegeräten gelieferten Daten unterscheiden sich je nach Art des Geräts und der verwendeten Schnittstelle deutlich voneinander. Daher ist es sinnvoll, von den Eingabegeräten und deren Schnittstellen zu abstrahieren. Dies wird erreicht, indem die Eingabedaten aller Geräte von einer Eingabeschicht aufbereitet und an eine vereinheitlichte Schnittstelle zur Bedienung der Anwendung weitergeleitet werden. Diese Schnittstelle zur Eingabeverarbeitung wird, zusammen mit dem GUI, in der Übersichtsgrafik Abbildung 5.1 als *Editor* bezeichnet.

Mit der Realisierung des *Editors* sowie mit der Aufbereitung der Daten, die von Tastatur und Maus geliefert werden befasst sich [Hol12].

5.1.5 Neuartige Eingabegeräte

Neben den für Arbeitsplatzrechner üblichen Eingabegeräten Tastatur und Maus, soll der Editor auch mittels „neuartiger“ Eingabegeräte bedienbar sein, die sich besonders für die Interaktion mit virtuellen 3D-Umgebungen eignen könnten. Dabei sind besonders solche Geräte interessant, die auch an einem handelsüblichen, aktuellen Desktop-PC angeschlossen werden können und relativ „preiswert“ sind. Die Bereitstellung von neuartigen Eingabegeräten und die Aufbereitung der Eingabedaten werden von der Arbeit [Buc12] abgedeckt, welche sich speziell mit der Anbindung der Microsoft Kinect und der Nintendo WiiMote befasst. Neben der direkten Nutzung dieser Geräte als „Mausersatz“² werden auch mit den Geräten ausgeführte Gesten und ein spezielles Kinect-Menü als Eingabemethode untersucht und für das Projekt nutzbar gemacht.

Diese Beiträge sind in der Übersichtsgrafik Abbildung 5.1 unter „Eingabegeräte“ und „Eingabeaufbereitung“ zu finden.

5.2 i>PM3D als Simulator X - Applikation

i>PM3D ist als Anwendung auf Basis von *Simulator X* (Abschnitt 4.2) konzipiert. Abbildung 5.2 zeigt, wie die Architektur des Projekts auf den von Simulator X bereitgestellten Funktionalitäten aufbaut. In den beiden folgenden Abschnitten wird zusammengefasst, welche Änderungen am Simulator-X-Basisystem vorgenommen worden sind und wie die im letzten Abschnitt dargestellten Projektteile im Kontext von *Simulator X* umgesetzt werden.

5.2.1 Modifikationen an Simulator X

Für i>PM3D wurde die von *Simulator X* (Abschnitt 4.2) bereitgestellte Physik-Komponente für spezielle Aufgaben erweitert. Die Physikengine wird für die Selektion von Modellobjekten, für die Realisierung

²Dies bedeutet in diesem Zusammenhang, dass die Geräte einen Cursor („Mauszeiger“) steuern, der die aktuelle Position in einer zweidimensionalen Ebene anzeigt. Bei einem „Klick“ wird eine Aktion auf dem darunter befindlichen Objekt ausgelöst.

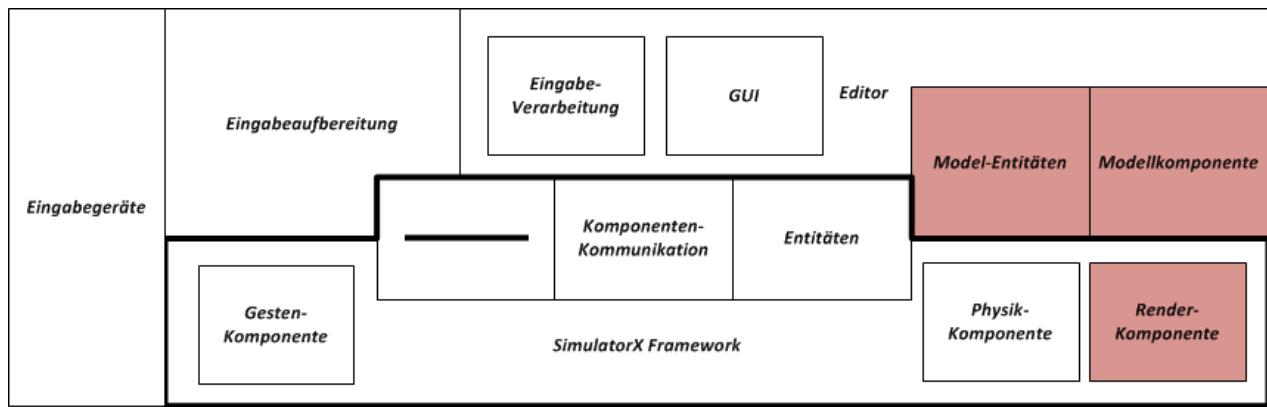


Abbildung 5.2: Architektur von i>PM3D, aufbauend auf Simulator X

von „Gravitationsebenen“, und die Erkennung von Kollisionen zwischen Modellobjekten eingesetzt. Den Einsatz Physikkomponente und die projektspezifischen Modifikationen beschreibt [Buc12].

Die ebenfalls mitgelieferte Renderkomponente, die für die grafische Ausgabe auf Basis von OpenGL zuständig ist, war für das Projekt allerdings nicht sinnvoll nutzbar. Daher wurde diese durch eine Anbindung an die im Rahmen dieser Arbeit entwickelte, flexible *Render-Bibliothek* (Kapitel 11) ersetzt, welche die einfache Erstellung von neuen Modell-Figuren ermöglicht und die Möglichkeiten moderner OpenGL-Grafikprogrammierung nutzt. Die Anbindung an *Simulator X* wird durch die in Abbildung 5.2 gezeigte *Renderkomponente* (Kapitel 10) geleistet.

5.2.2 Modellkomponente und Modell-Entitäten

Die im vorherigen Abschnitt als *Modellanbindung* bezeichneten Funktionalitäten werden im Simulator X - Kontext durch die **Modellkomponente** realisiert, die dem Editor eine Schnittstelle zur Verfügung stellt über welche die genannten Aktionen ausgelöst werden können. Die Modellelemente selbst zu bearbeiten, also deren Visualisierungsparameter und Prozessmodellattribute sowie die Position, Größe und Orientierung im Raum zu ändern, wird durch die von der Modellkomponente bereitgestellten **Modell-Entitäten** ermöglicht, welche durch den Editor manipuliert werden.

Dem Simulator X - Konzept folgend, beschreiben diese *Entities* außerdem, wie die dazugehörigen Objekte von der Physikkomponente behandelt und wie sie von der Renderkomponente angezeigt werden. Näheres zur Modellkomponente und den Modell-Entitäten ist im Kapitel zur *Modellanbindung* (Kapitel 9) zu finden.

5.2.3 Editor-Komponente und Eingabekonnektoren

Die Bedienschnittstelle, in Abbildung 3.2 als Editor beschrieben, ist eine Simulator X Komponente. Wird eine Modell-Entity fertiggestellt, wird sie dem Editor übergeben, so dass dieser sie in seine eigenen Datenstrukturen einbringen und für den Benutzer ansprechbar machen kann. Die Konnektoren der Eingabeaufbereitungsschicht setzen auf keiner Simulator X Komponente auf, benutzen dafür aber das Simulator X Event-System um die aufbereiteten Eingabedaten an die Bedienschnittstelle zu senden. Eine ausführlichere Beschreibung der Editor-Komponente, Eingabekonnektoren sowie der Kommunikation untereinander und mit anderen Komponenten ist bei [Hol12] zu finden.

Modellhierarchie

In der Prozessmodellierung ist es sinnvoll, neben den Modellen auch die zugrundeliegende Modellierungssprache und Visualisierung an spezielle Anforderungen anpassen zu können (siehe [Metamodellierung](#) (Abschnitt 2.2)). Daher war eine solche Flexibilität auch für das vorliegende Arbeit erwünscht ([Anforderung \(d\)](#) (Abschnitt 1.3)).

Daher wurde das Konzept verfolgt, die verwendete grafische Modellierungssprache über austauschbare Metamodelle zu definieren. ([Anforderung \(c\)](#)) Ein wichtiger Punkt ist, dass sich die abstrakte Syntax der Sprache und die konkrete Syntax (die „Visualisierung“) getrennt beschreiben lassen. Diesem Konzept folgt das bereits vorgestellte [Model Designer Framework](#) (Unterabschnitt 2.2.2). Die hier vorgestellte Modellhierarchie ist ähnlich zu der von MDF definierten aufgebaut und übernimmt einige Begriffe von dort. Auf wichtige Unterschiede wird in diesem Kapitel explizit hingewiesen.

Das Konzept und die Implementierung der vorliegenden Arbeit erreicht jedoch nicht die Flexibilität von MDF, da hier ein Modellierungswerkzeug für die POPM und kein generisches Framework realisiert werden sollte. Dennoch ist es möglich, in einem gewissen Rahmen Modifikationen an den verwendeten Modellelementen vorzunehmen und deren Visualisierung anzupassen.

Inwieweit sich die Modelle anpassen lassen und welche Einschränkungen bestehen wird im aktuellen Kapitel und bei der näheren Vorstellung der [Metamodelle](#) (Kapitel 7) deutlich. Am Ende des nächsten Kapitels ist ein [Anwendungsbeispiel](#) (Abschnitt 7.5) zu finden, welches zeigt, wie neue Elemente zur Modellierungssprache hinzugefügt werden können.

Die Anpassbarkeit der konkreten Syntax hat für den hier realisierten Prototypen vor allem den praktischen Nutzen, dass leicht mit der Visualisierung experimentiert werden kann, ohne den Quellcode der Anwendung ändern und neu übersetzen zu müssen. Grundsätzlich lässt sich auch die verwendete Modellierungssprache komplett austauschen, jedoch wird in dieser Arbeit davon ausgegangen, dass das vorgegebene [Prozess-Metamodell](#) (Abschnitt 7.4) genutzt wird.

[Abbildung 6.1](#) zeigt, wie sich die Hierarchie der Modelle darstellt, welche sich in einen **Editor-Model-Stack** und einen **Domain-Model-Stack** aufteilt. Nach einer kurzen Vorstellung der Modellierungssprache wird eine Übersicht über die beiden Model-Stacks gegeben.

6.1 LMMLight

Die Modelle werden mit Hilfe einer Metamodellierungssprache spezifiziert, die vom [Linguistic Meta Model](#) (Unterabschnitt 2.2.1) abgeleitet ist. Zum weiteren Verständnis ist es ausreichend, die in diesem Abschnitt gezeigten Grundelemente und Prinzipien zu kennen.

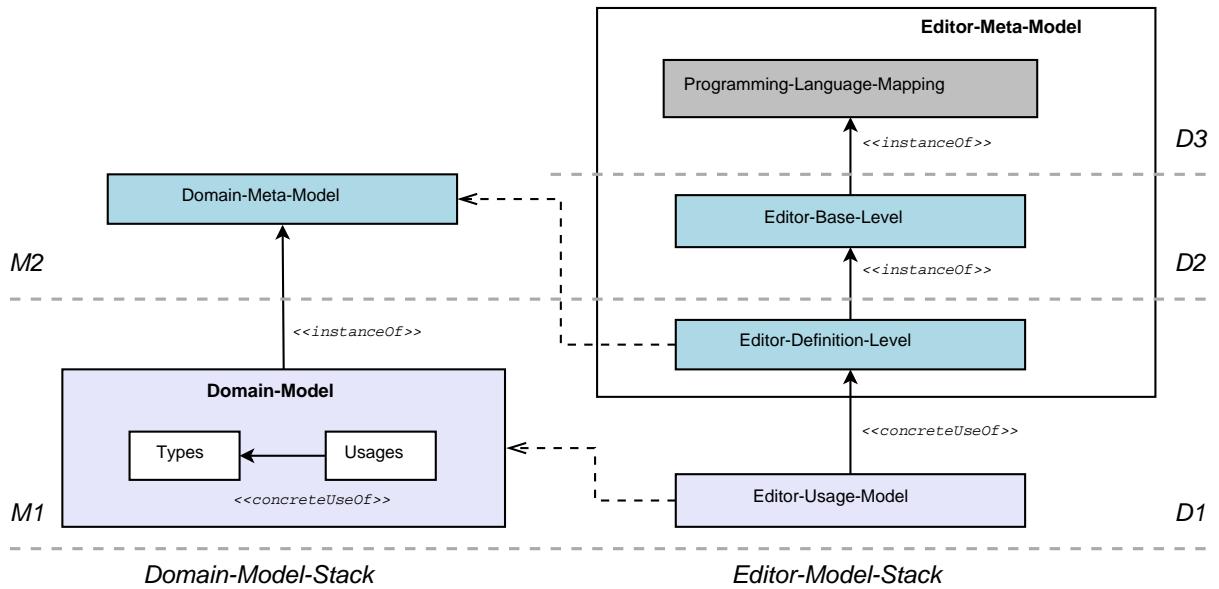


Abbildung 6.1: Modellhierarchie in i>PM3D (angelehnt an MDF [Rot11])

Die hier verwendete Sprache, im Folgenden **LMMLight** genannt, folgt in vielen Aspekten LMM, ohne jedoch alle weiterführenden Modellierungsmuster zu unterstützen, um eine einfache Implementierung zu ermöglichen. Konkret hat dies zur Folge, dass der textuelle Modell-Editor von OMME für die Erstellung von LMMLight-Modellen sinnvoll genutzt werden kann, solange auf die nicht unterstützten Modellierungsmuster verzichtet wird.

LMMLight unterstützt das Muster der **Spezialisierung von Instanzen** (`concreteUseOf`), welches unter anderem für die Realisierung des *Typ-Verwendungs-Konzepts* (Abschnitt 2.3) genutzt wird. Im Gegensatz zu LMM lassen sich in Spezialisierungen alle Attributzuweisungen des spezialisierten Concepts ohne Einschränkung überschreiben.

6.2 Editor-Model-Stack

Der *Editor-Model-Stack* von i>PM3D enthält alle Modelle, die dafür zuständig sind, die Visualisierungsparameter eines Domänenmodells zu beschreiben. Außerdem werden hier Parameter spezifiziert oder gesetzt, welche die physikalische Repräsentation oder die für das Modellement angebotenen Funktionalitäten im interaktiven Modellierungswerkzeug beeinflussen. Näheres hierzu wird im nächsten Kapitel erläutert. Mit „Repräsentation“ ist im Folgenden die Gesamtheit dieser Parameter gemeint.

Die Verknüpfung mit dem *Domain-Model-Stack* wird hergestellt, indem in Concepts des *Editor-Model-Stacks* eine Referenz auf *Domain-Model-Stack-Concepts* angegeben wird (Abbildung 6.2). In Abbildung 6.1 wird dies durch gestrichelte Pfeile dargestellt. Besagte Referenzen werden durch das Attribut `modelElementFQN` angegeben, welchem der voll qualifizierte Name (FQN) des referenzierten Concepts zugewiesen wird. Vollqualifizierte Namen entstehen nach dem Schema `<Model>.<Level>.<Package>.<Concept>`, beispielsweise `EMM.D1.nodeFigures.ProcessNode`.

6.2.1 Anpassbarkeit

Durch Anpassungen im *Editor-Model-Stack* können für ein Domänen-Metamodell mehrere verschiedene Repräsentationen erstellt werden. Im Vergleich zur Modellhierarchie von *MDF* (Unterabschnitt 2.2.2) ist das im *Designer-Model-Stack* von MDF definierte *Graphical-Meta-Model* und das *Editor-Meta-Model* zusammengelegt. Durch die fehlende Trennung von grafischer Darstellung und Editor-Mapping wird die

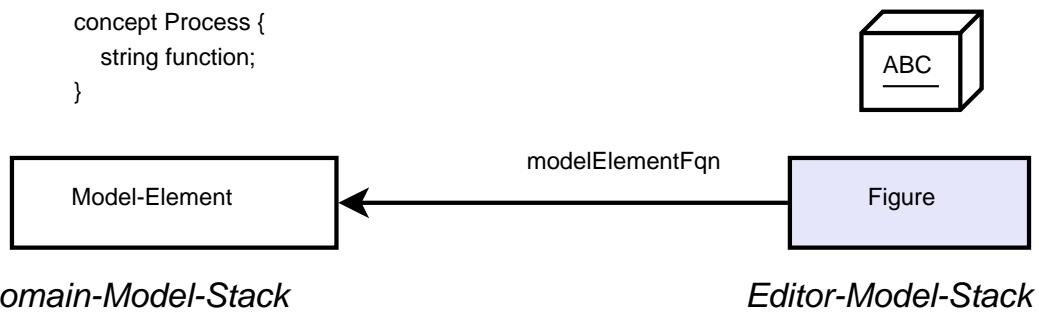


Abbildung 6.2: Assoziation zwischen abstraktem Modellelement und konkreter Repräsentation

Wiederverwendbarkeit im Vergleich zu MDF allerdings eingeschränkt. Bei getrennten Modellen ist es möglich, eine „Bibliothek“ von Visualisierungselementen bereitzustellen, aus der Elemente ausgewählt und in beliebig vielen Editor-Definitionen verwendet werden können. Da der Fokus dieser Arbeit auf der (perspektivenorientierten) Prozessmodellierung liegt, wurde jedoch darauf verzichtet, um die Implementierung zu vereinfachen. Dabei wird hingenommen, dass die Repräsentationen der einzelnen Domänenmodellelemente (auch „Figuren“ genannt) für jede neue Repräsentation des Domänenmodells komplett neu beschrieben werden müssen.

Bei der Erstellung der Figuren muss berücksichtigt werden, dass durch die Implementierung der *Modellkomponente* (Abschnitt 9.1) eine feste Auswahl an Visualisierungsparametern definiert ist. Welche dies sind, kann in der Beschreibung der *Modell-SVars* (Unterabschnitt 9.3.3) nachgelesen werden.

Editor-Definition- und *Editor-Meta-Model* können zwar konzeptionell – wie im MDF – unterschieden werden; jedoch wird in dieser Arbeit davon ausgegangen, dass diese zusammen in einem Modell (im Sinne von LMM) definiert werden, welches hier als das **Editor-Metamodell** bezeichnet wird.

Um eine andere Visualisierung festzulegen müsste das komplette Editor-Metamodell neu definiert werden, sinnvollerweise auf Basis des bestehenden Metamodells¹.

6.2.2 Übersicht über die Editor-Model-Ebenen

Abbildung 6.1 veranschaulicht, wie die Editor-Model-Ebenen, die im Folgenden vorgestellt werden, von „oben nach unten“ definiert sind. *Programming-Language-Mapping*, *Editor-Base-Level* und *Editor-Definition-Level* ergeben zusammen das **Editor-Metamodell**, welches die Repräsentation eines bestimmten Domain-Metamodells oder – anders gesagt – einen **Editor** für das Domain-Metamodell spezifiziert.

Programming-Language-Mapping

Auf der obersten Ebene des Stacks, die im Modell als Level D3 zu finden ist, wird die Abbildung auf eine Programmiersprache – in dieser Arbeit auf Scala – definiert, welche in *Scala-Mapping* (Abschnitt 7.1) beschrieben wird. In der Abbildung 6.1 wird diese Ebene als *Programming-Language-Mapping* bezeichnet.

Editor-Base-Level

Darunter befindet sich auf Level D2 der prinzipiell von der Modellierungsdomäne unabhängige Teil der Editor-Spezifikation. Hier werden Concepts bereitgestellt, die die Grundlage der Repräsentation für Typen aus dem Domänenmodell darstellen.

In der Abbildung 6.1 ist diese Ebene als *Editor-Base-Level* zu finden.

¹ „Copy-And-Paste-Wiederverwendung“

Die beiden bisher beschriebenen Ebenen D3 und D2 können prinzipiell beliebig definiert werden, soweit dies von LMMLight unterstützt wird.

Editor-Definition-Level

Die Modellebene D1 legt fest, auf welche Weise ein Elementtyp aus dem *Domain-Meta-Model* repräsentiert wird.

Auf dieser Ebene müssen die folgenden Packages definiert sein (vorgegeben durch die Implementierung):

- package `nodeFigures` definiert Concepts, die die Repräsentation von Knoten aus dem Domänenmodell beschreiben.
- package `connectionFigures` definiert Concepts, die die Repräsentation von Kanten aus dem Domänenmodell beschreiben.
- package `sceneryObjects` enthält die verwendbaren „Szenenobjekte“ (*Anforderung (h)*: Anzeige beliebiger 3D-Objekte). Szenenobjekt-Concepts haben keine Entsprechung im Domänenmodell, da sie kein Modellelement repräsentieren.

Damit ist fest vorgegeben, dass sich die Modellelemente in Knoten und Kanten unterscheiden lassen, also prinzipiell ein graphbasierter Ansatz genutzt wird (*Anforderung (a)*). Zusammen bilden diese Packages den in der [Abbildung 6.1](#) gezeigten *Editor-Definition-Level*.

Es dürfen auch noch weitere Packages vorkommen, die Concepts enthalten, welche von Concepts aus den obigen Packages referenziert werden. Dies können beispielsweise Concepts für die Definition von Farben oder der Größe eines Objekts sein.

Editor-Usage-Model

Ebenfalls auf Level D1 befindet sich das *Editor-Usage-Model*, das Verwendungen, also Spezialisierungen von Concepts aus dem *Editor-Definition-Level* enthält.

Analog zum *Editor-Definition-Level* sind die Verwendungen in drei Packages eingeteilt, die hier `nodeUsages`, `connectionUsages` und `sceneryObjectUsages` genannt werden müssen.

Zusammen ergeben diese Verwendungen die konkrete Repräsentation eines Domänenmodells. Diese Concepts spezifizieren hier also die Objekte, die vom Modellierungswerkzeug erstellt und auf der Zeichenfläche angezeigt werden.

Sie legen damit beispielsweise fest, wo sich Modellelemente im Raum befinden und welche Ausrichtung sie haben. Dies sind auch typische Parameter, in denen sich alle Verwendungen einer Instanz unterscheiden.

Dem Konzept der Spezialisierung von Instanzen folgend kann hier auch die konkrete Visualisierung des Objekts beeinflusst werden. Wird in den Verwendungen für ein Attribut kein Wert angegeben, wird der Wert aus dem konkret verwendeten Concept benutzt.

Modellelemente, die von derselben Instanz abstammen haben also grundsätzlich das gleiche Erscheinungsbild, solange keine Werte überschrieben werden.

6.3 Domain-Model-Stack

Der Domain-Model-Stack umfasst alle Modelle, welche die Modellierungsdomäne beschreiben. Durch das *Domain-Meta-Model* wird die abstrakte Syntax der Modellierungssprache festgelegt.

6.3.1 Domain-Meta-Model

Durch das *Domain-Meta-Model* werden die im *Domain-Model* erlaubten Modellelemente vorgegeben. An die Struktur des Modells, also den Aufbau aus Levels und Packages, werden durch die Implementierung keine besonderen Anforderungen gestellt.

Durch den *Editor-Definition-Level* (Unterunterabschnitt 6.2.2) wurde bereits vorgegeben, dass ein graph-basierter Visualisierungsansatz genutzt wird. Passend dazu werden hier Knoten definiert, die mittels Kanten verbunden sind.

In der Implementierung von i>PM3D wird angenommen, dass Knoten und Kanten über spezielle Attribute der Knoten logisch miteinander verbunden sind. So muss im Concept, das den Knotentyp beschreibt, jeweils ein Attribut für eingehende und ausgehende Kanten eines bestimmten Typs definiert sein. Diesen Attributen werden die ein- bzw. ausgehenden Kanten durch das Modellierungswerkzeug zugewiesen. Die Existenz von zugehörigen Attributen legt daher fest, in welcher Weise Kanten mit Knoten assoziiert werden können. Es wird vorgesetzt, dass die Attributnamen für eingehende Kanten mit dem Präfix `inbound` und die ausgehenden mit `outbound` beginnen. Der Rest des Attributnamens kann im Prinzip frei gewählt werden; jedoch wird in dieser Arbeit die Konvention benutzt, den Typnamen der Kante anzuhängen.

Ist also beispielsweise in einem Knotentyp für einen bestimmten Kantentyp nur ein `outbound`-Attribut definiert, sind nur Verbindungen erlaubt, die ihren Startpunkt bei jenem Knotentyp haben. Der Endpunkt müsste dann bei einem anderen Knotentyp liegen, der ein entsprechendes `inbound`-Attribut besitzt². Das Prinzip wird im nächsten Kapitel bei der Vorstellung des verwendeten *Prozess-Metamodells* (Abschnitt 7.4) und anschließend in einem *Anwendungsbeispiel* (Abschnitt 7.5) verdeutlicht.

Ansonsten können im Modellierungswerkzeug modifizierbare Modellattribute frei definiert werden, wobei beachtet werden muss, dass von der Implementierung nur Literatyp-Attribute allgemein (außer für die Assoziation von Knoten und Kanten, wie vorher beschrieben) unterstützt werden. Attribute, die Concepts referenzieren, können im Editor nicht angezeigt oder verändert werden und werden ignoriert³.

6.3.2 Domain-Model

Das *Domain-Model* enthält das konkrete Domänenmodell, wie es im Modellierungswerkzeug durch die zugehörigen Concepts aus dem *Editor-Usage-Model* (Unterunterabschnitt 6.2.2) visualisiert wird. Zusammen mit dem *Editor-Usage-Model* (Unterunterabschnitt 6.2.2) ergibt dies den aktuellen Zustand des angezeigten Modells, welcher persistiert und wieder geladen werden kann. Das *Domain-Model* muss den Level M1 enthalten, auf dem die im Folgenden genannten Packages definiert sind.

Für die Erzeugung von Knoten im *Domain-Model* wird immer das *Typ-Verwendungs-Konzept* (Abschnitt 2.3) verwendet. Dies bedeutet, dass im *Domain-Meta-Model* Concepts definiert sind, von welchen im *Domain-Model* eine Instanz („Typ-Concept“) erstellt wird. Von *Typ-Concepts* kann eine Verwendung (in Form einer Spezialisierung der Instanz) im *Domain-Model* erzeugt werden.

Die Implementierung gibt vor, dass die benutzerdefinierten Typen in einem Package mit dem Namen `types` abgelegt werden. Verwendungen davon werden im Package `nodeUsages` abgelegt.

Für Kanten kommt das Typ-Verwendungs-Konzept im Domänenmodell nicht zum Einsatz. Kanten sind daher direkte Instanzen von Typen aus dem *Domain-Meta-Model* und werden zum Package `connections` hinzugefügt.

²Im Domänenmodell sind Kanten also technisch gesehen immer „gerichtet“.

³Als „Ausweg“ kann ein zusätzlicher Knotentyp und eine passende Verbindung definiert werden, so dass der Sachverhalt vom Editor visualisiert und modifiziert werden kann.

Spezifikation der Metamodelle

Nachdem im vorherigen Kapitel eine Übersicht über die von i>PM3D unterstützte Metamodell-Hierarchie gegeben wurde, werden hier die im Projekt verwendeten Metamodelle für Editor und Domäne sowie deren Concepts genauer vorgestellt. Das verwendete Editor-Metamodell wird im Folgenden mit **EMM** bezeichnet, das Domain-Metamodell mit **PMM** (Prozess-Metamodell). Die im Folgenden genannten voll qualifizierten Namen (FQN) von Levels setzen sich aus dem Modellnamen und dem Levelnamen zusammen, getrennt durch einen Punkt. FQNs für Packages und Concepts entstehen, indem deren Namen in der gleichen Weise angehängt werden.

7.1 Scala-Mapping

Die oberste Ebene des *Editor-Model-Stacks* beinhaltet nur ein Package `base` (FQN `EMM.D3.base`) mit einem einzelnen Concept, `ScalaMapping`. In textueller Darstellung ist diese Ebene in [Anhang A](#) (Unterabschnitt A.1.1) nachzulesen.

Dieses Concept definiert Attribute, die festlegen, wie Concepts aus dem Modell auf Scala-Objekte abgebildet werden, um im Modellierungswerkzeug genutzt werden zu können. Für jedes Concept, das sich auf den weiter unten liegenden Ebenen befindet, muss das Attribut `scalaType` definiert werden, welches den korrespondierenden Scala-Typ angibt.

Optional ist das Attribut `typeConverter`, welches eine Klasse spezifiziert, die dazu genutzt wird, ein LMM-Concept in ein passendes Scala-Objekt umzuwandeln und umgekehrt.¹ Ohne TypeConverter-Angabe wird `scalaType` direkt als voll qualifizierter Klassenname interpretiert. Von dieser so angegebenen Klasse wird ein Objekt erstellt, welches das entsprechende Concept in der Anwendung vertritt.

Wird ein TypeConverter genutzt, muss der `scalaType` nicht zwingend ein Klassenname sein. Wie das Attribut interpretiert wird hängt vom jeweiligen TypeConverter ab. So ist es beispielsweise möglich, dass hier nur ein bestimmtes Interface bzw. Trait angegeben wird und die Wahl der konkreten Implementierung vom TypeConverter vorgenommen wird.

7.2 Editor-Base-Level

Level D2 ist als Instanz von D3 definiert. Daraus folgt, dass alle hier definierten Concepts Instanzen von `ScalaMapping` sein müssen. Die auf dieser Ebene definierten Concepts sind prinzipiell von der

¹Die Implementierung stellt TypeConverter für verschiedene Simplex3D-Vektoren und Quaternionen sowie für die Klassen `java.awt.Font` und `.Color` zur Verfügung. Weitere TypeConverter können auf Basis des TypeConverter-Traits (Scala-Package `mmpe.model.lmm2scala`) definiert werden.

Prozessmodellierung unabhängig, orientieren sich aber an deren Bedürfnissen.

In [Anhang A](#) (Unterabschnitt A.1.2) ist dieses Modell vollständig abgebildet. Auf D2 werden zwei Packages, `types` und `figures`, definiert.

7.2.1 Paket „types“

Das EMM.D2.`types`-Package definiert grundlegende Typen, die Visualisierungsparameter von Objekten und die Positionierung im Raum sowie deren Größe beschreiben. Dazu werden folgenden Typen angeboten:

- Dimension, Position: Spezifikation der Größe und der Position eines Objektes im dreidimensionalen Raum, welche in einem kartesischen Koordinatensystem angegeben werden. Die drei Attribute `x`, `y`, `z` werden im Editor auf einen Vektor mit drei Komponenten abgebildet. Hierfür wird der Vektortyp `Vec3` von [Simplex3D-Math](#) (Unterabschnitt 4.4.2) angeboten.
- Rotation: Angabe der Rotation mittels eines Quaternions². Die vier Attribute `x0`, `x1`, `x2` und `x3` werden auf ein Quaternionen-Objekt `Quat4` abgebildet, das ebenfalls von [Simplex3D](#) bereitgestellt wird.
- Color: Hiermit lassen sich Farben, die mittels im RGBA-Farbsystem als rot, grün, blau und alpha (Transluzenzwert) angegeben werden. Zu beachten ist hier, dass die Farben als Gleitkommazahl angegeben werden und einen Wertebereich von 0 bis 1 abdecken. Dieser Typ wird auf die `java.awt.Color`-Klasse abgebildet.
- Font: Definiert Parameter für die Schriftdarstellung. Die Attribute und deren erlaubte Werte orientieren sich an `java.awt.Font`, worauf dieses Concept abgebildet wird. Die Schriftfarbe muss separat mittels eines `Color`-Concepts angegeben werden.
 - `face`: Name der Schriftart
 - `size`: Größe, als Ganzzahl angegeben
 - `style`: Name des Schriftstils. Erlaubt sind hier „normal“, „bold“ und „italic“, die den Werten der Enumeration `FontStyle` von `java.awt` entsprechen.
- PhysicsSettings: Sub-Concepts dieses abstrakten Concepts werden genutzt, um Objekten eine physikalische Repräsentation zu geben, wenn diese nicht auf anderem Wege definiert wurde. Es werden kugel- (`PhysSphere`) und quaderförmige (`PhysBox`) Geometrien angeboten, wie sie von der durch [Simulator X](#) (Abschnitt 4.2) bereitgestellten Physikkomponente unterstützt werden. Für eine `PhysSphere` muss der Radius angegeben werden; eine `PhysBox` wird analog über die halben Seitenlängen (Attribut `halfExtends`, Typ `Dimension`) festgelegt.

7.2.2 Paket „figures“

Im EMM.D2.`figures`-Package werden die grundlegenden Figuren definiert, die zur Visualisierung von Domänenmodellelementen zur Verfügung stehen.

Hier wird eine graphbasierte Darstellungsform vorausgesetzt, das heißt, dass hier die speziell dafür benötigten Concepts bereitgestellt werden. [Abbildung 7.1](#) zeigt die Hierarchie der in diesem Paket definierten Basis-Figuren, die im folgenden näher beschrieben werden. Die gezeigten Attribute und Assoziationen werden von der Implementierung vorausgesetzt.

Das Package wird durch zwei abstrakte Basistypen, `EditorElement` und `SceneryObject` strukturiert. `EditorElement` ist der Basistyp aller Graphenelemente, welche sich wiederum in Kanten (`Edge`) und Knoten (`Node`) aufteilen.

²Quaternionen erlauben eine kompakte Darstellung von Rotationen im 3D-Raum [[www:quat](#)].

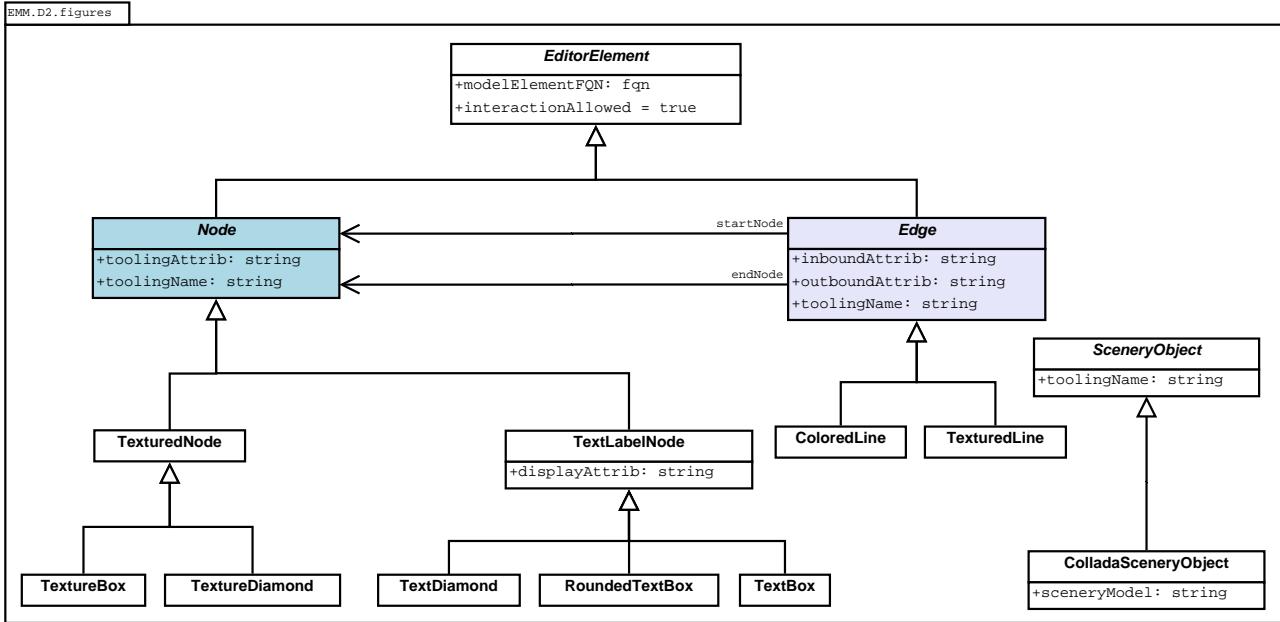


Abbildung 7.1: Hierarchie des `figures`-Pakets

Jedes `EditorElement` muss das Attribut `modelElementFQN` setzen, dass den voll qualifizierten Namen des repräsentierten *Domain-Concepts* angibt. Über das Attribut `interactionAllowed` lässt sich festlegen, ob eine Interaktion mit dem Modellelement durch den Benutzer erlaubt ist. Dies ist standardmäßig für alle Element auf „true“ gesetzt.

Das von `ScalaMapping` definierte Attribut `scalaType` legt für Concepts in diesem Package fest, durch welche Objekte diese konkret im Modellierungswerkzeug grafisch dargestellt werden. Es ist zu beachten, dass die Interpretation von `scalaType` hier nicht den *Scala-Mapping* (Abschnitt 7.1) angegebenen Konventionen folgt und der Wert kein Klassenname sein muss, obwohl kein `TypeConverter` angegeben wird. Wie die Werte interpretiert werden, ist später in einem *Anwendungsbeispiel* (Abschnitt 11.6) zu sehen, nachdem die dafür nötigen Grundlagen erläutert worden sind.

Knoten

Das Basis-Concept aller Knoten, **Node** definiert die Attribute `dim` (Typ Dimension), `pos` (Position) und `rotation` (Rotation), die dazu benutzt werden, sowohl das Erscheinungsbild als auch das physikalische Verhalten zu beschreiben. In der Implementierung wird sichergestellt, dass Visualisierung und physikalische Repräsentation immer zueinander passen. Das bedeutet beispielsweise, dass die für den Benutzer sichtbare Ausdehnung genau die ist, die auch für die Erkennung von Kollisionen oder bei der Auswahl von Elementen durch ein Eingabegerät genutzt wird.

Für die Visualisierung von Knoten sind ein texturierter (**TexturedNode**) und ein beschrifteter (**TextLabelNode**) Basistyp vorgesehen, die folgende Attribute definieren:

- **TexturedNode**:
 - `texture`: Pfad zu einer Bilddatei, die auf dem Knoten angezeigt wird.³
 - `backgroundColor`: Hintergrundfarbe des Knoten.
- **TextLabelNode**:
 - `displayAttrib`: Gibt den Namen eines Attributs aus dem zugeordneten Domänenkonzepts an, dessen textuelle Darstellung als Schrift auf dem Knoten angezeigt wird.

³Unterstützt werden PNG, JPEG, BMP und TGA

- `fontColor`: Schriftfarbe, als `Color`-Instanz spezifiziert.
- `backgroundColor`: Hintergrundfarbe, die an nicht von der Schrift abgedeckten Stellen angezeigt wird.
- `font`: Schriftart, angegeben als `Font`-Instanz

Es wird davon ausgegangen, dass für Knoten im Domänenmodell das Typ-Verwendungs-Konzept genutzt wird. Wie in [GUI](#) (Unterabschnitt 5.1.3) erwähnt, sollen verfügbare Knotentypen in einem Menü („Palette“) angezeigt werden, dass die Erstellung von neuen Modellelementen erlaubt.

Daher müssen alle Nodes folgende Attribute setzen:

- `toolingAttrib`: Legt fest, welches (String)-Attribut aus dem *Domain-Concept* zur Identifikation des Node-Typs in einer Palette angezeigt werden soll.
- `toolingTitle`: Hierdurch wird angegeben, unter welcher Kategorie ein Node-Typ in einer Palette eingesortiert werden soll. Diese „Überschriften“ korrespondieren mit den Knotentypen, die im *Domain-Meta-Model* definiert werden.

Kanten

Für Kanten stehen ein einfarbiger (`ColoredLine`) und ein texturierter Basistyp (`TexturedLine`) zur Verfügung. `TexturedLine` bietet die gleichen Attribute wie `TexturedNode` an; bei `ColoredLine` muss noch die Grundfarbe gesetzt werden (`color`). Zusätzlich wird bei beiden noch eine spekulare Farbe⁴, `specularColor` angegeben.

Bei Kanten wird davon ausgegangen, dass das Typ-Verwendungs-Konzept im Domänenmodell nicht zum Einsatz kommt und Verbindungen direkt instanziert werden. Wie Kantentypen innerhalb der grafischen Benutzeroberfläche bezeichnet werden sollen wird durch das Attribut `toolingName` festgelegt.

In Concepts, die Kantentypen repräsentieren müssen außerdem die Attribute von Knotentypen aus dem Domänenmodell angegeben werden, denen die Domain-Concepts der zugehörigen Verbindungen zugewiesen werden. `InboundAttrib` legt den Namens des Attributs fest, dem eingehende Kanten zugewiesen werden; `outboundAttrib` ist entsprechend das Attribut für die ausgehenden Kanten. Außerdem sind für Kanten noch die beiden Attribute `startNode` und `endNode` definiert. Diesen Attributen wird im *Editor-Usage-Model* jeweils das *Editor-Concept* zugewiesen, welches den Ausgangs- bzw. den Endknoten repräsentiert.

Szenenobjekte

Typen für Szenenobjekte werden vom Basistyp `SceneryObject` abgeleitet. Wie für Knoten werden Attribute für die Position, Größe und Rotation definiert. Wie der Typ innerhalb der grafischen Benutzeroberfläche bezeichnet werden soll wird durch das Attribut `toolingName` festgelegt.

Für Szenenobjekte kann eine physikalische Repräsentation (Typ `PhysicsSettings`) definiert werden, falls diese nicht anderweitig festgelegt wird.

Es gibt momentan nur eine Art von Szenenobjekten, das `ColladaSceneryObject`. Über das Attribut `modelPath` kann ein Pfad zu einer COLLADA-Datei⁵ angegeben werden. Eine Physikdefinition innerhalb des COLLADA-Modells wird nicht unterstützt. Daher muss für `ColladaSceneryObjects` im Modell eine Physikrepräsentation gesetzt werden wenn die Objekte bei der Kollisionsberechnung berücksichtigt werden sollen und Selektion durch den Benutzer möglich sein soll. Näheres zur COLLADA-Unterstützung in i>PM3D lässt sich bei [\[Hol12\]](#) (Unterabschnitt 7.5.2) nachlesen.

⁴„Spekulare Farbe“ ist ein Begriff, der oft im Zusammenhang mit dem Phong-Lichtmodell [\[Pho75\]](#) benutzt wird und dort für die spiegelnden Anteile des zurückgeworfenen Lichts steht.

⁵COLLADA ist ein XML-Austauschformat für 3D-Modelle und weitere Aspekte (Physik, Szenenbeschreibungen etc.) [\[collada\]](#)

7.3 Editor-Definition-Level

Auf dieser Ebene sind die Concepts zu finden, die die Repräsentationen für Knoten und Kanten aus dem Prozessmodell darstellen. Da hier nur Werte gesetzt und keine neuen Attribute definiert werden, wird hier auf eine gesonderte Beschreibung verzichtet. Eine beispielhafte Auswahl der hier definierten Concepts kann in [Anhang A](#) (Unterabschnitt A.1.3) nachgelesen werden. Das Aussehen einiger hier spezifizierter Figuren wird im nächsten Kapitel [3D-Visualisierung von Prozessen](#) (Kapitel 8) gezeigt.

7.4 Prozess-Metamodell

Das in dieser Arbeit verwendete *Domain*-Metamodell orientiert sich an den Metamodellen für die *POPM* (Unterabschnitt 2.1.2), wie sie in [Vol11] vorgestellt werden. Das vollständige Metamodell kann in *Prozess-Metamodell* (Abschnitt A.2) nachgelesen werden.

Das Prozess-Metamodell definiert nur ein Paket, PMM.M2.processLanguage.

Die einzelnen Perspektiven sind als abstrakte Basis-Concepts definiert, die Perspective erweitern. Abbildung 7.2 zeigt die Concept-Hierarchie, die sich unterhalb von Perspective aufspannt.

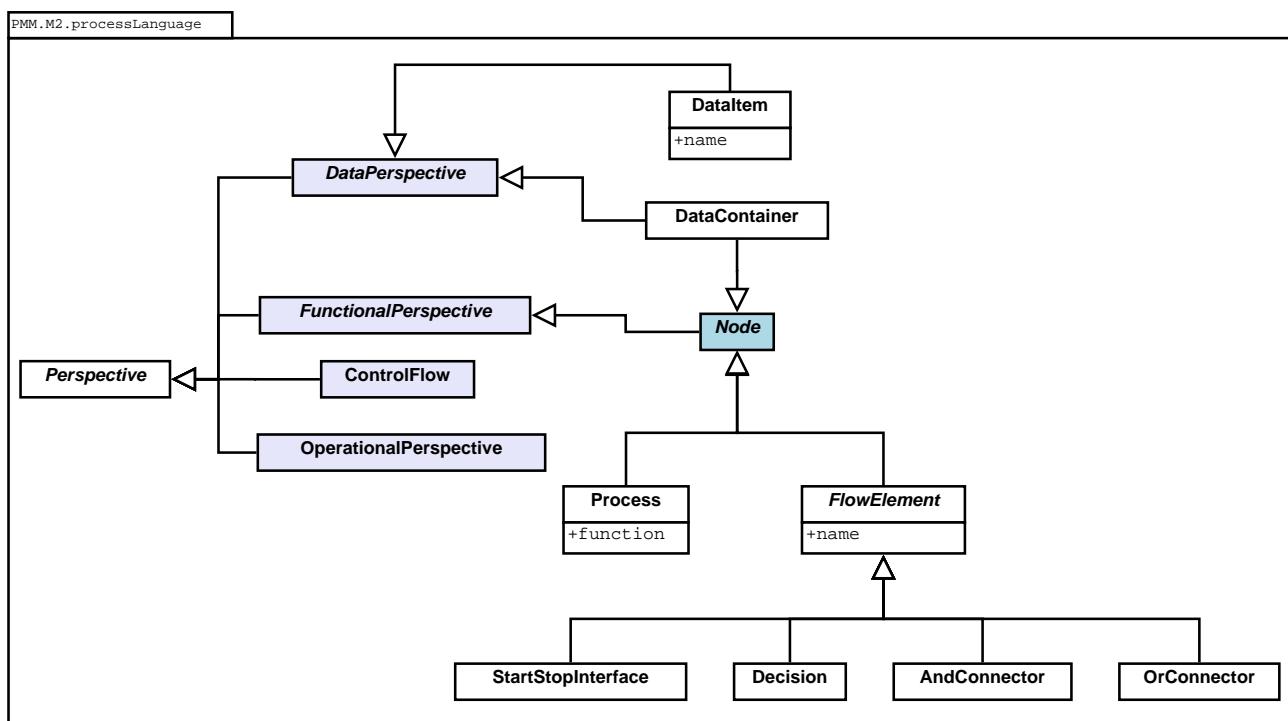


Abbildung 7.2: Perspektiven-Hierarchie im Prozess-Meta-Modell

Node gehört zur funktionalen Perspektive, davon sind wiederum Process und FlowElement abgeleitet. Process stellt einen Prozess im Sinne der POPM dar. Von FlowElement sind Kontrollflusselemente wie Konnektoren (AndConnector, OrConnector) und Entscheidungsknoten (Decision) abgeleitet.

Die Datenperspektive teilt sich auf in DataItem, welches einzelne Dateneinheiten repräsentiert, und in DataContainer, der DataItems zu einer Gruppe zusammenfasst.

Die bisher genannten Concepts bzw. Perspektiven lassen sich als Knoten des Prozessgraphen interpretieren. Die verhaltensorientierte Perspektive hingegen — vertreten durch ControlFlow — lässt sich als Kante betrachten, welche Nodes miteinander verbindet.

DataItems können über (gerichtete) Datenflüsse (DataFlow) miteinander verbunden werden. DataContainer ist gleichzeitig Teil der funktionalen Perspektive und kann daher über Kontrollflüsse mit anderen Nodes verbunden werden.

Im Unterschied zu den von [Vol11] definierten Metamodellen werden Beziehungen zwischen Knoten immer mittels expliziter Verbindungs-Concepts spezifiziert, die in der Editor-Repräsentation auf Kanten abgebildet werden. Ein DataItem wird beispielsweise über eine NodeDataConnection an einen Node angebunden. Abbildung 7.3 zeigt beispielhaft, auf welche Weise Kanten wie NodeDataConnection und ControlFlow mit Knoten assoziiert sind.

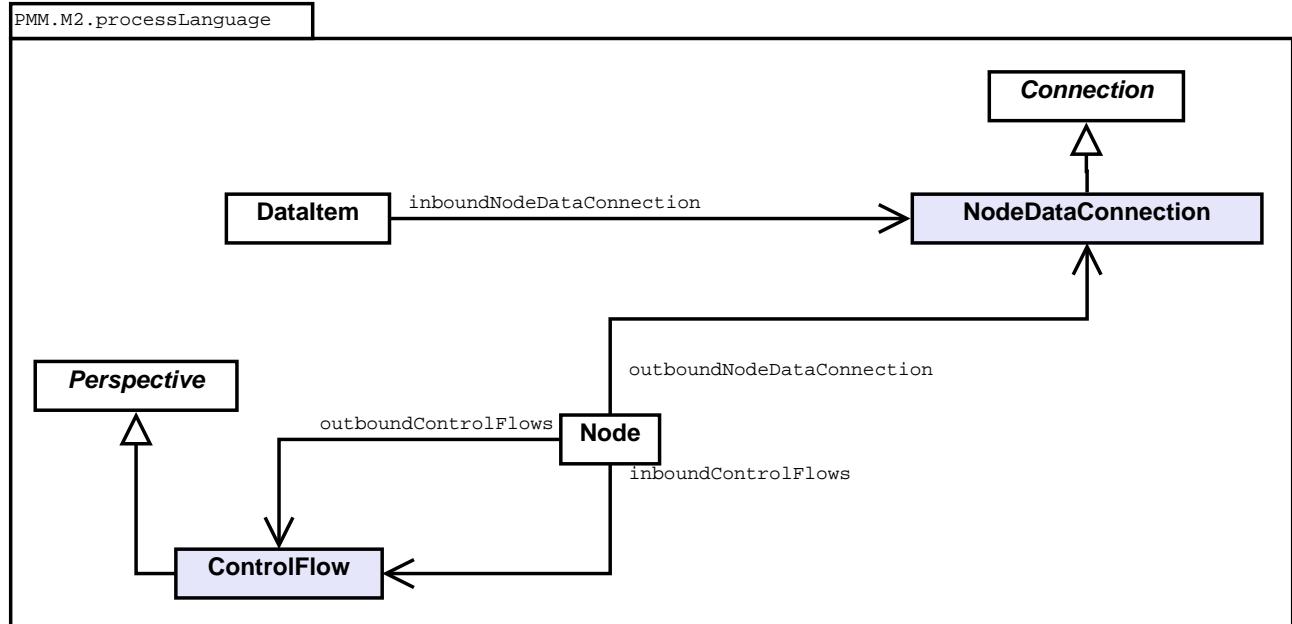


Abbildung 7.3: Die Kanten ControlFlow, NodeDataConnection und deren Assoziationen

7.5 Anwendungsbeispiel: Hinzufügen eines neuen Modellelements

Zur Verdeutlichung des bisher Gesagten soll hier gezeigt werden, wie ein neues Sprachelement zum Prozess-Meta-Modell hinzugefügt werden kann. Anschließend wird die dazugehörige Repräsentation im Editor-Meta-Modell ergänzt.

7.5.1 Änderungen am Prozess-Metamodell

Im Prozess-Metamodell fehlt bisher die Möglichkeit, die operationsbezogene Perspektive der *POPM* (Unterabschnitt 2.1.2) abzubilden. Ein Operations-Element soll durch einen Knoten dargestellt werden, der sich einem Prozess zuordnen lässt.

Die folgenden Änderungen erfolgen im Package PMM.M2.processLanguage.

Zuerst wird die Verbindung zwischen Prozessknoten und dem neuen Operationsknoten hinzugefügt:

```
concept ProcessOrgConnection extends Connection { }
```

Anschließend wird der Knoten definiert:

```
concept OrganizationalPerspective extends Perspective {
    string name;
    0..* concept ProcessOrgConnection inboundProcessOrgConnection;
}
```

Das Attribut `name` kann später vom Modellierungswerkzeug ausgelesen und verändert werden. `InboundProcessOrgConnection` drückt aus, dass dieser Knoten Endpunkt einer `ProcessOrgConnection` sein kann.

Abschließend muss die Verbindung noch im Prozessknoten bekannt gemacht werden:

```
concept Process extends Node {
    0..* concept ProcessOrgConnection outboundProcessOrgConnection;
    // weitere Attribute ...
}
```

Ein `Process` kann somit der Startpunkt einer solchen Verbindung sein.

7.5.2 Änderungen am Editor-Metamodell

Der soeben definierte Organisationsknoten soll durch eine Pyramide dargestellt werden, auf deren Seiten der Wert des Attributs `name` zu lesen ist. Bisher gibt es noch kein Basis-Concept für eine beschriftete Pyramide, also wird diese zum package `figures` im *Editor-Base-Level* (`EMM.M2.figures`) hinzugefügt:

```
concept TextPyramid extends TextLabelNode {
    scalaType = "test.TextPyramid";
}
```

`TextLabelNode` stellt schon alle für einen Text-Knoten benötigten Attribute bereit; daher muss in diesem Concept nur noch der Typ des Grafikobjektes angegeben werden. Wie ein passendes Grafikobjekt erstellt werden kann, wird in der [Fortsetzung dieses Beispiels](#) (Abschnitt 11.6) gezeigt.

Auf dem *Editor-Definition-Level* kann nun die Repräsentation für den Organisationsknoten-Typen im package `EMM.D1.nodeFigures` als Instanz der `TextPyramid` definiert werden.

Als Vorlage wird das vorhandene Concept `Process` genutzt:

```
TextPyramid OrganizationalNode {
    modelElementFQN = pointer PMM.M2.processLanguage.OrganizationalPerspective;
    displayAttrib = "name";
    toolingAttrib = "name";
    toolingTitle = "Organizational Unit";
    // weitere Attribute, die nicht zwingend geändert werden müssen ...
}
```

Die unter [Paket „figures“](#) (Unterabschnitt 7.2.2) erläuterten Attribute werden hier am Beispiel gezeigt:

- `modelElementFQN` gibt das zugehörige Concept aus dem Prozess-Metamodell an, das neu definiert wurde.
- `displayAttrib` legt fest, dass das Attribut `name` jenes Concepts als Text angezeigt werden soll.

Knoten werden nach dem Typ-Verwendungs-Konzept erstellt. `OrganizationalPerspective` ist also ein „Metatyp“, von dem im Modellierungswerkzeug erst konkrete Typen erstellt werden müssen.

- `toolingTitle` legt die Bezeichnung des Metatyps im Modellierungswerkzeug auf „Organizational Unit“ fest.
- `toolingAttrib` gibt an, dass ein erzeugter Typ mit dem Wert seines `name`-Attributs benannt wird.

Im nächsten Schritt wird eine Repräsentation für die neu definierte Verbindung zwischen Prozess und Organisationsknoten im package `EMM.D1.connectionFigures` festgelegt. Als Vorlage dient das `nodeDataEdge`-Concept:

```

ColoredLine ProcessOrgEdge {
    modelElementFQN = pointer PMM.M2.processLanguage.ProcessOrgConnection;
    toolingName = "Process-Organizational Assoc";
    outboundAttrib = "outboundProcessOrgConnection";
    inboundAttrib = "inboundProcessOrgConnection";
    // weitere Attribute ...
}

```

Der Wert von `inboundAttrib` entspricht dem Namen des Attributs im `OrganizationalPerspective-Concept` aus dem Prozess-Metamodell. So wird dem Werkzeug mitgeteilt, dass eingehende Verbindungen im Domänenmodell dem Attribut `inboundProcessOrgConnection` zugewiesen werden sollen.

3D-Visualisierung von Prozessen

Im Folgenden wird die Visualisierung von Prozessen im i>PM3D-Prototypen vorgestellt, wie sie durch das im vorherigen Kapitel vorgestellte Editor-Metamodell festgelegt wird. Außerdem werden durch die Implementierung vorgegebene Aspekte angesprochen, welche aber weitgehend unabhängig von der Prozessmodellierung sind. Dabei werden auch Hinweise gegeben, welche beim Hinzufügen von neuen Modellfiguren oder Ändern von Visualisierungsparametern beachtet werden sollten. Anschließend wird gezeigt, welche Nutzungsmöglichkeiten der dritten Dimension sich in i>PM3D umsetzen lassen und welche Erweiterungsmöglichkeiten bestehen, die für eine höhere Benutzerfreundlichkeit und Verständlichkeit sinnvoll sind.

8.1 Grundlegende Darstellung der grafischen Elemente

Wie im vorherigen Kapitel unter *Editor-Base-Level* (Abschnitt 7.2) erläutert, werden auf dem Editor-Base-Level grundlegende Figuren und deren Darstellung durch grafische Objekte im Modellierungswerkzeug definiert. Die konkreten Repräsentationen für bestimmte Typen aus dem Prozessmodell werden auf dem Editor-Definition-Level festgelegt. In den Metamodellen wurde schon vorgegeben, dass ein graphbasierter Visualisierungsansatz genutzt wird. Anwender, die bereits Erfahrung mit verbreiteten grafischen 2D-Prozessmodellierungssprachen haben, sollten durch das Aussehen der Modellelemente möglichst intuitiv verstehen können, welche Konzepte aus der Prozessmodellierung dargestellt werden.

8.1.1 Knoten

Für die Darstellung von Informationen auf den Knoten gibt es durch die auf dem *Editor-Base-Level* (Abschnitt 7.2) definierten Basis-Figuren `TextLabelNode` und `TexturedNode` grundsätzlich zwei Möglichkeiten. Die Beschriftung von `TextLabelNodes` kann dazu verwendet werden, textuelle Attribute aus dem Prozessmodell direkt anzuzeigen (*Anforderung (h)* (Abschnitt 1.3)).

Es sollten möglichst einfache, dreidimensionale geometrische Körper mit möglichst ebenen Seitenflächen wie Würfel oder Quader gewählt werden. Ebene Flächen eignen sich besonders gut zur Darstellung von Information; gekrümmte Flächen beeinträchtigen besonders die Lesbarkeit von (längerem) Textdarstellungen. Bei Würfeln oder ähnlichen Körpern ist es auch relativ einfach, einen (dreidimensionalen) Rahmen zu zeichnen, dessen Verwendung weiter unten in Abschnitt 8.2 dargestellt wird.

Abbildung 8.1 zeigt zwei Prozessknoten, auf welchen die Prozess-Funktion als Text angezeigt wird.

Da die Erstellung von Knoten nach dem *Typ-Verwendungs-Konzept* (Abschnitt 2.3) erfolgt, lässt sich die Visualisierung für jeden Knoten individuell anpassen. In der Abbildung wurde beim rechten Knoten zur Laufzeit die Hintergrundfarbe geändert.

Texte werden nach Bedarf an Wortgrenzen auf mehrere Zeilen verteilt und zentriert angezeigt. Weitere Details zur Schriftdarstellung können im Kapitel zur *Render-Bibliothek* (Unterabschnitt 11.5.2) nachgelesen werden.



Abbildung 8.1: Zwei Prozessknoten; links im Ursprungszustand, rechts als angepasste Verwendung (Screenshot aus i>PM3D)

Andererseits können Grafiken (Texturen) genutzt werden, um die Bedeutung eines Knotentyps zu visualisieren. So steht ein Pluszeichen für einen AND-Connector, wie in Abbildung 8.2 gezeigt wird.

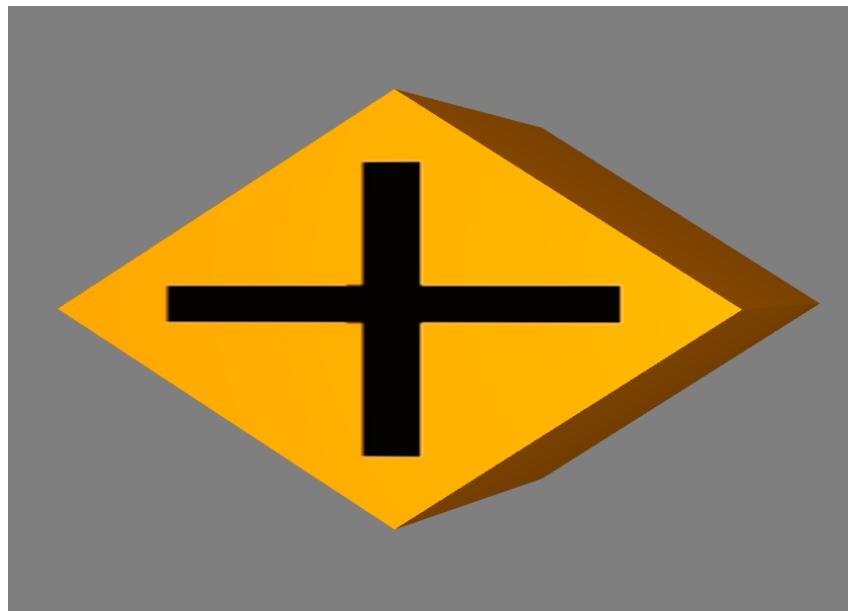


Abbildung 8.2: AND-Connector (Screenshot aus i>PM3D)

Blickwinkelabhängige Darstellung von Informationen

Durch die freie Beweglichkeit und Rotationsmöglichkeit der Kamera sowie der *Objekte* (Unterabschnitt 5.1.1) ergeben sich sehr unterschiedliche Beobachtungsperspektiven. Objekte können so von allen Seiten betrachtet werden. Trotzdem soll sichergestellt werden, dass Texte oder Symbole auf den

Objekten jederzeit erkennbar sind. Daher werden diese grundsätzlich auf allen Seiten dargestellt. Jedoch führt dies bei bestimmten Drehpositionen zu störenden und möglicherweise verwirrenden Darstellungen, wenn beispielsweise bei einem Würfel zwei oder sogar drei Seiten zu sehen sind, die dasselbe anzeigen.

Um dies zu verbessern, werden die Seiten abhängig von Betrachtungswinkel dargestellt. Wird eine Seite vom Benutzer weggedreht, wird die Schrift oder Textur nach und nach „ausgeblendet“, indem die Vordergrundfarbe je nach Winkel mit der Hintergrundfarbe gemischt wird. Ab einer gewissen Abweichung wird nur noch die Hintergrundfarbe angezeigt. So ist nur eine Seite deutlich zu erkennen und der Betrachter wird nicht durch die anderen Seiten abgelenkt.

Abbildung 8.3 zeigt links den ungünstigsten Grenzfall, in dem alle Seiten gleich deutlich dargestellt werden. Dies ist der Fall, wenn der Betrachter direkt auf eine Ecke blickt. Rechts ist der Knoten günstiger ausgerichtet und die Schrift ist auf der rechten Seite des Objekts kaum mehr zu erkennen.



Abbildung 8.3: Schriftdarstellung bei direkter Sicht auf eine Ecke (links) und bei günstigerer Perspektive (Screenshot aus i>PM3D)

Berücksichtigung der Eingabemethoden

Da i>PM3D nicht nur die klassischen Desktop-Bedienung mit Maus und Tastatur erlauben, sondern auch zur Evaluierung von neuartigen Eingabegeräten eingesetzt werden soll, müssen auch die Besonderheiten dieser Eingabemethoden berücksichtigt werden. Die im Projekt verwendeten 3D-Eingabegeräte [Buc12] haben nur eine relativ begrenzte Genauigkeit bei der Auswahl und Platzierung von Objekten. Vor allem ungeübten Benutzern kann es schwerfallen, Objekte zu selektieren und zu bewegen, besonders wenn die Objekte relativ klein sind.

Dies ist auch ein Grund, eine Graphdarstellung mit möglichst einfachen Objekten zu verwenden. Es wird deswegen auch verzichtet, Elemente nach dem geometrischen Visualisierungsansatz ineinander zu schachteln, wie es bei 2D-Werkzeugen wie *i>PM2* (Unterabschnitt 2.2.2) zu sehen war. Es ist sinnvoll, Quader (oder annähernd quaderförmige Geometrien) einzusetzen, da Knoten in die physikalische Simulation eingebunden sind und Quader von der verwendeten Physikkomponente direkt unterstützt werden¹. Die physikalische Simulation wird von den Eingabegeräten für die Selektion von Elementen genutzt, wie von [Buc12] beschrieben.

8.1.2 Kanten

Eine Kante sollte optisch leicht als Verbindung zwischen zwei Knoten erkannt werden können; außerdem muss ggf. visualisiert werden, welche Richtung die Kante besitzt. In i>PM3D werden Kanten durch einen

¹Von der Physikkomponente werden auch Kugeln unterstützt, allerdings ist die Verwendung von Quadern bisher fest in der Implementierung von i>PM3D vorgegeben.

(in y-Richtung) gestreckten 3D-Quader dargestellt, der vom Startknoten bis zum Endknoten reicht. Die Länge und Ausrichtung der Kanten wird automatisch angepasst, wenn die beteiligten Knoten im Raum verschoben werden. Dies wird von der in [Hol12] beschriebenen Editor-Komponente übernommen.

Die durch das Concept *TexturedConnection* (siehe *Editor-Base-Level* (Abschnitt 7.2)) bereitgestellte texturierte Verbindung dient dazu, gerichtete Kanten zu visualisieren. Eine Möglichkeit ist es, eine Textur mit farblich vom Hintergrund abgehobenen Dreiecken zu verwenden, die so platziert sind, dass an zwei Ecken der Verbindung ein Pfeil entsteht.

Abbildung 8.4 zeigt als Beispiel zwei Prozesse, die mit einem Kontrollfluss verbunden sind. Der Kontrollfluss läuft von „Pizza backen“ nach „Pizza verpacken“.

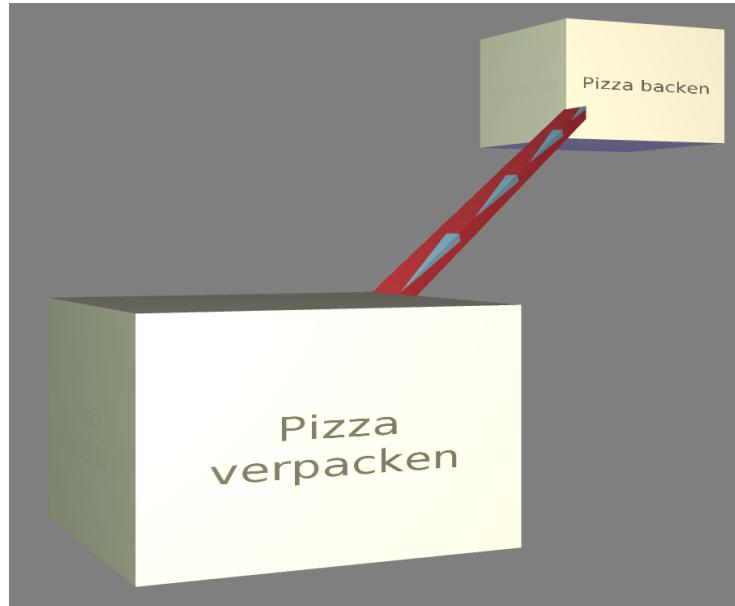


Abbildung 8.4: Gerichtete Kontrollflusskante von „Backen“ nach „Verpacken“ (Screenshot aus i>PM3D)

8.1.3 Szenenobjekte

Zusätzlich zu den Elementen des eigentlichen Prozessmodells gibt es noch die Möglichkeit, beliebige 3D-Grafikobjekte in die Szene einzufügen, die im Editor-Metamodell als *SceneryObject* bezeichnet werden (*Anforderung (h)*). Solche Szenenobjekte können zum Beispiel dafür eingesetzt werden, Abbilder von realen Objekten anzuzeigen. Szenenobjekte können genauso wie Knoten, selektiert, frei bewegt, skaliert und rotiert werden. Sie besitzen aber sonst keine anderen Möglichkeiten, das Erscheinungsbild zu beeinflussen.

8.2 Visualisierungsvarianten für interaktive Modelleeditoren

Da die hier vorgestellte Visualisierung in einem interaktiven Modelleeditor eingesetzt wird, ergibt sich noch die Anforderung, Visualisierungsvarianten der Modellelemente zu unterstützen. So sollen Interaktionen des Benutzers mit den Modellobjekten sichtbar gemacht werden, indem die Visualisierung der Objekte temporär verändert wird. Diese Modifikationen werden nicht im Editor-Usage-Model persistiert; daher werden alle Objekte im Normalzustand angezeigt nachdem ein Modell neu geladen wurde.

8.2.1 Hervorhebung

Diese Variante wird dafür eingesetzt, ein Objekt kurzzeitig beim Überfahren durch einen Cursor eines Eingabegeräts hervorzuheben. Dargestellt wird das abhängig von der Helligkeit der Grundfarbe des Objekts durch eine Aufhellung bzw. einer Abdunkelung der Farbe. Der Farnton wird dabei nicht verändert. Abbildung 8.5 zeigt im Vergleich ein hervorgehobenes Datenelement und eines im Normalzustand (rechts).

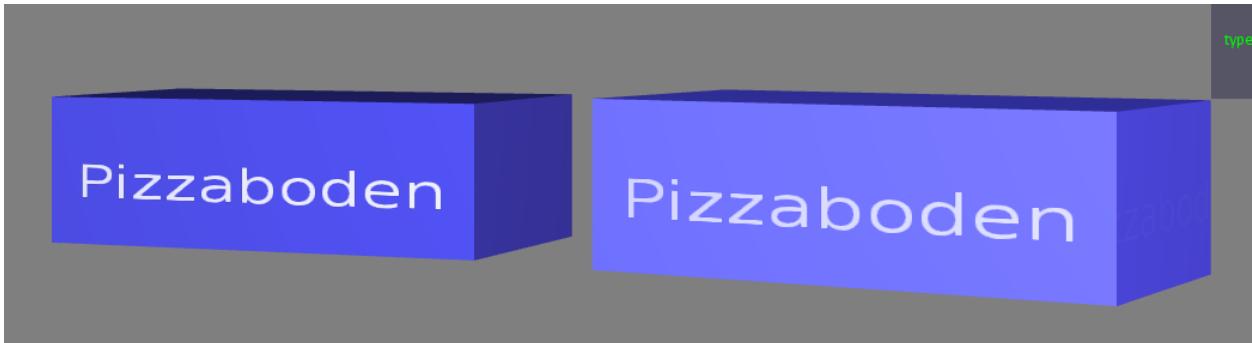


Abbildung 8.5: Datenknoten, normal (links) und hervorgehoben (Screenshot aus i>PM3D)

8.2.2 Selektion

Prozessmodellelemente und Szenenobjekte können durch den Benutzer ausgewählt werden. Selektierte Objekte sollen von unselektierten Objekten auch bei großer Entfernung und ungünstigen Blickwinkeln unterscheidbar sein, wobei aber jederzeit noch erkennbar sein muss, um welche Art von Modellelement es sich handelt.

Die Visualisierung des Selektionszustandes soll daher möglich auffällig sein, ohne das Erscheinungsbild allzu stark zu beeinflussen. Um die Selektion von der Hervorhebung unterscheidbar zu machen, wird für die Selektion der Rand des Objekts in der Komplementärfarbe eingefärbt. Die Definition des „Rands“ ist je nach Objekttyp unterschiedlich². In Abbildung 8.6 sind zwei selektierte Knoten zu sehen.

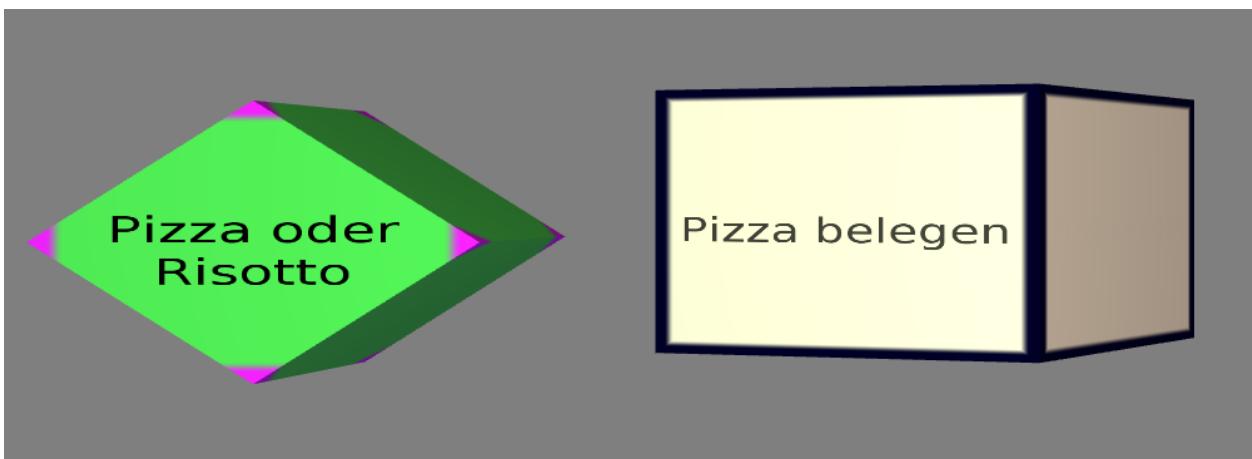


Abbildung 8.6: Entscheidungsknoten und Prozess im selektierten Zustand (Screenshot aus i>PM3D)

²Der Rand ist über die Texturkoordinaten definiert. Siehe (Unterabschnitt 11.5.1).

8.2.3 Deaktivierung

Objekte können durch den Modelleditor deaktiviert werden. Welche Bedeutung dies hat, wird vom Editor festgelegt. Zur Visualisierung dieses Zustandes wird das Objekt transluzent in einem Grauton dargestellt, der von der normalen Farbe abhängig ist. So kann man auch Elemente erkennen, die hinter dem deaktivierten liegen und von diesem verdeckt werden. Abbildung 8.7 zeigt einen deaktivierten Prozess, hinter dem sich ein anderer Prozess befindet.

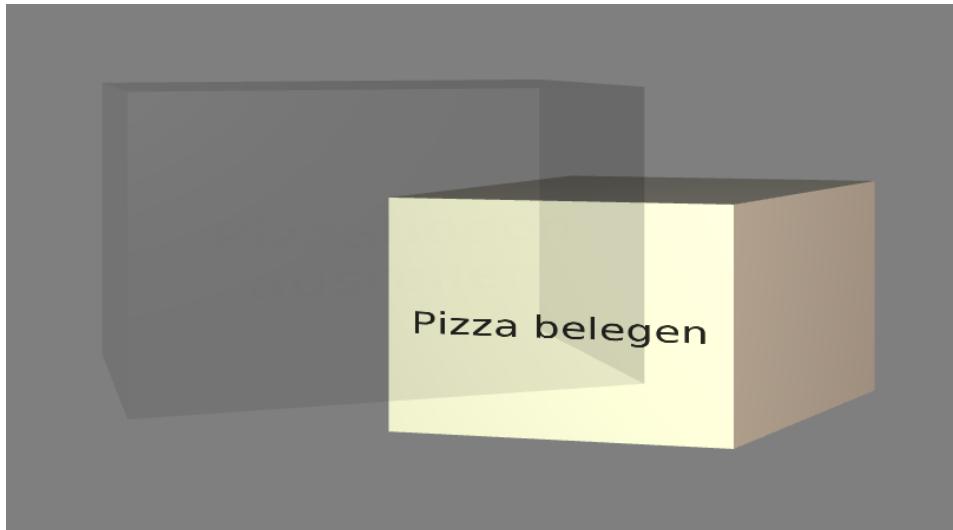


Abbildung 8.7: Deaktivierter (vorne, durchsichtig) und aktiver Prozess (Screenshot aus i>PM3D)

Die drei gezeigten Visualisierungsvarianten können frei kombiniert werden. So ist es möglich, ein gleichzeitig hervorgehobenes, selektiertes und deaktiviertes Modellelement darzustellen.

8.3 2D-Modellierungsflächen

Für eine übersichtliche Darstellung des Prozessmodells ist es häufig erwünscht, Elemente in einer bestimmten Weise anzurorden. Zur Vereinfachung der Platzierung werden in 2D-Modellierungswerkzeugen oft im Hintergrund dargestellte Gitter genutzt, die eine optische Hilfe darstellen. Noch hilfreicher können „magnetische“ Gitter sein, die grob in der Nähe platzierte Objekte automatisch auf feste, regelmäßige Positionen verschieben.

Um dies zu erreichen, wird die *Physikkomponente* (Unterabschnitt 5.2.1) genutzt. Sobald sich ein Objekt nahe genug an einer solchen 2D-Modellierungsfläche befindet, wird es nach dem Loslassen durch den Benutzer (Deselektion) von der „Gravitation“ der Ebene angezogen. Das Objekt bewegt sich solange, bis dessen Mittelpunkt die Fläche erreicht hat und dort automatisch angehalten wird. Näheres zur Implementierung dieser „Gravitationsflächen“ findet sich in [Buc12].

Grafisch werden diese Flächen transluzent dargestellt, wobei darauf Gitterlinien zu erkennen sind. Diese Linien haben allerdings keine physikalische Bedeutung, sondern dienen nur als optische Platzierungshilfe.

Abbildung 8.8 zeigt zwei solcher Flächen ober- und unterhalb des Betrachters.

8.4 Beleuchtung

Für die Beleuchtung der Szene werden mehrere Lichtquellen eingesetzt. Die primäre Lichtquelle befindet sich direkt an der Kamera und bewegt sich mit dieser. Die Lichtfarbe ist weiß, also wird der Farnton der



Abbildung 8.8: Zwei leere Modellierungsflächen (Screenshot aus i>PM3D)

beleuchteten Objekte unverfälscht dargestellt.

Zur Verbesserung der Orientierung befindet sich jeweils eine weniger intensive, farbige Lichtquelle an drei festen Positionen unterhalb (blau), links (grün) und rechts (rot) der Szene, von der Startposition der Kamera aus gesehen. So soll es für den Benutzer leichter zu erkennen sein, welche Seite der Objekte in Bezug auf die Ausgangsposition nach unten, links beziehungsweise nach rechts zeigt.

Die von der *Render-Bibliothek* (Kapitel 11) bereitgestellten Lichtquellen nach dem Phong-Lichtmodell [Pho75] sorgen für eine relativ realistische Beleuchtung bei vertretbarem Rechenaufwand. Für die Visualisierung von 3D-Graphmodellen stellt sich die Frage, wie die Lichtparameter am besten gewählt werden sollten, um eine möglichst hohe Lesbarkeit und eine gute Orientierung im Raum zu ermöglichen.

Im Phong-Lichtmodell wird das von einem Objekt reflektierte Licht in drei Beiträge unterschieden [AHH08].

Der Hauptanteil des reflektierten Lichts wird im Normalfall vom *diffuse*-Anteil beigesteuert, welcher abhängig vom Winkel zur Lichtquelle ist. Von der Lichtquelle eher abgewandte Seiten erscheinen daher dunkel, was sich ungünstig auf die Erkennbarkeit von Informationen auswirken kann.

Um dies auszugleichen, kann der *ambient*-Anteil (Umgebungslicht) erhöht werden, der vom Winkel unabhängig ist. Wird dieser zu hoch gesetzt, leidet allerdings der räumliche Eindruck.

Der *specular*-Anteil erzeugt spiegelnde Reflexionen auf Objekten, die auch von der Betrachterposition relativ zum Objekt abhängen. Dieser Anteil kann folglich die räumliche Orientierung unterstützen. Allerdings führt die starke Aufhellung an bestimmten Stellen dazu, dass sich Text dort schlecht ablesen lässt.

Außerdem kann bei (OpenGL)-Lichtquellen noch angegeben werden, wie stark die Helligkeit mit steigender Entfernung von der Lichtquelle abfällt. Hierdurch kann der Tiefeneindruck verbessert werden.

Ein starker Abfall der Beleuchtung führt aber zu Problemen, wenn gleichzeitig Objekte mit Text in der Nähe der Lichtquelle und weit entfernt in lesbarer Form dargestellt werden sollen. Objekte in der Nähe werden zu hell dargestellt, während weit entfernte Objekte zu dunkel sind. Genauso ergibt sich

bei gerichteten Verbindungen, die sich weit im Hintergrund befinden, das Problem, dass die darauf abgebildeten Richtungsmarkierungen schlecht zu erkennen sind.

Insgesamt hat sich bei Versuchen gezeigt, dass es schwierig ist, die Lichtparameter so zu setzen, dass eine in allen Situationen brauchbare Beleuchtung entsteht.

8.5 Visualisierung eines Beispieldatenmodells

Abbildung 8.9 zeigt den in i>PM3D modellierten Prozess „Pizza produzieren“. Zusätzlich zu den bisher gezeigten Knoten ist hier noch der Start-/Stop-Knoten – als roter, abgerundeter Quader dargestellt – zu sehen. Neben der funktionalen Perspektive, vertreten durch die Start-Stop-Knoten, Prozessknoten und zwei AND-Konnektoren, wird auch die Datenperspektive mit Datenknoten und dazwischen verlaufenden Datenflüssen dargestellt. Die Abbildung zeigt, wie sich durch das Deaktivieren von Knoten ein Teil des Modells (hier die Datenperspektive) ausblenden lässt, um die Ansicht auf momentan interessante Teile zu fokussieren und die Übersichtlichkeit zu erhöhen. Aktiviert ist nur der Datenfluss der Pizza selbst, von „Pizza belegen“ bis hin zum Ende des Prozesses; die restlichen Datenknoten und damit auch die dazwischenliegenden Verbindungen sind deaktiviert. Die dünnen, blauen Kanten zwischen den Prozessen und Datenknoten stellen NodeDataConnections dar, welche Nodes und DataItems des Prozessmodells miteinander assoziieren.

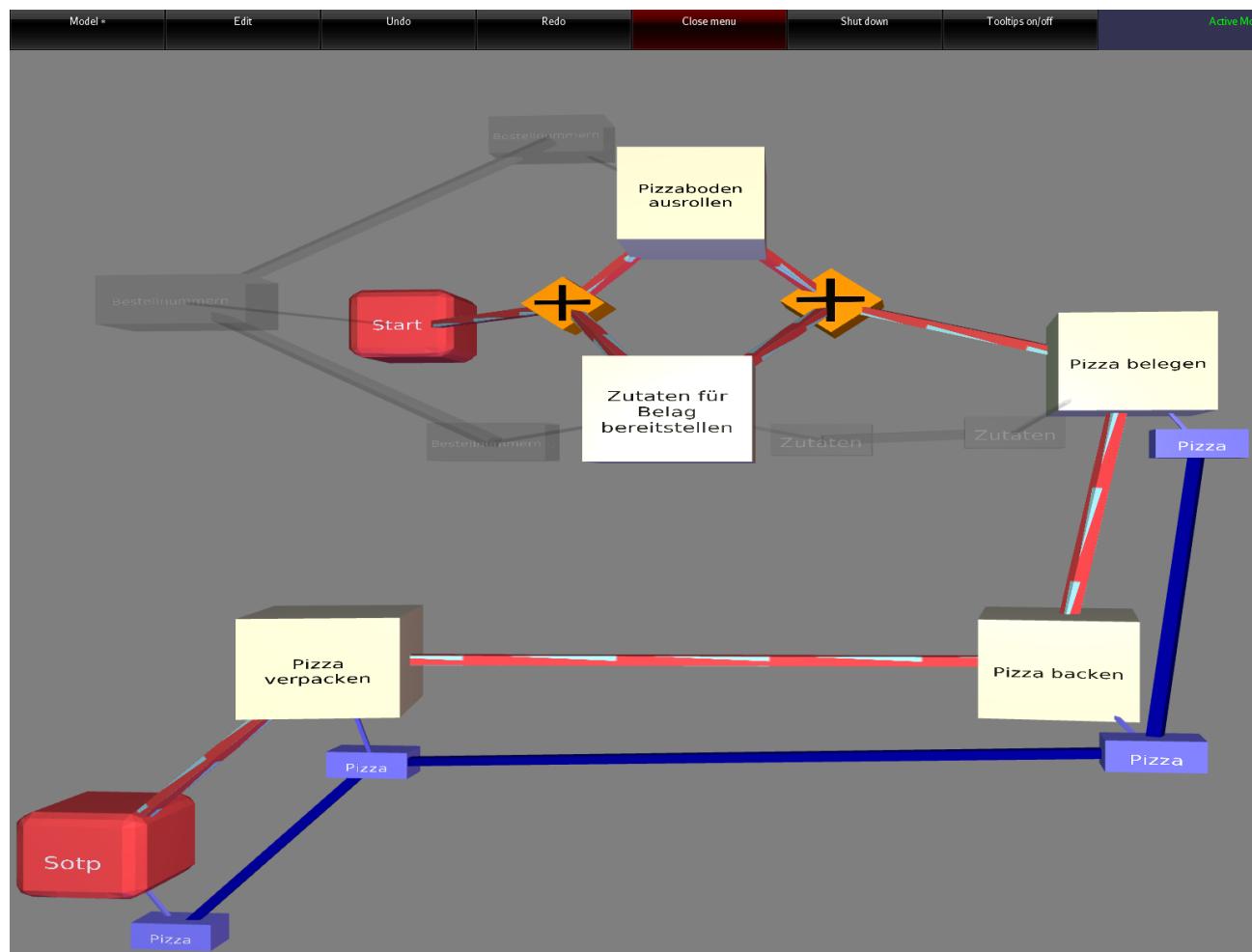


Abbildung 8.9: Beispiel für einen Prozess mit deaktivierten Datenfluss-Elementen (Screenshot aus i>PM3D)

8.6 Umsetzung von verschiedenen Nutzungsmöglichkeiten der dritten Dimension

In diesem Abschnitt soll gezeigt werden, welche [in](#) Abschnitt 3.4 genannten Nutzungsmöglichkeiten sich mit dem bisherigen Prototypen umsetzen lassen oder welche Erweiterungen dafür vorgenommen werden sollten. Die folgende Tabelle zeigt hierzu eine Übersicht über verschiedene Verwendungen der dritten Dimension und deren Umsetzung in i>PM 3D, welche anschließend näher ausgeführt wird.

3. Dimension verwendet für	aktuell umsetzbar in i>PM3D?
unterschiedl. Beziehungstypen	ja
Darstellung von Modellattributen	ja, aber manuelle Platzierung nötig
Zeitachse	ja, aber manuelle Platzierung nötig. 2D-Flächen als Swimlanes
mehrere / hierarchische Modelle	ja, aber ohne Verbindungen zw. den Modellen
mehrere Modelltypen	nein
Einbettung in virtuelle Umgebung	ja, aber ohne Animation

8.6.1 Visualisierung von Attributen

Welche Information durch den Abstand (die „Tiefe“) vermittelt werden soll, wird durch den Benutzer festgelegt. Durch die freie Platzierbarkeit lässt sich prinzipiell jede gewünschte geometrische Anordnung erreichen. Elemente nach ihrer Wichtigkeit anzuordnen, wie es [in](#) (Kapitel 3) vorgeschlagen wurde, ist damit möglich. Außerdem lassen sich durch unterschiedliche Entfernung der Knoten von einem Fixpunkt beliebige Attribute aus dem Prozessmodell visualisieren.

Allerdings ist der Nutzen dadurch eingeschränkt, dass der Benutzer die Positionierung der Elemente manuell vornehmen muss. Bei einer Änderung der Attribute muss auch die Tiefen-Visualisierung entsprechend angepasst werden. Dies ist allerdings aufwändig und fehleranfällig. Um dies für den Anwender effizient zu gestalten, müssten Algorithmen integriert werden, welche die Platzierung anhand von Attributwerten aus dem Prozessmodell automatisch übernehmen und die Visualisierung auch bei Änderungen am Prozessmodell zur Laufzeit synchron halten.

Eine gewisse Hilfe, besonders für diskrete Attribute, können die 2D-Modellierungsflächen darstellen. So kann jeder Fläche ein bestimmter Wert zugewiesen werden. Alle Elemente, die diesen Wert aufweisen, werden auf der passenden Fläche platziert. Die Fläche unterstützt den Benutzer bei der Anordnung, indem sie die Objekte automatisch in einer Ebene platziert, wenn sie in der Nähe der Fläche abgelegt werden. Um Objekte beispielsweise in die Kategorien „wichtig“ und „unwichtig“ einzuteilen, ließen sich zwei (parallele) Flächen definieren, anhand derer die Objekte sortiert werden können. Da bisher nur die Platzierung senkrecht zur Fläche automatisch erfolgt, sollte in einer Erweiterung von i>PM3D die Möglichkeit hinzugefügt werden, 2D-Layout-Algorithmen anzuwenden, um beispielsweise Knoten mit bestimmten Eigenschaften zu gruppieren oder für eine kreuzungsfreie Darstellung der Verbindungen in der Ebene zu sorgen.

8.6.2 Zeitliche Abläufe

Eine weitere Nutzungsmöglichkeit für die dritte Dimension ist die Darstellung von zeitlichen Abläufen. Diese werden in einem Prozess üblicherweise schon durch den Kontrollfluss grob vorgegeben, allerdings

lassen sich gewisse Aspekte mit einer sinnvollen 3D-Anordnung weiter betonen. Die Zeitdauer lässt sich durch die Entfernung zwischen Prozessknoten visualisieren. Einerseits lassen sich Prozessschritte, die (nahezu) gleichzeitig ausgeführt werden, und zugehörige Knoten – wie assoziierte Daten – gemeinsam auf einer 2D-Ebene darstellen, ähnlich wie es von [Gil](#) (Unterabschnitt 3.1.4) anhand von 3D-UML-Sequenzdiagrammen gezeigt wurde. Andererseits kann man 2D-Ebenen als „Swimlanes“ nutzen, wie sie aus BPMN bekannt sind. Dazu werden Prozessschritte auf parallel verlaufenden Flächen gruppiert, die zu einer bestimmten ausführenden Entität gehören.

8.6.3 Veranschaulichung von Beziehungen

Durch die Anordnung der Elemente die Bedeutung von Beziehungen zu unterstreichen ist ebenfalls möglich, wobei wiederum die 2D-Flächen hilfreich sind. So kann durch die Positionierung der Knoten auf einer Fläche leicht verdeutlicht werden, dass Kanten, die parallel zur Fläche verlaufen, eine andere Bedeutung haben als diejenigen, die aus der Ebene herausragen und zu Elementen führen, die auf einer anderen Fläche platziert sein können. Bei [GEF3D](#) Unterabschnitt 3.1.6 wurde dies genutzt, um gleichzeitig Beziehungen innerhalb eines Modells als auch Beziehungen zu Elementen eines anderen Modells (möglicherweise auch anderen Typs) zu visualisieren. i>PM 3D hat die Fähigkeit, mehrere Modelle gleichzeitig zu laden, allerdings werden Verbindungen zwischen mehreren Modellen und verschiedene Modelltypen zur gleichen Zeit noch nicht unterstützt ([siehe](#) Kapitel 9). Dies sollte in einer weiteren Entwicklung vorrangig hinzugefügt werden, um diese Anwendung der dritten Dimension zu erlauben, welche deutliche Vorteile zu 2D-Darstellungen verspricht.

8.6.4 Hierarchische Darstellung und mehrere Modelle

Die Visualisierung von hierarchischen Diagrammen in derselben 3D-Szene ist für die Darstellung von kompositen Prozessen hilfreich. Ein Verfeinerungsmodell sollte in einem separaten Modell abgelegt werden, welches im grobgranularen Modell durch einen kompositen Prozessknoten vertreten wird. Wird das Verfeinerungsmodell nach Bedarf „ausgeklappt“, sollte eine optische Verbindung zum Knoten im übergeordneten Modell die Zusammengehörigkeit anzeigen, wie es beispielsweise von [McIntosh](#) (Unterabschnitt 3.1.2) gezeigt wurde. Dies ist noch nicht möglich, da Verbindungen zwischen Modellen nicht erlaubt sind. Durch die Fähigkeit von i>PM3D, manuell mehrere Modelle laden zu können [Hol12], lassen sich solche Hierarchiestufen immerhin schon in separaten Modellen ablegen und in derselben 3D-Szene anzeigen. Weiterhin wäre eine direkte Unterstützung durch den Editor angebracht, welcher die Subdiagramme bei einer Benutzerinteraktion mit dem kompositen Prozessknoten automatisch öffnen sollte.

Da sich der Benutzer frei in der Szene bewegen kann, können unterschiedliche Betrachtungsperspektiven eingenommen werden. Die Perspektive kann so gewählt werden, dass die momentan für den Benutzer besonders interessanten Aspekte der 3D-Szene deutlich zu erkennen sind. Besonders hilfreich ist dies, wenn mehrere Modelle gleichzeitig dargestellt werden. Werden beispielsweise zwei Modelle angezeigt, wobei das eine aktuell im Vordergrund dargestellt wird und das andere im Hintergrund liegt, ist das Modell im Vordergrund wegen des geringeren Abstands zum Benutzer besser zu erkennen. [Abbildung 8.10](#) zeigt diese Situation. Soll nun das andere Modell genauer betrachtet werden, kann dies erreicht werden, indem die Szene einfach von hinten betrachtet wird. [Abbildung 8.11](#) stellt dieselben Modelle dar, allerdings hat sich der Benutzer erst um 180° gedreht und dann etwas nach hinten bewegt, um das Modell mit den blauen Prozessknoten zu fokussieren.

Die andere Möglichkeit ist, dass sich der Benutzer zum blauen Modell hinbewegt, wobei das gelbe dann hinter dem Benutzer liegt und nicht mehr zu sehen ist.

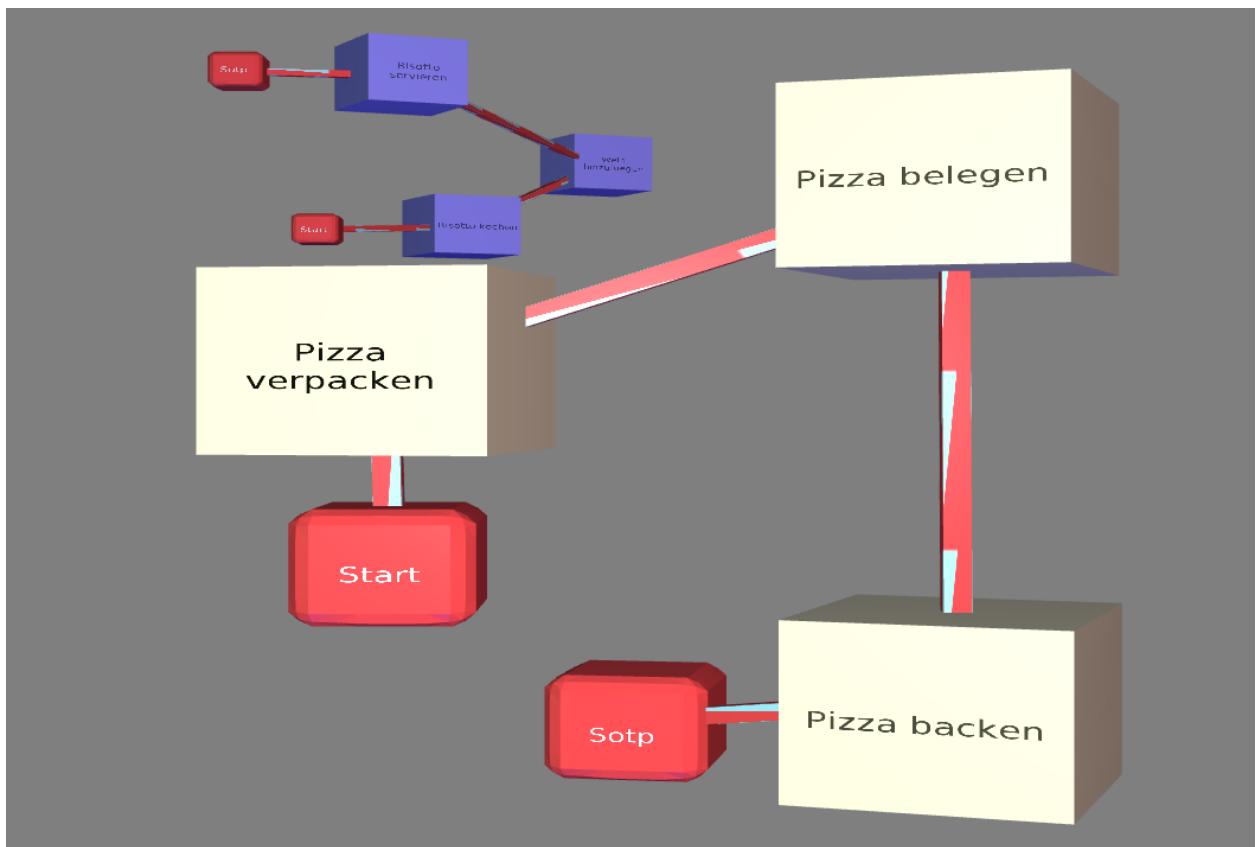


Abbildung 8.10: Darstellung von zwei Prozessen, Fokus auf gelbem Modell (Screenshot aus i>PM3D)

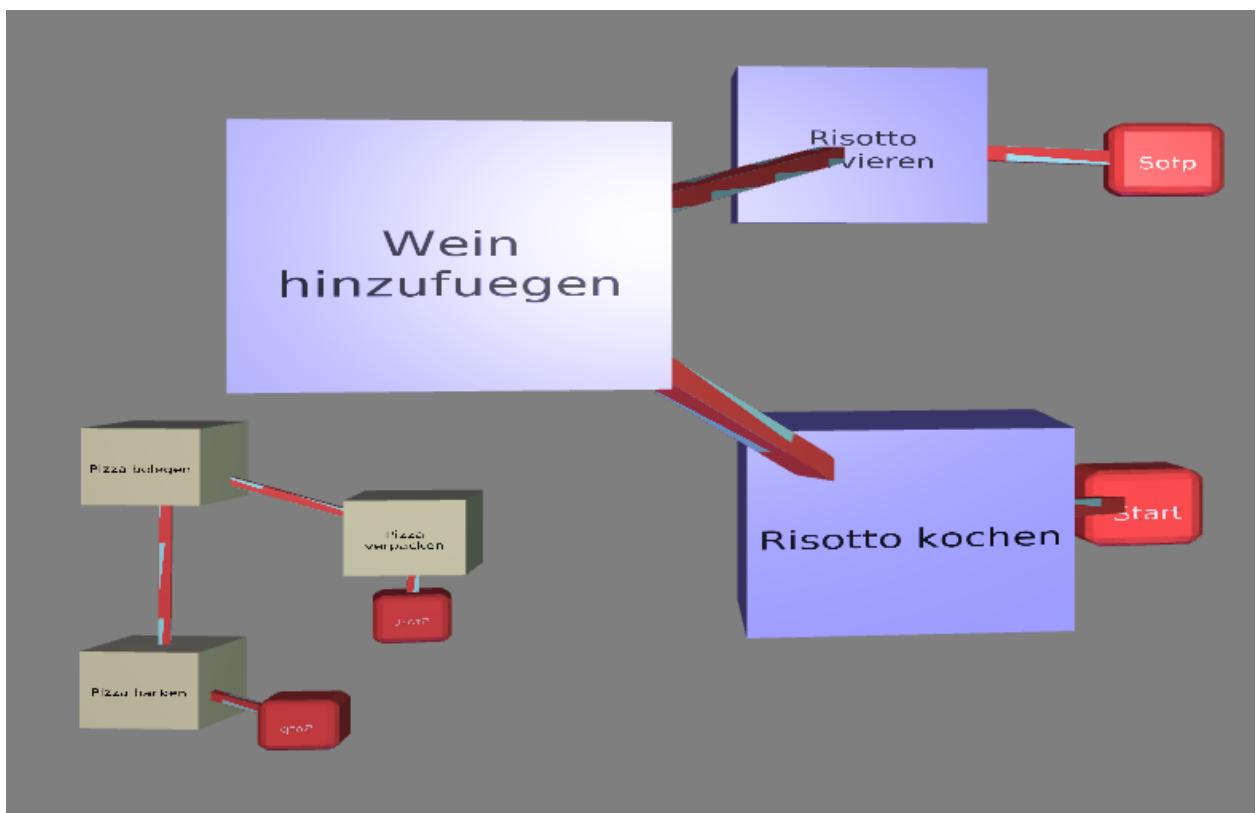


Abbildung 8.11: Darstellung von zwei Prozessen, Fokus auf blauem Modell (Screenshot aus i>PM3D)

8.6.5 Einbettung in eine virtuelle Umgebung

Durch die Möglichkeit, mittels Szenenobjekten beliebige 3D-Objekte einzubinden, lässt sich der Prozessgraph in eine virtuelle Ausführungsumgebung einbetten. Die Szenenobjekte können in i>PM3D aus COLLADA-Dateien geladen werden, welche von Anwendungen wie beispielsweise *Blender* [[www:blender](#)] erstellt werden können. So lässt sich grundsätzlich jede beliebige 3D-Szene aus einzelnen Objekten aufbauen, um die in Unterabschnitt 3.4.2 gezeigten Nutzungsmöglichkeiten zu realisieren.

Durch den modularen Aufbau des Systems auf Basis von Simulator X ist die Integration weiterer Funktionalität durch die Implementierung einer neuen Komponente einfach möglich. Bewegungsanimationen von Szenenobjekten – beispielsweise Werkstücken – sind bisher noch nicht integriert. Um dies nachzurüsten, müsste eine „Animationskomponente“ implementiert werden. Eine solche Komponente könnte Objekte automatisch oder vom Benutzer gesteuert (bspw. durch ein Kommando „zum nächsten Datenknoten bewegen“) entlang von Datenflüssen bewegen und so die Dynamik des Prozessablaufs veranschaulichen. Für *Szenenobjekte* (Unterabschnitt 7.2.2) kann eine physikalische Repräsentation in Form eines Quaders oder einer Kugel definiert werden, um diese in die Physiksimulation aufzunehmen. Dies kann durch eine Animationskomponente genutzt werden, um Kollisionen zwischen bewegten Szenenobjekten zu erkennen. Eine andere Möglichkeit ist die Integration von Komponenten, die automatische Pfadsuch-Algorithmen wie beispielsweise den A*-Algorithmus [[HNR72](#)] anwenden, welche unter anderem dazu genutzt werden können, optimale Laufwege zu finden.

8.7 Weitere Entwicklungsmöglichkeiten

Die momentan umgesetzte Visualisierung von Prozessen zeigt nach unserer³ Ansicht, dass eine 3D-Ansicht auf Prozessdiagramme durchaus praktikabel ist. Es zeigten sich bei ersten Versuchen mit dem i>PM3D Prototypen einige Probleme in Hinblick auf die Visualisierung, die teilweise schon angesprochen wurden oder im Folgenden noch erwähnt werden.

Um die Darstellung zu verbessern und den Nutzen für den Anwender weiter zu erhöhen, gibt es eine Vielzahl von Entwicklungsmöglichkeiten. Hier sollen vor allem einige dargestellt werden, die sich aus den Erfahrungen mit dem Prototypen ergeben haben und die auf Basis des momentanen Projektstands ohne grundlegende Veränderungen umgesetzt werden können.

8.7.1 Darstellung von Text

Die *Render-Bibliothek* (Unterabschnitt 11.5.2) stellt Text dar, indem dieser in ein 2D-Bild geschrieben und so als Textur auf dem Objekt angezeigt wird.

Andere Techniken, die eine höhere Darstellungsqualität erreichen, wie sie beispielsweise von *GEF3D* (Unterabschnitt 3.1.6) genutzt oder von [[RCL05](#)] vorgestellt werden, wurden ebenfalls in Betracht gezogen. Besonders die Möglichkeiten aktuellster Grafikhardware mit OpenGL4-Unterstützung, neue Geometrien direkt auf der Grafikeinheit per Tesselation-Shader zu erzeugen, könnten für die Implementierung von gut lesbaren und dennoch performanten Darstellungstechniken interessant sein.

Jedoch war die Schriftqualität des verwendeten texturbasierten Ansatzes ausreichend für den hier entwickelten Prototypen und lies sich einfach implementieren. Da die Darstellung von Schrift wichtig für Verständlichkeit und Nutzen der grafischen Repräsentation ist, sollte für weitere Arbeiten auf diesem Gebiet jedoch evaluiert werden, inwieweit sich die Qualität des hier genutzten Ansatzes verbessern lässt oder ob ein anderer Ansatz gewählt werden muss.

Bei ungünstigen Beobachtungssituationen, also bei großer Entfernung und schräger Betrachtung von Flächen, wird es im Prototypen schnell schwierig, Texte ohne Anstrengung zu lesen. Es müssen eher

³Damit sind der Autor dieser Arbeit und [[Hol12](#)] sowie [[Buc12](#)] gemeint.

große Schriften gewählt werden und daher lässt sich relativ wenig Information auf den Knoten darstellen. Außerdem muss der Kontrast zwischen Textfarbe und Hintergrund immer sehr hoch sein, um eine angemessene Lesbarkeit zu erreichen. Eine bessere Darstellungsqualität würde hier für mehr Flexibilität sorgen.

Eine sinnvolle Erweiterungsmöglichkeit wäre es, die Anzeige von Informationen bei weit entfernten Objekten automatisch zu vereinfachen⁴, indem beispielsweise ein Text abgekürzt und größer dargestellt wird. So wäre es möglich, Knoten mit größerem Abstand immerhin noch zu unterscheiden. Dafür kann ein zusätzliches Attribut im Prozessmodell definiert werden, dass eine Abkürzung für ein längeres Textattribut angibt.

8.7.2 Konfigurierbarkeit

Abgesehen von den im *Metamodell* (Abschnitt 7.2) konfigurierbaren Visualisierungsparametern fehlt es noch an weiteren Möglichkeiten, die grafische Darstellung zu beeinflussen.

So wäre es angebracht, eine Konfigurationsmöglichkeit für die *Beleuchtung* (Abschnitt 8.4) anzubieten. Wie in jenem Abschnitt gesagt ist es schwierig, Einstellungen zu finden, die für alle Situationen gut geeignet sind. Diese hängen auch von der verwendeten Anzeige und von Einflüssen wie Umgebungslicht oder der persönlichen Wahrnehmung des Benutzers ab.

In der grafischen Oberfläche sollte es hierzu eine Möglichkeit geben, Lichtquellen zu setzen und deren Parameter zu verändern. Es bietet sich an, auch sinnvolle Standardeinstellungen bzw. auswählbare Profile zur Verfügung zu stellen, um den Benutzer nicht mit zu vielen Aufgaben zu überfordern. Lichtquellen sind in Simulator X über zugehörige Licht-Entities erstell- und konfigurierbar, wie es auch von der *Renderkomponente* (Kapitel 10) unterstützt wird.

Ähnliches gilt für *2D-Modellierungsflächen* (Abschnitt 8.3). Sie sind momentan in der Implementierung fest vorgegeben, da es in der GUI noch keine Konfigurationsmöglichkeit gibt. Die Flächen können aber ebenfalls nach Bedarf erstellt und über zugehörige Entities konfiguriert werden.

Es wäre sinnvoll, die aktuellen Einstellungen für Lichtquellen und Modellierungsflächen auch in die Editor-Modelle aufzunehmen und damit persistent zu machen.

8.7.3 Räumliche Darstellung

Die räumliche Darstellung, vor allem der Tiefeneindruck ist für das Verständnis von 3D-Visualisierungen wichtig [WTS89] [WM08]. Modellierungsflächen und eine passende Beleuchtung können hilfreich sein, um dem Benutzer die räumliche Orientierung zu erleichtern, wie es der Prototyp zeigt.

Jedoch ist die Darstellung von 3D-Szenen auf einem PC-Bildschirm oder Projektor üblicherweise nur eine 2D-Projektion, bei der ein realistischer Tiefeneindruck fehlt. Dies macht es manchmal schwierig zu erkennen, welche Objekte näher am Betrachter liegen und welche sich im Hintergrund befinden.

Es besteht die Möglichkeit, sich an der Größe der Objekte zu orientieren. Jedoch kann dies auch scheitern, wenn Objekte unterschiedlich groß sein dürfen, wie es momentan der Fall ist. Die Skalierung von Modellelementen allerdings komplett zu verbieten würde eine zu starke Einschränkung bedeuten.

Andere Effekte, die aus der „Umwelt“ bekannt sind und die einen besseren räumlichen Eindruck ermöglichen können sind die Bewegungsparallaxe, Stereoskopie ([siehe](#) Unterabschnitt 3.3.1) und Schatten. Der Bewegungsparallaxen-Effekt lässt sich durch seitliche Bewegung des Benutzers in der Szene erzeugen und gibt einen Eindruck davon, wie weit Objekte von diesem entfernt sind.

⁴In der Computergrafik wird das Prinzip als „Level Of Detail“ bezeichnet.

Schatten

Ein Schattenwurf der Objekte kann verdeutlichen, wie weit Objekte von einer Fläche entfernt sind und wie der Betrachter zur Lichtquelle orientiert ist. Jedoch müsste getestet werden, inwieweit dies hilfreich ist und ob Schatten nicht zu häufig dazu führen, dass sich Informationen im Modell schlecht erkennen lassen. Konfigurationsmöglichkeiten oder eine „intelligente“ Schattenberechnung, die weniger auf realistische Effekte setzt aber dafür Lesbarkeitsaspekte berücksichtigt sind hier angebracht.

Voll immersive virtuelle Welten

Eine weitere Entwicklungsmöglichkeit wäre es, *voll immersive virtuelle Welten* (Unterabschnitt 3.3.2) zu nutzen. Dies ist auch ein Anwendungsgebiet, das von der hier verwendeten Plattform Simulator X unterstützt wird. Besonders Anzeigen mit hoher Auflösung würden Vorteile für Lesbarkeit und Verständlichkeit mit sich bringen ([siehe](#) Unterabschnitt 3.3.1).

Das Ziel des Projekts ist es aber eher auf technisch noch sehr aufwändige sowie teure Lösungen zu verzichten und ein System für die „breite Masse“ bereitzustellen. Durch die ständige technische Weiterentwicklung könnten solche Systeme aber in Zukunft durchaus eine praktische Alternative zu üblichen Benutzerschnittstellen für diverse Einsatzgebiete werden.

Verdeckung

Problematisch ist die in 3D-Visualisierungen auftretende Verdeckung von Informationen durch andere Modellelemente, wie schon bei dem von *Brown* (Unterabschnitt 3.2.3) vorgestellten 3D-Prozesseditor zu sehen war. Ist ein Element verdeckt, kann im Prototypen die Betrachterposition verändert werden, um die Sicht auf das Element freizugeben. Allgemein sollten Modelle aber so erstellt werden, dass aus „üblichen“ Betrachtungsrichtungen möglichst wenig Verdeckung auftritt, um sich nicht ständig hin- und herbewegen zu müssen. Eine andere Möglichkeit ist es, die verdeckenden Elemente transluzent zu machen, wie es im Prototypen durch das Deaktivieren von Elementen möglich ist.

Interessant wäre es auch, die Durchsichtigkeit von verdeckenden Elementen automatisch zu beeinflussen wie es unter dem Stichwort „dynamic transparency“ von [EAT09] vorgestellt wird. Objekte würden nach ihrer Wichtigkeit für die aktuelle Betrachtungssituation eingeteilt. Unwichtige Objekte, „distractors“ genannt, würden automatisch transluzent⁵ dargestellt falls sie wichtige Objekte („targets“) verdecken. So könnte durch den Benutzer beispielsweise festgelegt werden, dass aktuell „Datenknoten“ besonders wichtig sind und nicht verdeckt werden dürfen.

8.7.4 Darstellung von Kanten

Ein großes Problem bei 3D-Visualisierungen können schlecht erkennbar Verbindungen sein; vor allem die Richtung festzustellen, ist bei weit entfernten Kanten oft nur schwer möglich. Dies zeigte sich bei den Versuchen mit den Prototypen. Hier lässt sich feststellen, dass es keine Lösung gibt, die für jeden Anwendungsfall optimal geeignet ist.

Gerichtete Kanten werden in i>PM3D durch eine sich wiederholende „Pfeiltextur“ auf Verbindungen dargestellt ([siehe](#) Unterabschnitt 8.1.2). Das hat den Vorteil, dass die Richtung auch erkennbar ist, wenn die Verbindung zu großen Teilen durch andere Objekte verdeckt wird.

Der Ansatz, die Richtung durch eine dreidimensionale Pfeilspitze darzustellen, leidet unter dem Problem der Verdeckung. Eine solche Darstellung liegt jedoch näher an bekannten visuellen Sprachen und sollte daher unterstützt werden. Damit gäbe es auch mehr Möglichkeiten um den Typ von Verbindungen durch

⁵Es muss nicht das komplette Objekt durchsichtig sein; es reicht aus, wenn Teile eines Objekts transluzent sind, die auch wirklich für eine Verdeckung sorgen.

verschiedene Pfeilspitzen oder -enden besser zu unterscheiden. Bisher kann dies nur über die Farbe, Variation der Textur, und die Dicke dargestellt werden.

Die Darstellung als gerade Linie, wie sie momentan verwendet wird, kann dazu führen, dass Knoten verdeckt oder andere Elemente geschnitten werden. Das Problem sich kreuzender Verbindungen ist immerhin nicht so groß wie im 2D-Bereich, da die zusätzliche Dimension zur Vermeidung genutzt werden kann.

Verbindungen könnten alternativ auch gekrümmt oder aus mehreren Liniensegmenten aufgebaut gezeichnet werden, um solche Probleme weiter einzudämmen, wie es auch in 2D-Werkzeugen häufig zu sehen ist. Kanten, die als „gebogene 3D-Röhren“ dargestellt werden, zeigen [SA94] und [BD04] (Abbildung 8.12). Von [HW09] wird eine Benutzerstudie zur Effizienz von unterschiedlichen Darstellungsformen für gerichtete Kanten vorgestellt, deren Richtung beispielsweise auch durch Farbverläufe und andere Farbeffekte angezeigt werden könnten.

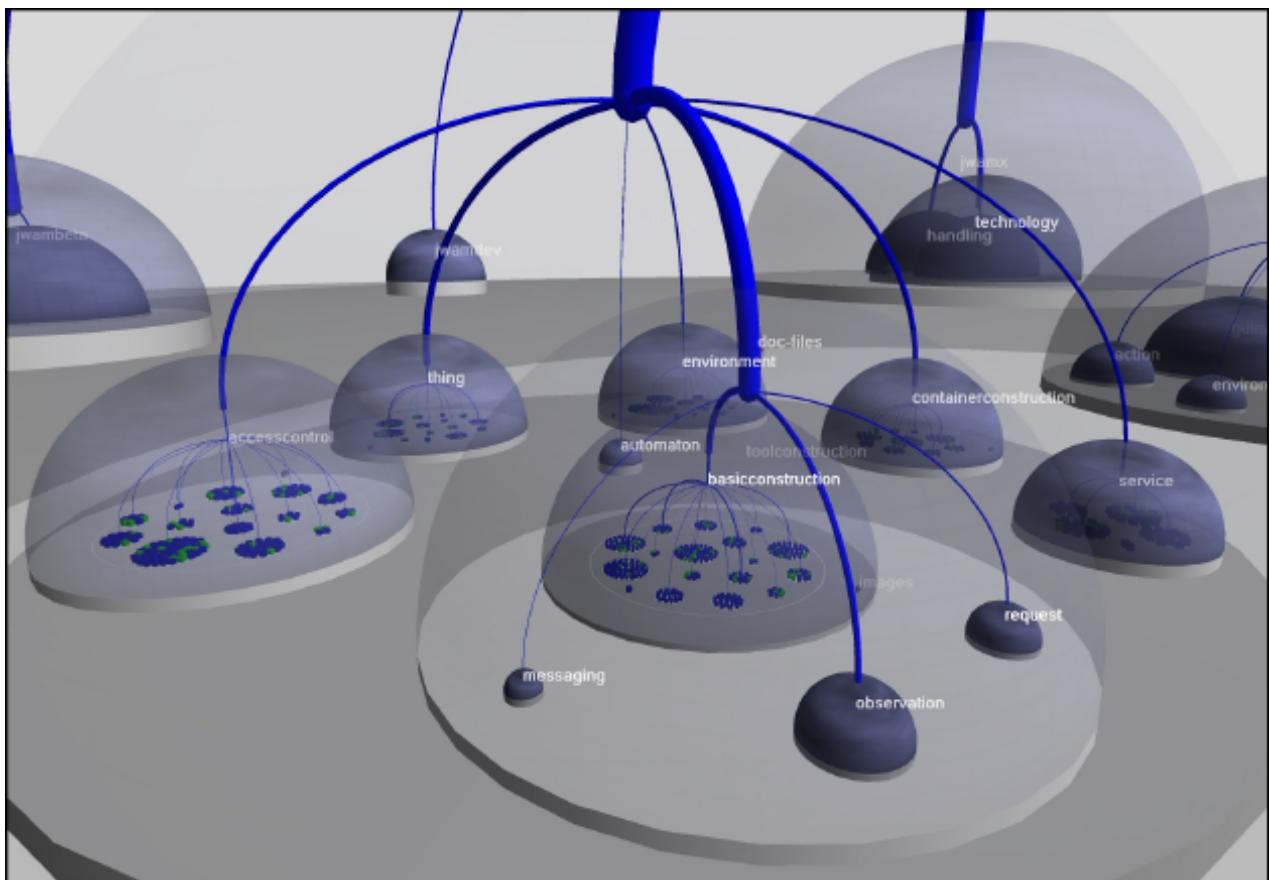


Abbildung 8.12: Visualisierung von Beziehungen mit gekrümmten 3D-Röhren aus [BD04]

Modellanbindung

Innerhalb des Prozessmodellierungswerkzeugs stellt die Modellanbindung (Abbildung 9.1) das Bindeglied zwischen einer Benutzerschnittstelle, wie sie von der Editorkomponente [Hol12] realisiert wird und dem zu manipulierenden Modell dar (*Anforderungen (e,f)* (Abschnitt 1.3)).

Genauer ergeben sich für die Modellanbindung folgende funktionale Anforderungen:

1. Erstellen und Löschen von Modellelementen
2. Bewegen, Rotieren und Skalieren von Modellelementen
3. Verändern von Prozessmodellattributen
4. Modifizieren der Visualisierungsparameter von Modellelementen; beispielsweise Farbe oder Schriftart
5. Erstellen, Laden und Speichern von Prozessmodellen und deren visueller Repräsentation

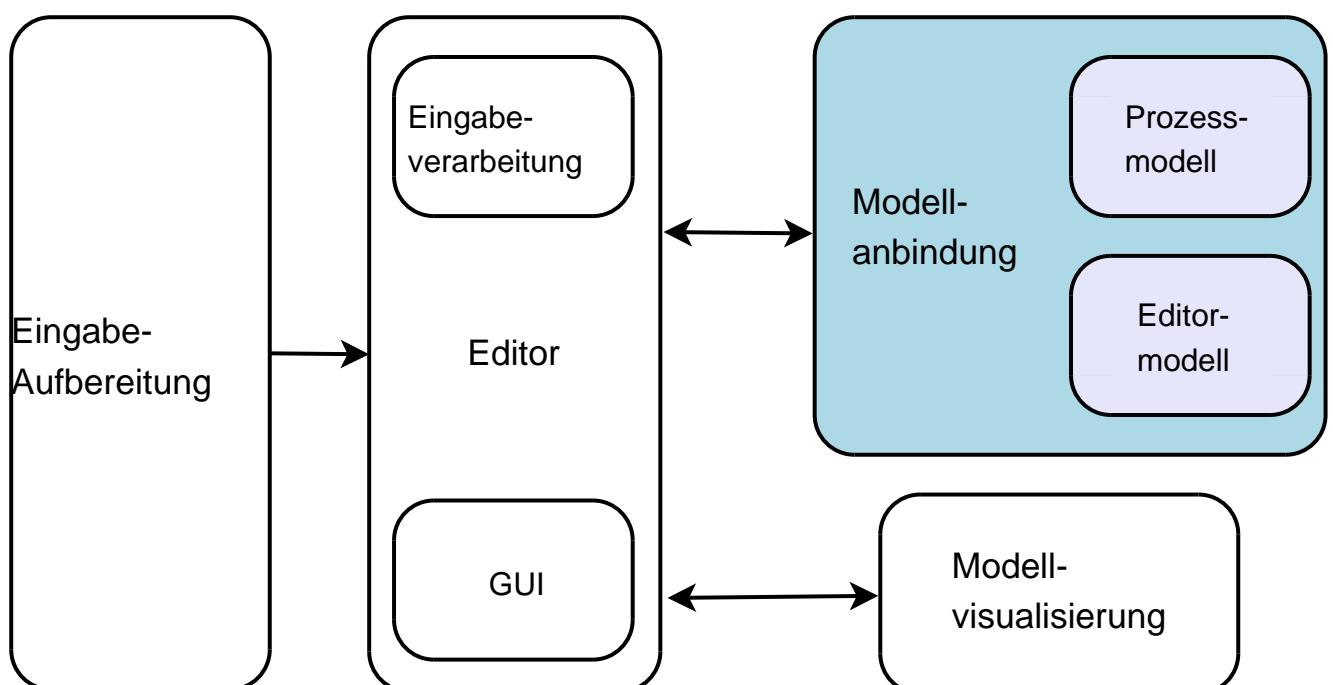


Abbildung 9.1: Die Modellanbindung im Kontext von i>PM3D

9.1 Modellkomponente

Wie in *Simulator X* (Abschnitt 4.2) erläutert, bestehen Simulator-X-Anwendungen aus einer Reihe von Komponenten, die jeweils ein wohl definiertes Aufgabengebiet abdecken. Die `ModelComponent` stellt folglich dem System alle Funktionalitäten zur Verfügung, die im Zusammenhang mit der Manipulation von Modellen stehen (*Anforderung (e)*).

So wird der Zugriff auf die Modelle vollständig von der `ModelComponent` gekapselt; es gibt für andere Systembestandteile keine Möglichkeit, direkt darauf zuzugreifen. Abgesehen von der erhöhten Übersichtlichkeit und Wartbarkeit wird dies auch durch die Actor-basierte Architektur und damit nachrichtenbasierte Kommunikation („message passing“) von Simulator X vorgegeben.

9.1.1 Commands

Die Modelfunktionen werden zum Teil als `Commands` bereitgestellt, die über das Kommunikationssystem von *Simulator X* (Abschnitt 4.2) an die `ModelComponent` geschickt werden. In der momentanen Umsetzung werden diese `Commands` ausschließlich durch die Editorkomponente genutzt, die von [Hol12] beschrieben wird.

Es existieren `Commands` für die folgenden Funktionen:

- Laden von Metamodellen
- Laden, Speichern, Erstellen, Schließen und Löschen von (Usage-)Modellen
- Erstellen und Löschen von Knoten und Szenenobjekten
- Erstellen einer Kante zwischen zwei Knoten

Alle anderen Manipulationsmöglichkeiten — das sind diejenigen, die nur Parameter einzelner Modellelemente betreffen — werden über `ModelEntities` bereitgestellt, welche weiter unten in diesem Kapitel (Abschnitt 9.3) erläutert werden.

9.1.2 Übersicht

Da die Funktionalität der `ModelComponent` relativ umfangreich ist, ist diese wiederum in die in Abbildung 9.2 gezeigten „Unterkomponenten“ aufgeteilt.

Das Trait `Component` wird von Simulator X bereitgestellt und muss von allen Komponenten implementiert werden. Dessen Methoden beziehen sich überwiegend auf den „*Lebenszyklus*“ (Abschnitt 9.4) von `Entities`. Die Implementierung dieser Methoden erfolgt durch das eingemischte Trait `ModelComponentEntityLifecycle`.

Von Trait `ModelComponentHandlers` werden die Funktionen bereitgestellt, die eingehende Nachrichten (vor allem `Commands`) von anderen Komponenten verarbeiten und diese gegebenenfalls beantworten. Solche Funktionen werden in Simulator X als „Handler“ bezeichnet.

Das Laden (mittels „*LMMLight-Parser*“) und Speichern („*ModelToText*“) sowie die Verwaltung der geladenen (Meta-)Modelle wird der `ModelComponent` durch das Object `ModelContext` bereitgestellt.

9.2 Modell-Persistenz

Das Prozessmodellierungswerkzeug muss in der Lage sein, neue Modelle zu erstellen, diese abzuspeichern und wieder zu laden (*Anforderung (f)*). Die Modelle werden in der Sprache *LMMLight* (Abschnitt 6.1) beschrieben, welche in Dateien in einer textuellen Darstellung abgelegt und daraus wieder geladen werden kann.

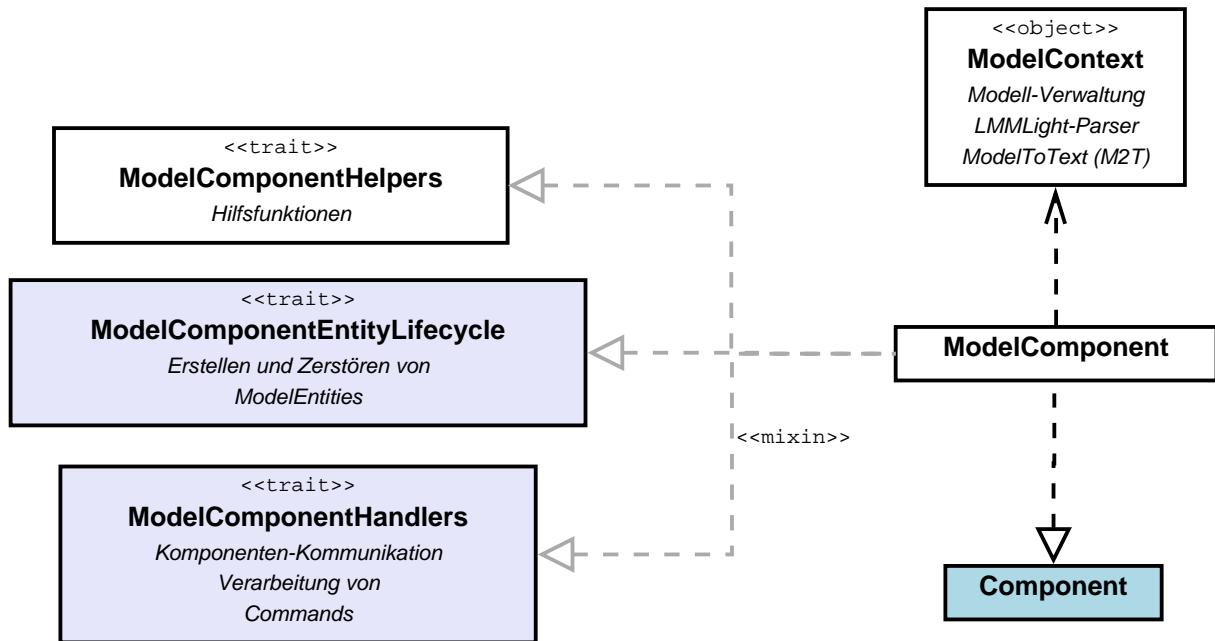


Abbildung 9.2: Unterkomponenten der ModelComponent (vereinfacht)

Für das Laden wird der im Rahmen dieser Arbeit entstandene **LMMLight-Parser** genutzt, der mit Hilfe der Scala-*Parser-Kombinatoren* (Unterabschnitt 4.1.5) implementiert wurde. Der Parser liefert einen Syntaxbaum der textuellen Eingabe, der aus „unveränderlichen“ (immutable) Objekten aufgebaut ist.

9.2.1 Speicherrepräsentation eines LMMLight-Modells

Um die Modelle in der Anwendung verändern zu können, wird der vom Parser gelieferte Syntaxbaum in eine andere Struktur überführt. Der so erzeugte Objektgraph ist an die an die EMF [[www:emf](#)]-Repräsentation zur Laufzeit angelehnt, wie sie in OMME von XText [[www:xtext](#)] erzeugt wird.

Vom Graphen wird der hierarchische Aufbau von LMM, wie in *Linguistic Meta Model* (Unterabschnitt 2.2.1) gezeigt abgebildet. Die Elemente von LMM werden durch analog benannte Klassen repräsentiert, die mit dem Buchstaben „M“ beginnen.

So wird die „Wurzel“ von einer MModel-Instanz gebildet, der sich MLevels unterordnen, die wiederum MPackages mit MConcepts sowie weiteren MPackages enthalten. Weiterhin kann ein MConcept andere MConcepts referenzieren. So ergibt sich ein azyklischer, gerichteter Graph. Ausgehend von einem MModel-Objekt kann die ModelComponent in einem Modell navigieren und es modifizieren, beispielsweise neue Concepts anlegen.

Abbildung 9.3 zeigt beispielhaft einen Ausschnitt aus der Speicherrepräsentation des *Domain-Model-Stacks* (Abschnitt 6.3). Im Beispiel ist das Concept ProcessUsage eine Verwendung von ProcessA. Mit ProcessUsage ist daher eine MConceptReference assoziiert, welche die Spezialisierungsrelation zwischen den beiden Concepts repräsentiert. ProcessUsage hat außerdem eine ausgehende Kante zu einer anderen (nicht gezeigten) Verwendung. Ausgedrückt wird dies durch die Zuweisung vom Typ MConceptAssignment, welche wiederum die zugehörige Kante ControlFlowA referenziert. Die Zuweisung gehört zu einem Attribut mit dem Namen „outboundControlFlows“, das im Concept Node aus dem *Prozess-Metamodell* (Abschnitt 7.4) definiert ist.

9.2.2 Vereinfachung des Umgangs mit Modellen

Um den Zugriff auf die Modelle zu vereinfachen und öfter vorkommende Aufgaben auszulagern, wurde eine Reihe von Adapters für die in der Speicherrepräsentation der Modelle genutzten Klassen imple-

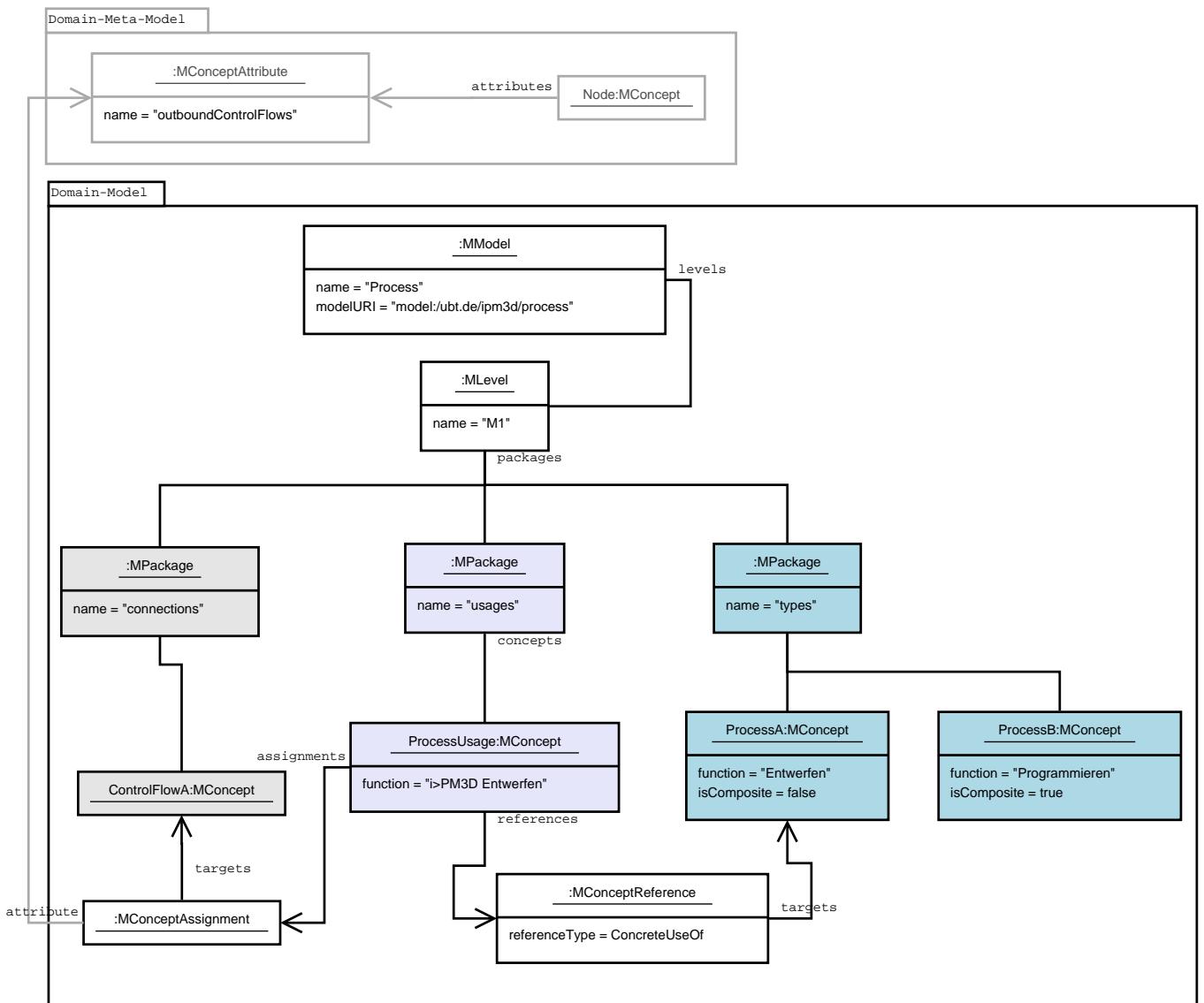


Abbildung 9.3: Speicherrepräsentation eines Beispiel-Prozessmodells (Ausschnitt)

mentiert. Ein Beispiel dafür ist der `MConceptAdapter`, dessen Methoden beispielsweise den schnellen Zugriff auf alle zuweisbaren Attribute (`assignableAttributes`), das Setzen von Werten (`setValue`) oder die Abfrage von Concept-Beziehungen (`instanceOf`) erlauben. Für alle Adapter werden *Implizite Methoden* (Unterabschnitt 4.1.4) angeboten, die die gekapselten Objekte direkt um die Methoden „erweitern“, die in den Adaptern definiert sind.

9.2.3 Laden von Metamodellen

Wie in *Modellhierarchie* (Kapitel 6) beschrieben wurde, werden Metamodelle für die Spezifikation der verwendeten Modellierungssprache und deren Repräsentation eingesetzt. Diese sollen prinzipiell austauschbar sein. Dazu wird von der Modellkomponente die Funktion bereitgestellt, Metamodelle zur Laufzeit zu laden.

Um das Laden der Modelle anzustoßen, ist folgendes Command definiert:

```
case class LoadMetaModels(domainModelPath: String, editorModelPath: String, loadAsResource: Boolean)
  extends Command
```

Von `loadAsResource` wird angegeben, ob die Pfade als Java-Resource-Path zu einer Metamodell-Datei interpretiert („true“) oder direkt im Dateisystem gesucht werden sollen („false“).

Es wird zur Vereinfachung der Implementierung davon ausgegangen, dass die Metamodelle der Domäne und des Editors immer paarweise geladen werden. Mehrere Repräsentationen zu einer Domäne zu laden ist somit noch nicht möglich. Die Modellkomponente lässt prinzipiell das Laden von mehreren Metamodell-Paaren zu. Jedoch wird dies von der Editorkomponente [Hol12] noch nicht unterstützt.

Nachdem die Metamodelle geladen worden sind, werden von der Modellkomponente Informationen aus den Modellen ausgelesen, die für die Editorkomponente relevant sind. Zum Einen ist dies eine Auflistung der verfügbaren Kanten- und Szenenobjekttypen, die vom Benutzer erzeugt werden können und die der Editor zu diesem Zweck in seiner Palette anzeigt. Zum Anderen wird der Editor über die Knoten-„Metatypen“ informiert, von denen nach dem Typ-Verwendungs-Konzept zur Laufzeit Typen vom Benutzer angelegt werden.

Die Kommunikation zwischen Editor- und Modellkomponente wird in Abbildung 9.4 am Laden von Metamodellen beispielhaft gezeigt. Nachrichten, die mit Großbuchstaben beginnen stellen Commands beziehungsweise Replies dar; Nachrichten mit Kleinbuchstaben sind gewöhnliche Methodenaufrufe und -rückgabewerte.

9.2.4 Laden und Schließen von und Umgang mit mehreren Modellen

Ein konkretes „Prozesmodell“ wird geöffnet, indem das zugehörige *Domain-Model* und *Editor-Usage-Model* geladen werden. Zusammen werden diese beiden Modelle im Folgenden vereinfachend das *Usage-Modell* genannt, welches den aktuellen Zustand eines Prozessmodells und dessen Repräsentation im Editor umfasst¹.

Das Command `LoadUsageModels` ist analog zum Command `LoadMetaModels` definiert, wie im vorherigen Unterabschnitt beschrieben.

Es können von der Anwendung zur Laufzeit mehrere Usage-Modelle (zu denselben Metamodellen) geladen werden. In der `ModelComponent` ist jeweils ein Usage-Model-Paar als „aktiv“ gekennzeichnet. Commands wie das Erstellen von Knoten beziehen sich immer auf das aktive Usage-Model. Welches Modell „aktiv“ ist kann über das Command `SetActiveUsageModel` geändert werden.

¹Die Benennung „Usage-Model“ ist eigentlich nicht ganz passend, da das *Domain-Model* (Unterabschnitt 6.3.2) auch die vom Benutzer erstellten Knotentypen umfasst. Da diese Bezeichnungsweise in der Implementierung zu finden ist, wird diese hier ebenfalls genutzt.

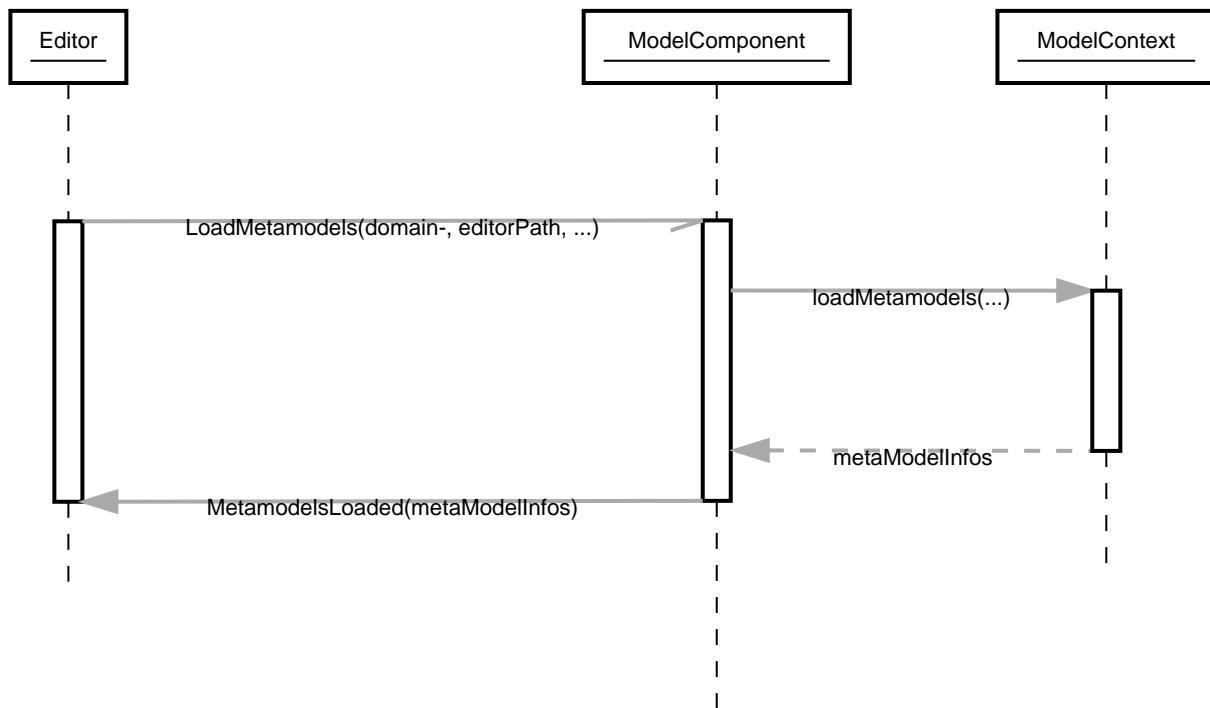


Abbildung 9.4: Sequenzdiagramm LoadMetaModels (vereinfacht).

Modelle können über `CloseUsageModel` wieder geschlossen werden, wobei alle seit dem letzten Speichern erfolgten Änderungen verloren gehen.

Der Umgang mit mehreren Modellen wird auch von der Editorkomponente unterstützt.

Nachdem ein Usage-Model geladen wurde, wird der Aufrufer analog zum Laden der Metamodelle über die im *Domain-Model* definierten Knotentypen informiert.

9.2.5 Speichern von Modellen

„Speichern“ bedeutet hier, dass die Änderungen an Modellelementen in das Usage-Model zurückgeschrieben werden und das dieses anschließend in textueller Form persistiert wird. Analog zu `LoadUsageModels` werden bei `SaveUsageModels` zwei Dateinamen für Domain- und Editormodell angegeben. Java-Resource-Pfade sind hier nicht erlaubt.

Um die Speicherrepräsentation des Modells wieder in eine durch den LMMLight-Parser lesbare², textuelle Darstellung zu überführen, wird der in [StringTemplate](#) (Unterabschnitt 4.4.1) vorgestellte Wrapper für die StringTemplate-Bibliothek genutzt.

9.3 Modell-Entitäten

Objekte, mit denen verschiedene Teile des Systems interagieren, werden in [Simulator X](#) (Abschnitt 4.2) durch Entities beschrieben. Es ist daher zweckmäßig, für jedes Modellelement sowie für Szenenobjekte eine zugehörige Entity zu erstellen. ModelEntities werden von der ModelComponent erzeugt, wenn über ein Command die Erstellung von neuen Elementen angefordert oder ein Modell geladen wird.

Wie aus [Simulator X](#) (Abschnitt 4.2) bekannt werden Entities mit Hilfe von EntityDescriptions beschrieben, die aus Aspects aufgebaut sind. In Abbildung 9.5 ist eine solche Entity-Definition zu sehen. Der Ablauf bei der Erstellung einer ModelEntity aus einer EntityDescription wird im Abschnitt [Übersicht über den Lebenszyklus von Model-Entitäten](#) (Abschnitt 9.4) vorgestellt.

²Die Darstellung ist aber auch durchaus „menschlesbar“ und wird ähnlich formatiert wie im Metamodell-Editor von OMME.

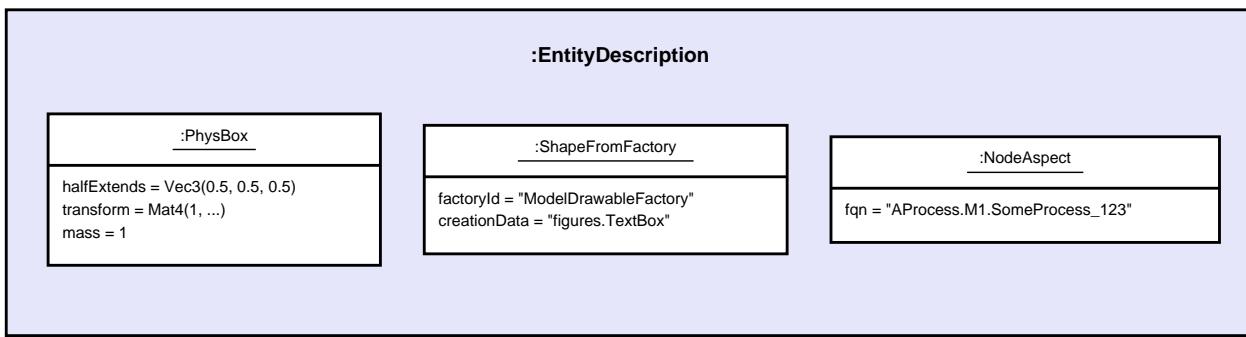


Abbildung 9.5: EntityDescription für einen Knoten (nur ausgewählte und vereinfachte Attribute)

9.3.1 Aspekte

Die zur Erstellung von ModelEntities genutzten Aspects werden im Folgenden beschrieben.

Physik

Knoten und Szenenobjekte sollen in die physikalische Simulation eingebunden werden, um Kollisionen zu erkennen und eine Auswahl der Elemente zu ermöglichen [Hol12] [Buc12]. Hierfür stellt die Physikkomponente verschiedene Aspects bereit, die besagen, dass eine bestimmte physikalische Repräsentation zu einer Entity erzeugt werden soll. Da bisher nur annähernd quaderförmige Geometrien für die Visualisierung von Knoten genutzt werden, wird hier für alle Knoten der PhysBox-Aspect (Abbildung 9.5) verwendet.

Kanten definieren keinen Physik-Aspect und besitzen daher keine physikalische Repräsentation³.

Grafik

Die *Renderkomponente* (Kapitel 10) stellt verschiedene RenderAspects bereit, die der Renderkomponente alle nötigen Informationen mitteilen, um ein Visualisierungsobjekt zur entsprechenden Entity anzulegen.

Szenenobjekte, welche aus COLLADA-3D-Modelldateien geladen werden, werden von der Renderkomponente selbst erzeugt. Solche Szenenobjekte sind statisch durch das 3D-Modell definiert. Das bedeutet in diesem Zusammenhang, dass ihr Erscheinungsbild zur Laufzeit nicht geändert werden kann (abgesehen von Position, Rotation und Skalierung). In der Entity-Beschreibung wird dafür der ShapeFromFile-Aspect angegeben.

Für Knoten und Kanten wird dagegen der ShapeFromFactory-Aspect genutzt, der besagt, dass sich die *Renderkomponente* (Kapitel 10) das Grafikobjekt von einer externen Factory erzeugen lassen soll. In Abbildung 9.5 ist zu sehen, dass im Aspect die ModelDrawableFactory angegeben wird, welche alle Grafikobjekte für Knoten und Kanten erzeugt. Parameter creationData gibt den Typ des gewünschten Objekts an, der in den Figuren im *Editor-Metamodell* (Unterabschnitt 7.2.2) spezifiziert wurde. Die ModelDrawableFactory wird später in einem *Anwendungsbeispiel* (Abschnitt 11.6) modifiziert, um ein Grafikobjekt für einen neuen Knotentyp hinzuzufügen.

Modell

Für die drei Elementtypen Knoten, Kanten und Szenenobjekte gibt es jeweils einen Aspect, der von ModelAspect abgeleitet ist, wie beispielsweise den NodeAspect, wie er in Abbildung 9.5 zu sehen ist.

³Dies ist nicht nötig, da die Auswahl von Kanten nicht unterstützt werden soll und Kollisionen mit Verbindungen eher als hinderlich gesehen wurden. Außerdem könnte eine große Anzahl von Verbindungen schnell zu Geschwindigkeitsproblemen der Physiksimulation führen.

ModelAspects sind der ModelComponent zugeordnet und enthalten für Nutzer der ModelEntity relevante Informationen.

Für alle Elemente, die von ModelEntities repräsentiert werden wird ein voll qualifizierter Name (fqn) vergeben, der das Element eindeutig innerhalb des Systems identifiziert. Dieser Name wird in Commands verwendet, die sich auf bestimmte Elemente beziehen, wie beispielsweise das Verbinden oder Löschen von Knoten. Bei Knoten und Kanten wird dafür die FQN des entsprechenden Modellelementes aus dem *Domain-Model* genutzt. Szenenobjekte werden über die FQN des *Editor-Usage-Concepts* identifiziert⁴.

Außerdem wird ein Identifikationsstring (creatorId) mitgeliefert, der vom Ersteller eines Elements definiert wird. Mit „Ersteller“ ist hier der Absender des entsprechenden Commands oder die ModelComponent selbst gemeint. Diese ID kann von diesem dafür benutzt werden, neu erstellte Entities in internen Datenstrukturen richtig zuzuordnen.

9.3.2 Setzen und Auslesen von Position, Ausrichtung und Größe

(Dieser Unterabschnitt beschreibt von Simulator X vorgegebene Funktionalität. Projektspezifische Anpassungen sind mit Fußnoten versehen.)

Position und Ausrichtung sind – wie in der Computergrafik üblich [AHH08] – zu einer Transformations-Matrix zusammengefasst. Die Skalierung eines Objekts wird durch einen Vektor (mit drei Komponenten) angegeben⁵. Beide Werte werden für Knoten und Szenenobjekte von der Physikkomponente verwaltet.

Sie können von anderen Komponenten verändert werden, indem eine Nachricht an die Physikkomponente geschickt wird:

```
physics ! SetTransformation(newTransformationMatrix)  
physics ! SetScale(newScaleVector)
```

Von der Physikkomponente werden außerdem zwei SVars zur Entity hinzugefügt (Typen ScaleVec und Transformation), die allerdings nur lesend genutzt werden dürfen.

Beispielsweise kann so die aktuelle Transformation ausgegeben werden⁶:

```
processEntity.svarGet(Transformation) {  
    value => println("current transformation of processEntity: " + value)  
}
```

Dabei ist zu beachten, dass der Get-Aufruf „nicht-blockierend“ erfolgt. Wie in *Simulator X* (Abschnitt 4.2) beschrieben wurde, muss der Wert einer SVar eventuell von einem anderen Actor über das Kommunikationssystem angefragt werden. Die anonyme Funktion (im Code-Beispiel in geschweiften Klammern) wird ausgeführt, sobald die Antwort vorliegt.

Für Objekte ohne Physik-Aspekt (Kanten) werden die genannten SVars durch die Renderkomponente bereitgestellt. Diese leisten dasselbe, dürfen aber auch verändert werden:

```
processEntity.svarSet(Transformation)(newTransformationMatrix)
```

9.3.3 Modell-SVars

Weitere Parameter der Modellobjekte lassen sich ebenfalls über SVars auslesen und setzen.

⁴Dass hier die FQNs aus dem Modell genutzt werden hat keine besondere Bedeutung und ist nur ein „Implementierungsdetail“, auf das man sich nicht verlassen sollte.

⁵Skalierung wurde für das Projekt hinzugefügt. Dazu wurde die Physikkomponente modifiziert und die selbstgeschriebene Renderkomponente entsprechend ausgelegt.

⁶Die svarGet-Methode (ebenfalls svarSet und weitere) wurde für das Projekt in einem impliziten Wrapper für Entities definiert um den Zugriff auf SVars zu „verschönern“.

Diese SVars lassen sich in drei Gruppen einteilen. SVars können direkt Attribute aus den beiden zugrunde liegenden (Meta)-Modellen abbilden oder von der Modellkomponente definiert sein:

1. **Domain-Model-SVars:** Solche SVars werden zu Attributen erzeugt, die im *Domain-Meta-Model* definiert sind und denen in Concepts im *Domain-Model* Werte zugewiesen werden können. Sie stellen somit die Schnittstelle dar, über die Modellattribute wie die Funktion eines Prozesses oder der Name eines Konnektors verändert werden können. Unterstützt werden alle literalen Datentypen; den SVars werden die passenden Scala-Datentypen zugewiesen.
2. **Editor-Model-SVars:** Diese SVars werden nach Bedarf aus den Attributen des *Editor-Metamodels* erstellt. Sie erlauben es, die Visualisierung der Elemente anzupassen, wie sie im Editormodell beschrieben wird⁷. Neben literalen Attributen werden hier auch Attribute unterstützt, die Concepts referenzieren. Letztere werden für die meisten hier genannten SVars benötigt.

Welche Editor-Attribute unterstützt werden wird von der `ModelComponent` festgelegt; dies sind⁸:

- Hintergrundfarbe (`backgroundColor`)
- Schrift (`font`)
- Schriftfarbe (`fontColor`)
- Texturpfad (`texture`)
- Liniendicke (`thickness`)
- Spekulare Farbe (`specularColor`)

3. **Editor-SVars:** Dies sind SVars, die keine direkte Entsprechung im Modell haben und deren Werte daher auch nicht persistiert werden. Sie sind automatisch für alle Modellelemente definiert oder werden durch Modellattribute „aktiviert“.

- SVars für die Auswahl von *Visualisierungsvarianten* (Abschnitt 8.2):
 - Deaktivierung (`disabled`),
 - Hervorhebung (`highlighted`)
 - Selektion (`selected`)
- Parameter für die Visualisierungsvarianten
 - Breite des Selektionsrahmens (`borderWidth`)
 - Hevorhebungsfaktor (`highlightFactor`)
 - Transluzenzfaktor bei deaktivierten Elementen (`deactivatedAlpha`)

Alle hier genannten SVars werden von der Modellkomponente aktiviert, wenn im Modell das Attribut `interactionAllowed` auf „true“ gesetzt ist.

Alle SVars müssen eindeutig durch eine `SVarDescription` beschrieben werden, der ein Symbol zur Identifizierung und einen Scala-Datentyp umfasst. Die Symbole für Editor-SVars beginnen mit `editor`; Symbole für *Domain-Model-SVars* werden mit `model` gekennzeichnet. Daran wird der Attributname aus dem Modell oder im Falle der *Editor-SVars* einer der unter 3. genannten Bezeichner angehängt, abgetrennt durch einen Punkt.

⁷Es wäre auch erlaubt, Attribute zu integrieren, die nicht direkt die Visualisierung betreffen, aber das Editor-Verhalten modifizieren. Dies wird bisher aber nicht genutzt.

⁸Es war nicht möglich, die Implementierung (auf einfacherem Wege) so flexibel zu gestalten wie bei Domain-Model-SVars, was leider dazu führt, dass man keine Attribute hinzufügen kann ohne die `ModelComponent` anzupassen.

Anwendungsbeispiel

Die Nutzung erfolgt analog zu den schon gezeigten *SVars* (Unterabschnitt 9.3.2), wozu ebenfalls ein impliziter Wrapper definiert ist. Im folgenden Beispiel wird die Funktion eines Prozessknotens und die Schriftfarbe über die zugehörige Entity verändert:

```
processEntity.svarSet("model.function")("Ausarbeitung schreiben")
processEntity.svarSet("editor.textColor")("Color.BLACK")
```

9.4 Übersicht über den Lebenszyklus von Model-Entitäten

Dieser Abschnitt zeigt kurz, welche wichtigen Schritte im „Lebenszyklus“ einer `ModelEntity` durchlaufen werden.

Komponenten in *Simulator X* (Abschnitt 4.2) definieren eine Reihe von Methoden, die vom Framework beim Erstellen oder Löschen einer Entity aufgerufen werden.

Die Erstellung einer `ModelEntity` folgt dem folgenden Schema:

1. Der Vorgang wird beispielsweise durch ein `CreateNode`-Command vom Editor angestoßen. Die Modellkomponente erzeugt daraufhin eine `EntityDescription` mit den *Aspekten* (Unterabschnitt 9.3.1) und übergibt diese an das Framework (Methode `EntityDescription.realize`), welches die Erstellung der Entity verwaltet und die folgenden Methoden aufruft.
2. Die Methode `getAdditionalProvidings` gibt eine Sequenz von `SVarDescriptions` zurück, die zu der Entity hinzugefügt werden sollen. Im Falle der Modellkomponente sind dies `SVarDescriptions` zu den im vorherigen Abschnitt beschriebenen *SVars*.
3. Anschließend wird die Methode `getInitialValues` aufgerufen, welche Initialwerte für die definierten *SVars* zurückgeben soll. Die Modellkomponente liest hierzu die Attributzuweisungen aus den Modell-Concepts aus oder setzt Standardwerte.
4. Nach Fertigstellung einer Entity wird `newEntityConfigComplete` aufgerufen. Die Modellkomponente fügt die Entity zu ihrer internen Repräsentation hinzu und verbindet die Domain-Model-*SVars* mit den Attributen im Modell. Dies heißt, dass auf der *SVar* eine „Observe“-Funktion registriert wird, die bei jeder Änderung des *SVar*-Wertes auch den Wert im dahinterliegenden Domain-Concept ändert.⁹
5. Zum Abschluss werden Observer benachrichtigt, die auf die Erstellung von neuen Entities hören. Dies ist hier konkret die Editorkomponente, die auf diesem Weg die Entity zu ihrer internen Repräsentation hinzufügen kann.

Der genannte Ablauf spielt sich auch parallel für die anderen Komponenten ab, für die Aspects in der Entity definiert sind; hier also für die Render- und gegebenenfalls die Physikkomponente.

Beim Löschen spielt sich Folgendes ab:

1. Das Löschen wird beispielsweise durch ein `DeleteNode(fqnToDelete)`-Command vom Editor initiiert. Daraufhin startet die Modellkomponente den Löschvorgang, indem auf der zur FQN gehörigen Entity die Methode `remove` aufgerufen wird.
2. Simulator X entfernt nun die Entity aus dem System und ruft dabei in der Komponente die `removeFromLocalRep`-Methode auf. In dieser Methode sollen interne Verweise und zugehörige Daten in den Komponenten entfernt werden.

⁹Bei den Editor-Model-*SVars* wird ein anderer Ansatz genutzt, da diese teilweise häufig geändert werden (vor allem die Position). Diese *SVars* werden erst beim Speichern des Modells ausgelesen und zurückgeschrieben, um Probleme mit der Ausführungsgeschwindigkeit zu vermeiden.

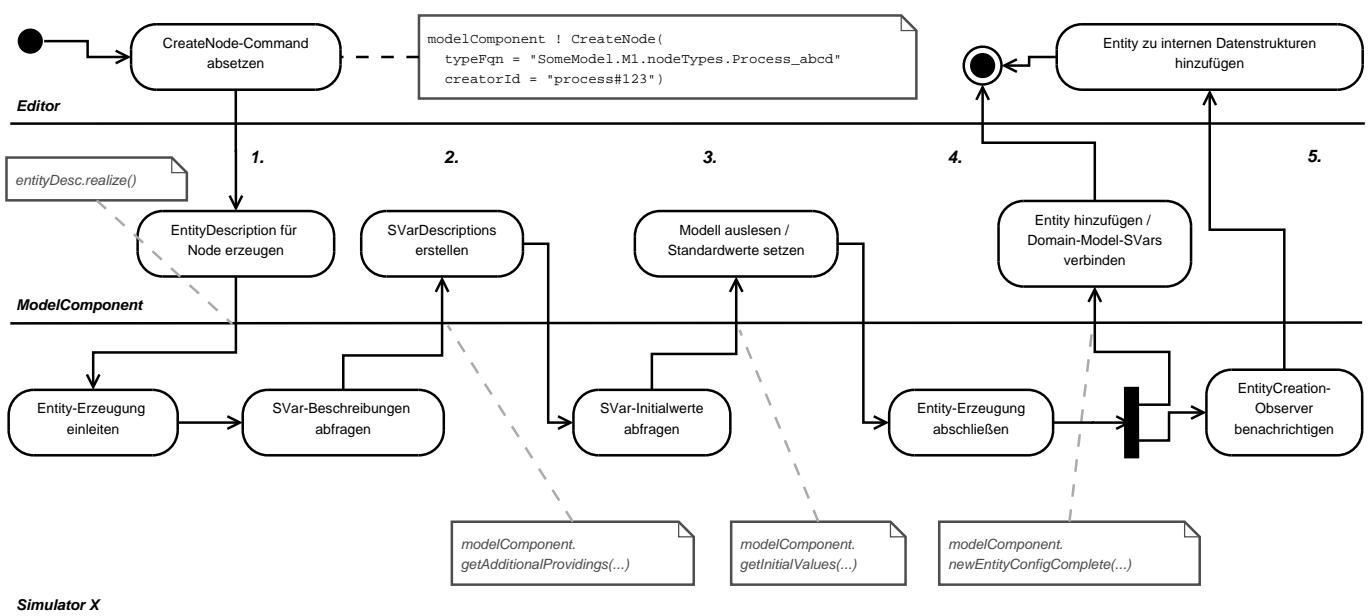


Abbildung 9.6: Ablauf der Erstellung einer ModelEntity durch die Editorkomponente

Renderkomponente

In diesem Kapitel wird die Renderkomponente vorgestellt, die die Grafikfunktionen¹ für i>PM3D bereitstellt. Von dieser Komponente wird die im Rahmen dieser Arbeit entstandene *Render-Bibliothek* (Kapitel 11) an Simulator X angebunden. Dadurch wird die von Simulator X bereitgestellte Komponente für die grafische Darstellung ersetzt, deren Fähigkeiten nicht ausreichten, um die hier vorgestellte Visualisierung auf einfacherem Wege zu implementieren.

Die eigentlichen Render-Aufgaben werden an einen Actor (Klasse `MMPERenderActor`) delegiert, der von der Renderkomponente (`MMPERConnector`) gestartet wird². Nachrichten, die Grafikfunktionen betreffen werden von anderen Komponenten an die Renderkomponente geschickt und an den RenderActor weitergeleitet.

10.1 RenderActor

Für den RenderActor musste ein eigener Scheduler für Scala-Actors erstellt werden, da der Standard-Scheduler die Actor-Arbeitspakete in beliebig wechselnden Threads ausführen kann. Dies ist für die Ausführung von OpenGL-Funktionen problematisch. Deswegen wird die Ausführung der Render-Aufgaben auf einen Thread beschränkt.

Der RenderActor definiert zwei Render-Ebenen. Die zuerst gezeichnete Ebene umfasst alle 3D-Objekte wie den Prozessgraphen und Szenenobjekte. Darüber wird eine Ebene gezeichnet, die 2D-Elemente wie Cursor der Eingabegeräte oder das Eightpen-Menü [Buc12] enthält. Beide Ebenen werden durch jeweils eine *RenderStage* (Unterabschnitt 11.3.2) gezeichnet.

Es gibt die Möglichkeit, den RenderActor durch Plugins zu erweitern, die an verschiedenen vordefinierten Erweiterungspunkten im Render-Prozess eingreifen können. Plugins können neue Grafikobjekte erzeugen und zu einer der beiden RenderStages hinzufügen oder selbst OpenGL-Funktionen ausführen. Dies wird im Projekt für die Darstellung der Nifty-GUI-Menüs [Hol12] und des Eightpen-Menüs [Buc12] genutzt.

10.2 OpenGL-Versionsproblematik

Die Library Nifty-GUI bringt eine eigene Render-Implementierung auf Basis von OpenGL 1.x mit, welche auch Funktionen nutzt, die in OpenGL 3.3 als „veraltet“ („deprecated“) eingestuft sind. Von der Render-

¹Die Implementierung umfasst auch die Übersetzung von Tastatur- und Mausdaten, die von LWJGL geliefert werden, in Simulator X - Events. Für diese Arbeit sind aber nur die Grafikfunktionen relevant.

²Dieser Aufbau ergibt sich aus der Idee, für die Darstellung der Szene mehrere Bildschirme nutzen zu können, wie es unter Anderem für ein CAVE-System nötig wäre. Dazu könnten der Renderkomponente mehrere RenderActors zugeordnet werden. Dies war vorgesehen, wird jedoch nicht überall in der Implementierung umgesetzt und daher nicht unterstützt.

Bibliothek werden dagegen nur Funktionen genutzt, die in OpenGL 3.3 verfügbar sind.

Wegen Nifty-GUI muss die Renderkomponente OpenGL 3.3 im Kompatibilitätsmodus betreiben, der auch die „deprecated“-Funktionen unterstützt. Es ist möglich, dass dies auf manchen Hardwareplattformen zu Geschwindigkeits- oder Darstellungsproblemen führt.

10.3 Projektspezifische Erweiterungen

Die Renderkomponente unterstützt die von Simulator X bereitgestellten RenderAspects für die Definition von Lichtquellen-Entities (`PointLight`) sowie den Aspect `ShapeFromFile`, der die Renderkomponente anweist, die grafische Repräsentation einer Entity aus einer COLLADA-Modelldatei zu laden.

Im Folgenden werden die Funktionalitäten aufgelistet, welche von der ursprünglichen Renderkomponente nicht unterstützt werden.

10.3.1 ShapeFromFactory

Mit der hier entwickelten Renderkomponente ist es möglich, die grafische Repräsentation einer Entity von einer Factory-Klasse oder -Actor erzeugen zu lassen. Damit lassen sich in der Anwendung beliebige Grafikobjekte nutzen, die mit Hilfe der *Render-Bibliothek* (Kapitel 11) erstellt wurden.

Hierfür ist der RenderAspect `ShapeFromFactory` definiert. Dies wird im Projekt für die Erstellung der Grafikobjekte für Modellelemente – also der Knoten und Kanten des Prozessmodells – genutzt.

10.3.2 Modellierungsflächen

Für die Darstellung von *2D-Modellierungsflächen* (Abschnitt 8.3) wurde ein `PlaneAspect` bereitgestellt, der SVars für die Konfiguration von Transluzenz, Farbe, Linienbreite und -dichte und das Deaktivieren der Linien definiert.

10.3.3 2D-Grafikobjekte

Der Aspect `Image2D` wird dafür genutzt, bewegliche und skalierbare 2D-Grafikobjekte darzustellen. Bei der Erstellung muss ein Pfad zu einer Bilddatei angegeben werden. Position und Größe sind über SVars modifizierbar. Diese Werte können entweder über die Bildschirm-Pixelkoordinaten oder über normalisierte Koordinaten (von -1 bis +1 in x und y-Richtung) angegeben werden. Welches Koordinatensystem genutzt wird, wird über den Konstruktor in `coordsAreNormalized` angegeben.

10.3.4 User-Entity

Simulator X stellt eine User-Entity bereit, über deren SVars `HeadTransform` und `ViewPlatform` die Position des Benutzers in der 3D-Szene bestimmt wird. Diese wird von der Renderkomponente erzeugt, die zusätzliche SVars definiert, über die `Viewport`³ - (Aspect `ViewportSettings`) und Render-Frustum-Einstellungen⁴ (Aspect `FrustumSettings`) abgefragt werden können⁵.

³Größe und Nullpunkt der Zeichenfläche für OpenGL, angegeben in Pixel.

⁴Diese Einstellungen legen die perspektivische Projektion fest. [[www:frustum](#)]

⁵Die Werte lassen im Prinzip sich auch verändern, nur wird dies von der Implementierung noch nicht vollständig unterstützt.

Render-Bibliothek

Für die im vorherigen Kapitel beschriebene Renderkomponente wurde eine Anbindung an eine Grafikchnittstelle benötigt, mit der sich die folgenden Anforderungen realisieren lassen:

1. Nutzung der Fähigkeiten moderner Grafikhardware, unter Anderem um eine hohe Ausführungs geschwindigkeit zu erlauben.
2. Möglichkeit, spezielle Effekte mit Hilfe direkter Programmierung der Grafikhardware flexibel implementieren zu können (Shaderprogrammierung).
3. Die Grafikausgabe sollte auf verbreiteten Plattformen funktionieren, mindestens auf Windows und Linux.
4. Bereitstellung von in der Computergrafik üblichen Abstraktionen wie einer Kamera, Lichtquellen und Grafikobjekten, die die Manipulation von Visualisierungsparametern zur Laufzeit erlauben.
5. Unterstützung von transluzenten Objekten
6. Darstellung von Schrift und Texturen auf 3D-Objekten, möglichst skalierbar
7. „Bild-in-Bild“-Techniken („off-screen rendering“), also das Zeichnen von 3D-Teil-Szenen in 2D- Bilder, die im Grafikspeicher abgelegt sind und so in eine größere 3D-Szene eingebettet werden können.

Anforderung (5) wird im Projekt für die Darstellung von **deaktivierte Modellelementen** (Unterabschnitt 8.2.3) sowie **Modellierungsflächen** (Abschnitt 8.3) gebraucht. Anforderung (7) ist auf den Wunsch zurückzuführen, komplexe Menüs wie das Eightpen-Menü [Buc12] erstellen zu können, die selbst wieder 3D-Objekte enthalten.

Die Anforderungen (1, 2, 3) werden – auf niedriger Ebene – nur von OpenGL in einer relativ aktuellen Version ab 3.0 erfüllt. Da „modernes“ **OpenGL** (Abschnitt 4.3) im Prinzip nur eine direkte Schnittstelle zur Grafikhardware ist, ist die Programmierung wegen fehlender Abstraktion und Wiederverwendbarkeit relativ aufwändig.

Im Gegensatz dazu stehen zahlreiche Frameworks, die verschiedene Aufgabengebiete und Abstraktionsebenen abdecken und auf Low-Level-Schnittstellen wie OpenGL aufsetzen. Diese könnten die Anforderungen (4–7) selbst abdecken oder zumindest dafür benutzt werden, diese zu implementieren. Solche Systeme werden oft als „Game-Engines“ bezeichnet, die in erster Linie für die Erstellung von 3D-Computerspielen gedacht sind, sich aber auch durchaus für andere Visualisierungsaufgaben einsetzen lassen¹.

¹Gezeigt wird dies von [AG09]; siehe Unterabschnitt 3.1.3. Dort sei die C++-Game-Engine Panda3D (über eine Python-Anbindung) genutzt worden, um schnell einen Prototypen für einen 3D-Zustandsdiagramm-Editor zu erstellen.

Für die Java-Plattform sind mit JME3 und Ardor3D zwei relativ aktuelle, in Java implementierte 3D-Frameworks verfügbar. JME3 lag zu Beginn der vorliegenden Projektes nur in einer Alpha-Version vor und schied deshalb für diese Arbeit aus. Allgemein ist es bei solchen eher komplexen Frameworks schwer einzuschätzen, ob alle benötigten Grafikeffekte ohne viel Aufwand oder sogar nur mit Änderungen an den Frameworks selbst realisiert werden können. Insbesondere war dies aufgrund der relativ spärlichen (Wiki-) Dokumentation von Ardor3D der Fall.

Diese Frameworks bieten außerdem eher „zu viele“ Fähigkeiten an, die über die reine Grafikausgabe hinausgehen, wie beispielsweise eine Physik-Unterstützung oder die Einbindung von Eingabegeräten. Oft werden verschiedenste Aufgaben wie die Grafikrepräsentation und die räumliche Strukturierung der Szene in einen zentralen „Szenengraphen“ integriert, mit dem eine Anwendung interagiert. Dies ist für die vorliegende Anwendung nicht sinnvoll, da sie auf *Simulator X* (Abschnitt 4.2) basiert, welcher genau diese „Vermischung“ von Aufgaben verhindern soll und selbst schon eine Komponenten-Infrastruktur für die Realisierung von 3D-Anwendungen bereitstellt.

Für die Integration in Simulator X über die Renderkomponente war deshalb eher eine „leichtgewichtige“ Lösung wünschenswert, welche nur die Grafikausgabe auf eine einfach benutzbare und erweiterbare Weise realisiert.

11.1 Übersicht

Im Rahmen dieser Arbeit wurde somit eine an die Erfordernisse des Projekts angepasste Render-Bibliothek auf Basis von OpenGL erstellt, die allerdings auch für weitere Projekte verwendet werden kann. Andere Anwendungen könnten ebenfalls auf Simulator X-Basis mit der neuen *Renderkomponente* (Kapitel 10) realisiert werden oder die Render-Bibliothek direkt nutzen.

Als Anbindung an OpenGL wird *LWJGL* (Abschnitt 4.3) genutzt. Es werden nur OpenGL-Funktionen genutzt, die in der Version 3.3 standardmäßig verfügbar sind. Das heißt, dass auf im OpenGL-Standard „deprecated“ markierte Funktionalität vollständig verzichtet wurde, welche in Zukunft komplett entfernt werden oder zu Geschwindigkeitsproblemen führen könnte. Damit ist zu erwarten, dass die Render-Bibliothek auch mit den aktuellsten und zukünftigen OpenGL-Versionen sowie neuer Grafikhardware – zumindest in den nächsten Jahren – kompatibel sein wird.

Die Render-Bibliothek orientiert sich in den Grundkonzepten an der für C++ verfügbaren *Visualization Library* [www:vislib]. Im Gegensatz zu vielen, umfassenden 3D-Engines sollten dem Benutzer der Bibliothek möglichst wenig Einschränkungen bei der Gestaltung seines Programms auferlegt werden. Bei der Konzeption der Render-Bibliothek stand die Wiederverwendbarkeit von einzelnen Bestandteilen im Vordergrund, die sich wieder zu höheren Abstraktionen zusammensetzen lassen.

Solche Abstraktionen werden – wenn sie allgemein verwendbar sind – direkt von der Render-Bibliothek angeboten, können aber auch speziell für eine bestimmte Anwendung erstellt werden. Durch das Prinzip soll der Programmierer von oft wiederkehrenden Aufgaben entlastet werden, aber trotzdem die vollen Möglichkeiten von OpenGL nutzen können, wenn nötig.

Höhere Abstraktionen sollen auch von Programmieren ohne tiefgreifende Computergrafik- und OpenGL-Kenntnisse genutzt werden können. Ein *Beispiel* (Abschnitt 11.6) dafür ist die Möglichkeit, auf einfachem Wege ein neues Grafikobjekt für die Darstellung von Modellelementen zu erstellen.

Die Library lässt sich grob in zwei Schichten, eine **Low-Level-API** und einer **Higher-Level-API** aufteilen, die im Folgenden vorgestellt werden.

11.2 Low-Level-API

Das als Grundlage genutzte LWJGL bietet nur eine sehr dünne Abstraktionschicht oberhalb von OpenGL, die vor allem dazu dient, OpenGL-Datentypen auf die Java VM abzubilden und umgekehrt. Die von LWJGL angebotenen Funktionen entsprechend weitestgehend denen, die durch den OpenGL-Standard vorgegeben und aus Programmiersprachen wie C bekannt sind.

Die Low-Level-API² sorgt nun für die objektorientierte Kapselung von OpenGL-Basiselementen und verschiedene Vereinfachungen für Standardfälle. Diese Schicht ermöglicht es, sehr nahe an den Konzepten von OpenGL zu entwickeln, ohne bei Routineaufgaben selbst viel OpenGL-Code schreiben zu müssen. Klassen- und Methodennamen orientieren sich, wie bei der Visualization Library, vorwiegend an den gekapselten OpenGL-Funktionen.

Hier soll nur eine kurze Übersicht über die Funktionalitäten gegeben werden, da diese für das Verständnis dieser Arbeit weniger wichtig sind und sehr OpenGL-spezifisch sind.

Vertex Buffer Object (VBO)-Klassen für verschiedene Datentypen Vereinfachen die Verwaltung des Grafikspeichers, beispielsweise den Transfer von Daten dorthin.

Uniform, UniformBlock- und VertexAttribute-Klassen Daten lassen sich so bequem zum Shaderprogramm auf der Grafikkarte übertragen. Die UniformBlock- und VertexAttribute-Klassen bauen auf der VBO-Abstraktion auf.

Beispiel für eine Uniform-Verwendung:

```
val color = ConstVec4(1, 1, 1, 1)
val colorUniform = GLUniform4f("color")
colorUniform.set(color)
```

Shader- und ShaderProgram-Klasse Übernehmen das Kompilieren und Linken von Shadern sowie die Verwaltung von Uniforms und UniformBlocks.

Renderbuffer und Framebuffer-Klassen (FBO) Abstraktionen für das Offscreen-Rendering (Anforderung 5)

Zeichenbefehle Kapseln die Zeichenfunktionen, welche OpenGL anweisen ein Objekt zu zeichnen. Es werden die Funktionen DrawArrays und DrawElements unterstützt.

Sonstige Abstraktionen: Textur- und Textursampler-Klassen, Viewport und Hintergrundfarbe (glClearColor), OpenGL-Einstellungen (wie glEnable oder glDepthFunc), VertexArrayObjects

11.3 Higher-Level-API

Diese Schicht stellt im Wesentlichen Schnittstellen und häufig benötigte Implementierungen für die Aufgaben bereit, die grafischen Objekte und den eigentlichen Rendervorgang zu beschreiben, der jene Objekte schließlich „auf den Bildschirm bringt“. Zur Implementierung werden die von der Low-Level-API bereitgestellten OpenGL-Abstraktionen genutzt.

In dieser Bibliothek wird ein solcher Rendervorgang durch sogenannte **RenderStages** beschrieben. Objekte, die von solchen RenderStages angezeigt werden können werden als **Drawable** bezeichnet.

11.3.1 Drawable

Zu zeichnende Objekte werden durch eine Klasse beschrieben, welche von einer Basisklasse **Drawable** abgeleitet ist. Solche **Drawable**-Klassen müssen eine Beschreibung der Geometrie (**Trait Mesh**), der Position und Größe (**Transformation**) und der Darstellungsweise (**Effect**) enthalten. Die Implementierung

²Zu finden im Package mmpe.renderer.gl

ist dabei sehr flexibel möglich und kann an die Anforderungen des konkret dargestellten Objekts und der Anwendung angepasst werden.

In den Traits sind nur Methoden vorgegeben, welche die von einem „Renderer“ benötigten Daten liefern müssen:

- Mesh stellt dem Renderer die Zeichenbefehle sowie Vertex-Attribute bereit, üblicherweise sind das Vertexkoordinaten, Normalen und Texturkoordinaten.
- Transformation liefert die Transformationsmatrix des Grafikobjekts.
- Effect ist für die Bereitstellung von Shader-Beschreibungen und zugehörigen Uniforms zuständig.

Ein Renderer kann selbst implementiert werden oder es kann eine RenderStage (nächster Abschnitt) dafür konfiguriert und genutzt werden.

Drawables stellen im Normalfall eine Schnittstelle für die Anwendung bereit, über die sich Attribute des Grafik-Objektes auslesen und setzen lassen. So könnte eine Transformation, die für ein bewegliches Objekt eingesetzt wird, einen Setter bereitstellen, der das Verändern der aktuellen Position erlaubt.

Die Render-Bibliothek stellt eine Reihe von Implementierungen dieser Traits zur Verfügung. Diese sind zwar auf die Bedürfnisse des i>PM3D-Projekts abgestimmt, aber möglichst allgemein gehalten und damit wiederverwendbar.

Sinnvollerweise werden Drawables erstellt, indem Traits zusammengemischt werden, die die genannten Basis-Traits Mesh, Transformation und Effect implementieren. So kann mit diesem Konzept beispielsweise ein Würfel definiert werden, indem eine entsprechende Mesh-Implementierung erstellt wird. Durch die Verwendung von unterschiedlichen Effect-Traits können auf einfacherem Wege verschiedene dargestellte Varianten eines Objekts erstellt werden.

Abbildung 11.1 zeigt dieses Drawable-Konzept an einem Beispiel. Es wird nur eine Auswahl der Methoden dargestellt.

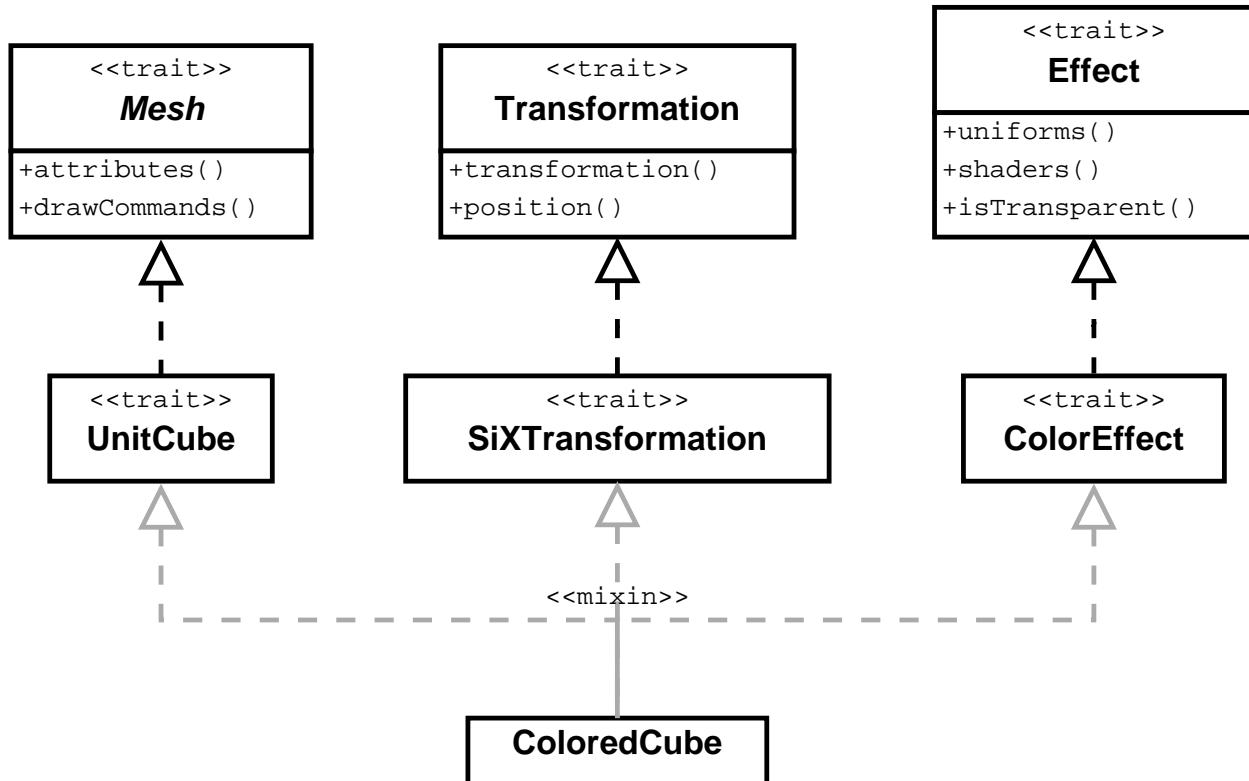


Abbildung 11.1: Zusammensetzung eines farbigen Würfels aus den Basis-Traits

Effects selbst können relativ kompliziert aufgebaut sein. Es ist sinnvoll, diese wieder aus verschiedenen Traits zusammenzusetzen, die Teilfunktionalitäten implementieren. Solche Traits sind in der Render-Bibliothek mit der Endung `-Addon` versehen. Beispielsweise existiert ein `PhongLightingAddon` für die Bereitstellung von Lichtparametern und ein `TextDisplayAddon`, welches die Anzeige von Schrift auf den Objekten implementiert.

Abbildung 11.2 zeigt ein Beispiel für einen Effect, der aus zwei Addons zusammengesetzt wird. Addons stellen oft Uniforms (`material-` und `lightUniforms` im Beispiel) zur Verfügung, die im Effect kombiniert und von der `uniforms`-Methode zurückgegeben werden. Mittels der Methoden `diffuse` und `specular` kann die Anwendung die Reflexionseigenschaften eines Objekts verändern.

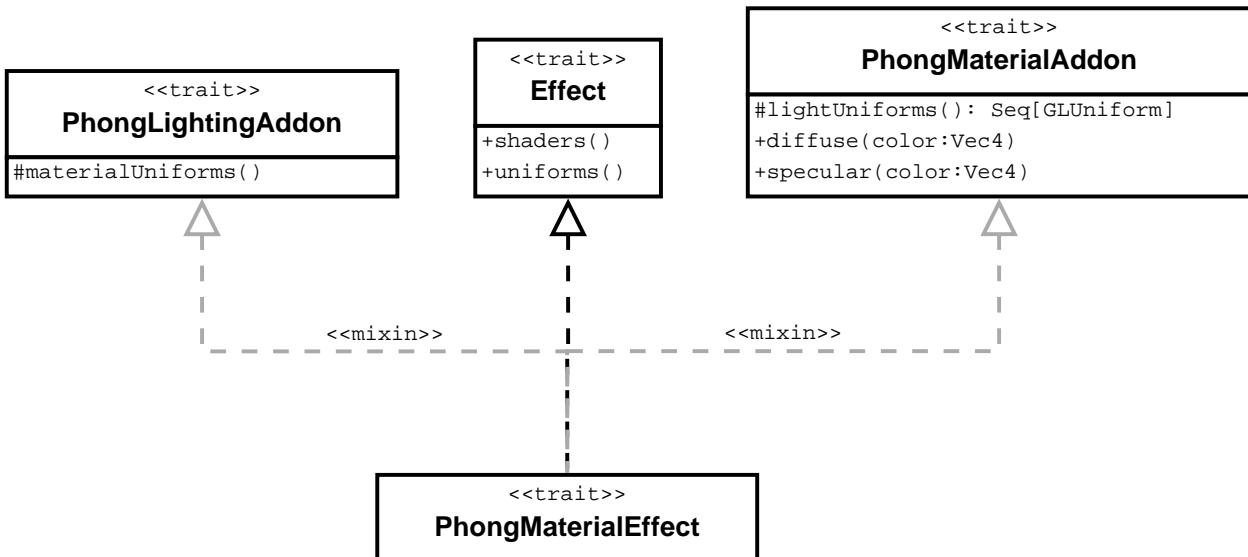


Abbildung 11.2: Zusammengesetzter PhongMaterialEffect

Ressourcen, die potenziell von vielen verschiedenen Drawables geteilt werden können werden im Drawable nur durch eine abstrakte Beschreibung dargestellt. Texturen werden über eine `TextureDefinition` beschrieben; Shaderquelldateien über eine `ShaderDefinition`.

11.3.2 RenderStage

`RenderStages` sind für das Zeichnen der grafischen Objekte zuständig. Die Anwendung übergibt einer `RenderStage` einmal pro Frame³ alle zu zeichnenden `Drawables`. Diese werden in der bereitgestellten Implementierung der `RenderStage` zuerst sortiert und anschließend gezeichnet. Eine Sortierung wird durchgeführt, um transluzente Objekte (Anforderung 7) in der richtigen Reihenfolge zu zeichnen sowie um unnötige Zeichenoperationen und OpenGL-Zustandswechsel zu vermeiden. Durch Angabe einer Render-Priorität in den `Drawables` kann manuell eine bestimmte Reihenfolge erzwungen werden, wenn dies für spezielle Zeichenaufgaben nötig ist.

Von der `RenderStage` werden zu den von `Drawables` definierten `Texture`- und `ShaderDefinition`s Objekte der Low-Level-API nach Bedarf erzeugt. Diese werden für das Zeichnen von mehreren `Drawables` wiederverwendet, um Grafikspeicher und Zeit zu sparen.

Abgegrenzte Funktionalitäten können in ein `RenderStagePlugin` ausgelagert werden. So stellt die Render-Bibliothek unter anderem Plugins für die Verwaltung von Texturen und die Umsetzung von Lichtquellen bereit.

Abbildung 11.3 zeigt eine zusammengesetzte `RenderStage`.

³Wie in „Frames Per Second“ (FPS). Damit ist ein „Einzelbild“ gemeint.

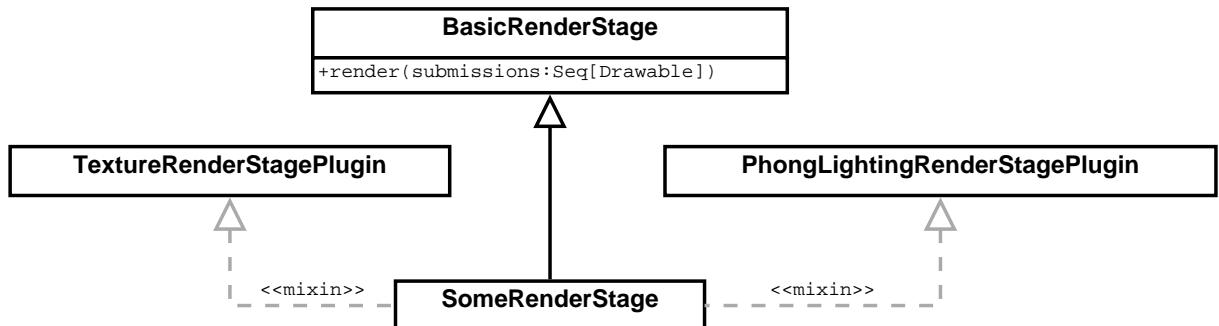


Abbildung 11.3: RenderStage mit eingemischten Plugin-Traits

11.3.3 Weitere Abstraktionen

Licht

Die Render-Bibliothek unterstützt das Phong-Beleuchtungsmodell, welches pixelgenau ausgewertet wird. Für die Anwendung werden Klassen bereitgestellt, die die von „altem“ OpenGL bekannten „Lichtquellen“ bereitstellen und sich an deren Schnittstelle orientieren. Lichtquellen können entweder entfernungsabhängig (`PositionalLight`) oder -unabhängig sein (`DirectionalLight`).

Implementiert wird die Beleuchtung auf Scala-Seite durch das Zusammenspiel des `PhongLightingRenderStagePlugins` mit dem Effect-Addon `PhongLightingAddon`. Die eigentlichen Lichtberechnungen wurden in GLSL-Shaderfunktionen implementiert, die von verschiedenen Fragment-Shadern genutzt werden können.

Kamera

Die Klasse `Camera` repräsentiert eine bewegliche und rotierbare Kamera, die einer `RenderStage` zugewiesen werden kann und damit die Perspektive des Betrachters festlegt. Es werden die von OpenGL bekannten Funktionen (als Methoden von `Camera`) angeboten, die eine perspektivische (`glFrustum`, `gluPerspective`, `gluLookAt`) oder orthogonale Projektion (`glOrtho`) konfigurieren. Außerdem stellt die Klasse verschiedene Methoden bereit, die für Umrechnungen von Bildschirm- in 3D-Raumkoordinaten und umgekehrt genutzt werden können (analog zu den OpenGL-Funktionen `glProject` und `glUnProject`).

Diese werden im Projekt von Eingabegeräten genutzt, die mit 2D-Daten arbeiten und diese beispielsweise für die Auswahl von 3D-Objekten entsprechend umrechnen müssen. Aufgrund der von Simulator X geforderten Komponentenaufteilung werden die Methoden von den Nutzern nicht direkt aufgerufen, sondern von der [Renderkomponente](#) (Kapitel 10) gekapselt. Nutzer müssen analog zu den Methoden definierte Nachrichten verwenden, die über das Kommunikationssystem von Simulator X verschickt werden.

11.4 COLLADA2Scala-Compiler

Das Laden von Modellen direkt aus COLLADA-XML-Dateien ist relativ zeitaufwändig. Außerdem unterstützt der genutzte COLLADA-Loader [Hol12] bisher noch nicht die Wiederverwendung der geladenen Modelldaten. So wird für jede Instanz eines solchen 3D-Modells zusätzlicher Grafikspeicher belegt. Ein weiteres Problem ist, dass der Loader „fertige“ `Drawables` liefert, die nicht für die Darstellung von Modellelementen (Knoten und Kanten) genutzt werden können.

Aufgrund dessen wurde ein „Compiler“ entwickelt, der mit Hilfe des COLLADA-Loaders ein Modell lädt und daraus eine Repräsentation der in dem Modell definierten Geometrie in Scala-Code erzeugt. Die so erzeugte Scala-Quelldatei enthält ein Trait, das *Mesh* (Unterabschnitt 11.3.1)) implementiert.

Optional kann direkt eine .jar-Datei erstellt werden.

Am Ende des Kapitels wird im Anwendungsbeispiel die Nutzung des COLLADA2Scala-Compilers demonstriert.

11.5 Spezielle Erweiterungen für i>PM3D

In diesem Abschnitt werden abschließend die Erweiterungen vorgestellt, die speziell für die Realisierung der Prozessvisualisierung bereitgestellt werden (*Anforderung (g)* (Abschnitt 1.3)). Hier wird auch gezeigt, wie die oben beschriebenen Ebenen der Render-Bibliothek und die GLSL-Shader zusammenwirken. Außerdem soll verdeutlicht werden, wie *Drawables* (Unterabschnitt 11.3.1) als Schnittstelle zwischen grafischer Darstellung und Anwendung dienen.

11.5.1 Unterstützung für deaktivierte, hervorgehobene und selektierte Elemente

Für die *Visualisierungsvarianten für interaktive Modelleditorien* (Abschnitt 8.2) wurde eine Fragment-Shaderfunktion erstellt, welche die Farbe eines Objektes abhängig von den aktiven Visualisierungsvarianten verändert kann. Ein Shader, der diese Funktion nutzt, definiert Uniforms, mit welchen die Varianten ausgewählt werden können.

Auf Scala-Seite werden diese Uniforms vom `SelectionHighlightAddon` verwaltet, welches auch eine Schnittstelle für die Anwendung bereitstellt.

Die Varianten lassen sich über im Addon definierte Setter aktivieren:

```
drawable.disabled = false  
drawable.highlighted = false  
drawable.selectionState = DrawableSelectionState.Selected
```

Durch den Aufruf eines solchen Setters wird die zugehörige Uniform geändert und die Änderung somit zum Shaderprogramm weitergegeben, nachdem etwaige Konvertierungen durchgeführt wurden.

Zusätzlich können noch folgende Parameter gesetzt werden:

- `borderWidth`: Breite des Selektionsrahmens.
- `highlightFactor`: Wert, mit dem die berechnete Farbe multipliziert wird um Hervorhebung darzustellen. Bei dunklen Grundfarben wird stattdessen mit $1 / \text{highlightFactor}$ multipliziert.

„Deaktiviert“ wird durch einen Grauton dargestellt, der wie folgt aus den Komponenten der Grundfarbe berechnet wird: $\text{grauwert} = (\text{rot} + \text{blau} + \text{grün}) * 0.2$.

Außerdem wird das Objekt transluzent gezeichnet. Der Selektionsrahmen wird im deaktivierten Zustand abhängig von der resultierenden Helligkeit von „grauwert“ entweder hellgrau oder dunkelgrau dargestellt.

Die Shaderfunktion zeichnet den „Selektionsrahmen“ abhängig von den (2D)-Texturkoordinaten, die üblicherweise von 0 bis 1 reichen. Auf jeder Seite wird ein Bereich mit der Breite „`borderWidth`“ als Rahmen in der Komplementärfarbe zum Hintergrund gezeichnet.

So wird durch die Texturkoordinaten die Form des Rahmens definiert; für die in der Arbeit verwendeten Objekte war dies ausreichend. Jedoch könnten sich bei anderen Figuren Probleme ergeben, da die Texturkoordinaten auch für die Ausrichtung der Textur oder der Schrift genutzt werden. Für solche

Objekte könnte allerdings leicht ein zusätzliches Vertex-Attribut definiert werden, welches die Koordinaten für die Positionierung des Rahmens liefert.⁴

11.5.2 Darstellung von Text

Für die Beschriftung von Modellknoten wurde eine gut lesbare und trotzdem einfach umsetzbare Technik für das Rendering von Schrift benötigt. Hierfür wurde die 2D-API (`java.awt`) der Java-Klassenbibliothek zur Hilfe genommen. Zur Verwendung mit OpenGL wird die Schrift in eine Textur geschrieben, die dann auf die Objekte aufgebracht werden kann. Um die Darstellungsqualität zu erhöhen, wird die Antialiasing-Funktion von `Graphics2D` genutzt.

Zur Darstellung von Text müssen Drawables den Trait `TextDisplayAddon` einmischen und die genutzte `RenderStage` muss die Plugins `TextDisplayRenderStagePlugin` sowie `TextureRenderStagePlugin` einbinden. Im `RenderStagePlugin` wird bei jeder Änderung des Textes oder Schrifteinstellungen die Schrift-Textur neu erstellt, so dass im nächsten Frame der neue Text angezeigt wird.

Der Text kann im Drawable mit

```
drawable.text = "irgendein Text"
```

verändert werden. Außerdem werden Einstellmöglichkeiten für die Schriftart, -größe und -stil (`java.awt.Font`) und die Schriftfarbe (`java.awt.Color`) angeboten.

Der Text wird zentriert angezeigt und am Wortende umgebrochen, falls der horizontale Platz nicht ausreicht. Die „Schriftgröße“ wird als Mindestgröße interpretiert; falls ein Objekt eine Skalierung größer eins aufweist, wird die Größe der Schrift proportional mitskaliert. Bei einer Skalierung kleiner eins wird der für die Schrift zur Verfügung stehende Platz verkleinert.

Die Skalieroperationen werden von einer Shaderfunktion realisiert.

Um auch bei größeren Entferungen von der Kamera und kleiner Schrift noch eine angemessene Lesbarkeit zu erreichen kann Mipmapping genutzt werden, das auch von der Render-Bibliothek unterstützt wird. Aufgrund von Problemen mit verschiedenen Grafikkarten, die für das Projekt getestet wurden, ist dies standardmäßig jedoch nicht aktiviert.

11.5.3 SVarSupport - Einbindung der Modell-Drawables in i>PM3D

Visualisierungsparameter der Modellelemente werden über die `ModelComponent` bereitgestellte `SVars` (Unterabschnitt 9.3.3) gesetzt. Den `SVar`-Wert zu verändern hat alleine noch keinen Effekt; die Wertänderungen müssen an die Drawables weitergeleitet werden, welche die Anbindung an die Grafikschnittstelle realisieren.

Die Verbindung der `SVars` mit den Attributen der Drawables erfolgt über Traits, die das Trait `SVarSupport` implementieren. Solche `SVarSupports` werden in Modell-Drawables eingemischt, wie im Anwendungsbeispiel im folgenden Abschnitt gezeigt wird.

Diese Traits stellen eine Methode, `connectSVars` bereit, die von der Renderkomponente aufgerufen wird nachdem diese ein `Drawable` erzeugt hat. So werden in dieser Methode für alle vom Trait unterstützten `SVars` *Observe-Handler* registriert, die bei jeder Änderung des `SVar`-Wertes aufgerufen werden. Üblicherweise leiten diese Funktionen die neuen Werte an Setter des Drawables weiter, wie sie in den vorherigen Abschnitten gezeigt wurden.

Für `SVars`, deren Typ erst zur Laufzeit bekannt ist, kann der Methode eine „Ersetzungsliste“ übergeben werden. Eine solche Ersetzung ist beispielsweise für die Darstellung von Text auf Modellknoten nötig.

⁴Dies ist auch ein Beispiel, dass die Flexibilität von modernem OpenGL (und der Render-Bibliothek) zeigt, die im Gegensatz zu alten OpenGL-Versionen beliebige Vertex-Attribute unterstützen.

Im *Editor-Metamodell* (Unterabschnitt 7.2.2) wird festgelegt, welches *Domain-Attribut* als Text dargestellt werden soll. Die Modellkomponente liest den Namen des Attributs aus und definiert eine Ersetzung der Text-SVar durch die entsprechend benannte *Domain-Model-SVar* (Unterabschnitt 9.3.3). Beispielsweise wird für einen Prozessknoten die Text-SVar durch die `model.function`-SVar ersetzt. Für letztere SVar wird so ein *Observe-Handler* registriert, der dem Setter für den aktuellen Text des Drawables jede Änderung an `model.function` weitergibt. Aufgrund dessen wird bei jeder Änderung an dieser SVar (bspw. durch eine Eingabe in einem Menü) der sichtbare Text auf dem Grafikobjekt angepasst.

Das gleiche Prinzip wird für Visualisierungsparameter (bspw. Farben oder die Schriftart) aus dem Editor-Modell angewendet. Abbildung 11.4 zeigt beispielhaft, wie sich eine Änderung an der Hintergrundfarbe eines Prozessknotens Benutzer auswirkt und welche Schritte vom Editor-Menü bis zur Grafikkarte durchlaufen werden.

Beispiele für SVarSupports

Für die im vorherigen Unterabschnitt beschriebene Textdarstellung wird das Trait `TextDisplaySVarSupport` angeboten. Im Normalfall wird dieses zusammen mit dem `BackgroundSVarSupport` genutzt, welches das Setzen der Hintergrundfarbe übernimmt. Das Trait `SelectionHighlightSVarSupport` stellt die Anbindung der *Visualisierungsvarianten* (Unterabschnitt 11.5.1) bereit.

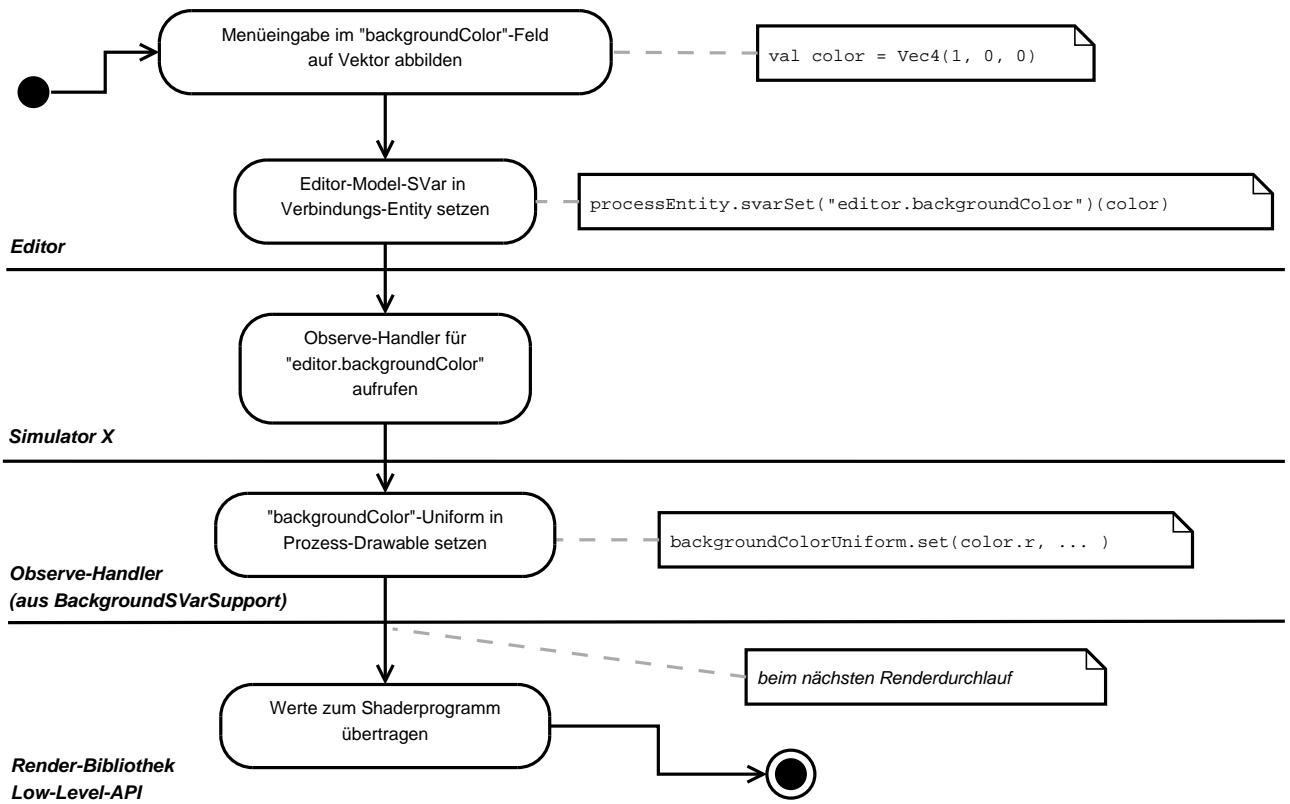


Abbildung 11.4: Ablauf bei Änderung der Hintergrundfarbe eines Prozesses durch den Benutzer

11.6 Anwendungsbeispiel: Erstellen von neuen Modell-Figuren

Zum Abschluss wird gezeigt, wie sich ein neues Grafikobjekt erstellen lässt, das für die Visualisierung eines Knotens eingesetzt werden soll. Dies ist die Fortsetzung des [Anwendungsbeispiels](#) (Abschnitt 7.5) für das Hinzufügen eines neuen Modellelements zum Metamodell.

Die Geometrie des Objekts kann zum Einen manuell erstellt werden, indem das Trait `Mesh` implementiert wird. Als Vorlage kann eines der mitgelieferten Meshes, wie `mmpe.renderer.mesh.UnitCube` genutzt werden.

Einfacher ist die Nutzung des COLLADA2Scala-Compilers, der wie folgt aufgerufen werden kann (Linux):

```
$ collada2scala pyramid.dae test.Pyramid pyramid.jar
```

Damit wird aus einer COLLADA-Datei `pyramid.dae` eine `pyramid.jar` erstellt, die im Package `test` einen Mesh `Pyramid` enthält.

Im nächsten Schritt wird die neue Figur zum object `mmpe.model.BaseDrawables` hinzugefügt.

Es soll eine Pyramide erstellt werden, auf deren Seiten Text angezeigt werden kann. Dafür kann beispielsweise die `TextBox` als Vorlage genommen und wie folgt abgeändert werden:

```
1 class TextPyramid extends EmptyDrawable("textPyramid")
2   with Pyramid
3   with SiXTransformation
4   with SelectableAndTextEffect
5   with SelectionHighlightSVarSupport
6   with TextDisplaySVarSupport
7   with BackgroundSVarSupport
```

Geändert wurde nur die `Mesh`-Implementierung in der 2. Zeile sowie der Name des Objekts in der 1. Zeile.

Abschließend wird in `mmpe.model.ModelDrawableFactory` zur Fallunterscheidung in der Methode `createDrawables` („`figureFqn match ...`“) ein weiterer Fall hinzugefügt:

```
case "test.TextPyramid" =>
  val drawable = new TextPyramid
  Seq(drawable)
```

Nun kann die texturierte Pyramide im Editor-Metamodell genutzt werden. Das hier angegebene „`test.TextPyramid`“ entspricht dem `scalaType`, wie er im Metamodell gesetzt muss werden muss um diese Figur zu referenzieren.

11.6.1 Anmerkungen

Am ersten Code-Beispiel ist zu sehen, wie ein `Drawable` für ein Modellelement prinzipiell definiert wird. Die Zeilen 2 bis 4 geben die von [Drawable](#) (Unterabschnitt 11.3.1) geforderte Implementierung an. Es wird von der Renderkomponente vorausgesetzt, dass `SiXTransformation` für alle 3D-Drawables genutzt wird. `SelectableAndTextureEffect` wird für alle texturierten Figuren genutzt, die die [Visualisierungsvarianten](#) (Unterabschnitt 11.5.1) unterstützen. Analog dazu ist `SelectableAndTextEffect` für die Textdarstellung definiert, welcher das [TextDisplayAddon](#) (Unterabschnitt 11.5.2) nutzt.

In den letzten drei Zeilen werden die `SVarSupports` eingemischt, welche die Verbindung zur Anwendung herstellen.

Fazit und Ausblick

In der vorliegenden Arbeit wurde die Einbindung von Prozessmodellen und deren Visualisierung im i>PM3D-Prototypen sowie die Spezifikation der verwendeten Modellierungssprache durch Metamodelle vorgestellt. Aufgrund der Komplexität dieses Themas, der dabei auftretenden Fragestellungen und der insgesamt noch eher schwach ausgeprägten Forschungslage zum Thema 3D-Visualisierung von Prozessen kann dies jedoch nur ein Anfang sein. In Zukunft wäre es wichtig, die Effizienz von 3D-Visualisierungsansätzen genauer zu untersuchen, wofür der Prototyp eine Grundlage bietet. Dabei sollten auch Benutzer eingebunden werden, die sich zwar mit dem Thema Prozessmodellierung auskennen, jedoch noch kaum Erfahrung im Umgang mit 3D-Umgebungen haben.

Eingabe und Benutzerinteraktion wurden in dieser Arbeit größtenteils ausgeklammert, da diese von [Hol12] und [Buc12] im Rahmen des Projekts bearbeitet wurden. Dies ist aber eigentlich ein „untrennbarer“ Bestandteil, wenn man die Effizienz von 3D-Visualisierungen – insbesondere in interaktiven Modellierungswerkzeugen – bewerten will. Bei eigenen Versuchen mit dem Prototypen hat sich besonders gezeigt, dass die Effizienz mit dem Vorhandensein gut funktionierender und intuitiver Eingabemethoden steht und fällt. Daher ist es besonders lohnenswert, zukünftige Arbeiten in diese Richtung zu lenken.

Probleme und naheliegende Erweiterungsmöglichkeiten, welche speziell die Visualisierung betreffen, wurden schon in Abschnitt 8.6 und Abschnitt 8.7 besprochen. Besonders wichtig wären in diesem Zusammenhang umfangreichere Konfigurationsmöglichkeiten, die nach Möglichkeit auch in das Metamodell aufgenommen werden sollten, um eine einheitliche Konfiguration der Visualisierungsparameter zu ermöglichen. Außerdem sollten in einer Weiterentwicklung automatische Verfahren zur Unterstützung des Benutzers, beispielsweise Layout-Algorithmen und eine intelligente Wahl der Lichtparameter integriert werden. Daneben kann untersucht werden, inwieweit sich geometriebasierte Ansätze, wie bspw. aus i>PM:sup:2 bekannt, in den dreidimensionalen Raum übertragen lassen.

Einige bei der Vorstellung von verwandten Arbeiten zur Visualisierung gezeigte Nutzungsmöglichkeiten der dritten Dimension können mit dem Prototypen schon realisiert werden. Vielversprechend ist die Einbettung der abstrakten Prozessmodelle in eine *virtuelle Umgebung* (Unterabschnitt 3.4.2) mit Abbildern realer Objekte, wie es mit i>PM3D prinzipiell schon möglich ist, da beliebige 3D-Objekte in die Szene eingebunden werden können. Elemente nach ihrer „Wichtigkeit“ oder nach anderen Kriterien in unterschiedlicher Entfernung zu zeigen, ist aufgrund der freien Platzierbarkeit der Elemente umsetzbar, verursacht allerdings viel Aufwand für den Benutzer. Hierfür wäre eine algorithmische Unterstützung vonnöten, die beispielsweise anhand von Attributwerten aus dem Prozessmodell Elemente in einer bestimmten Weise anordnet.

Im Prototypen lassen sich gleichzeitig mehrere Modelle darstellen. So ist es prinzipiell möglich, hierarchische Modelle mit mehreren Verfeinerungsstufen darzustellen, wie es für die Darstellung von kompositen

Prozessen benötigt wird. Jede Stufe kann so durch ein Modell repräsentiert werden, dessen Modellelemente sich nach Bedarf auf einer *Modellierungsfläche* (Abschnitt 8.3) anordnen lassen.

Es fehlt allerdings noch entsprechende Unterstützung, um Beziehungen zwischen mehreren Modellen darzustellen und abzuspeichern. Verschiedene Modelltypen anzeigen und verknüpfen zu können, ist eine weitere lohnende Entwicklungsmöglichkeit. Dazu müsste die gleichzeitige Verwendung von mehreren Metamodellen durch die Editorkomponente unterstützt werden ([siehe](#) Unterabschnitt 9.2.3).

Da am Lehrstuhl, an dem dieses Projekt entstanden ist, ebenfalls die umfangreiche Metamodellierungsumgebung OMME entwickelt wird, wäre eine Integration von i>PM3D in diese überlegenswert. Für die Realisierung des Prototypen wurden schon Prinzipien aus OMME wie die Austauschbarkeit und Separation der Metamodelle für Editor und Modellierungsdomäne sowie die Spezialisierung von Instanzen genutzt. Um die Flexibilität weiter zu erhöhen und die Anwendbarkeit für verschiedene Modellierungsdomänen zu vereinfachen, sollte nach dem Vorbild von MDF die Spezifikation der grafischen Repräsentation vom Editor-Modell getrennt werden.

Vielversprechend könnten auch Kombinationen von 3D und 2D-Visualisierungen sein, um die Vorteile aus beiden „Welten“ nutzen zu können. Dies könnte durch die Integration von 2D-Elementen in i>PM3D realisiert werden. Außerdem wäre es technisch möglich, i>PM3D auf Basis von *Simulator X* in die Eclipse-Plattform zu integrieren. Damit könnten 3D-Editoren neben mit MDF spezifizierten 2D-Editoren in derselben Umgebung genutzt werden.

Der Anpassbarkeit der Visualisierung und der Modellierungskonstrukte, die durch die Verwendung des Metamodellierungs-Ansatzes entsteht, muss auch eine entsprechende Flexibilität der Implementierung gegenüberstehen. In der Modellanbindung wurde diese noch nicht ganz erreicht, da durch diese bisher eine feste Auswahl an Visualisierungsparametern vorgegeben wird, um die Implementierung zu erleichtern. Dies könnte – möglicherweise auch mit Änderungen in künftigen Versionen von *Simulator X* – wohl flexibler realisiert werden. *Simulator X* war aber grundsätzlich für die Realisierung des Projekts gut geeignet; besonders die Entkoppelung von Funktionalitäten durch die Komponentenaufteilung und das *Entity / SVar*-Konzept, welche für die Bereitstellung der Modelfunktionalitäten genutzt wurden haben sich als positiv herausgestellt. Weitere Funktionalitäten zu integrieren, wie beispielsweise die Simulation von Prozessen, visualisiert durch die Anzeige von Prozessdaten in der 3D-Szene und durch Animationen von prozessrelevanten Entitäten (bspw. Personen oder Werkstücke) ist durch die Einbindung von zusätzlichen Komponenten leicht möglich. Weiterhin lässt sich auf diesem Wege eine beliebige Erweiterungsmöglichkeiten realisieren, die allgemein nützlich sind oder auf einen bestimmten Anwendungsfall zugeschnitten sein können.

Durch die Render-Bibliothek, die im Rahmen dieser Arbeit entwickelt wird, ist eine Integration von weiteren Figuren, wie in einem Anwendungsbeispiel im letzten Kapitel gezeigt wurde, leicht möglich. Diese könnten auch speziell auf eine 2D-Darstellung ausgelegt sein, um einen 2D-Editor mit i>PM3D zu realisieren, wobei hier noch einige Änderungen an der restlichen Implementierung vorgenommen werden müssten. Allgemein hat sich die Render-Bibliothek durch ihre Erweiterbarkeit und der leichten Einbindung in das Anwendungsprogramm als sehr praktisch für die Umsetzung herausgestellt. Mit jener sollten sich auch andere 3D-Anwendungen gut umsetzen lassen, ohne die „Freiheiten“ des Programmierers einzuschränken, möglicherweise auch auf Basis von Simulator X über die hier entwickelte Integration über die Renderkomponente.

Insgesamt lässt sich zum Projekt sagen, dass ein durchaus benutzbarer Prototyp eines 3D-Prozessmodellierungswerkzeugs entstanden ist, der als Basis für weitere Entwicklungen dienen kann (und sollte). Prinzipiell lässt sich i>PM 3D auch schon für die Visualisierung und Bearbeitung von anderen Modelltypen nutzen, die sich ebenfalls in einer graphbasierten Form darstellen lassen, beispielsweise für die Modellierung von Proteinen oder Reaktionsnetzwerken in der Bioinformatik. Eine ausführbare Version von i>PM3D befindet sich auf der beigelegten **DVD** (Anhang B).

Literaturverzeichnis

Literatur

- [AG09] Joacim Alvergren und Jonatan Granqvist. *3D Visualization of Complex Software Models: Practical Use and Limitations*. Bachelor of Software Engineering and Management Thesis. Universität Göteborg. 2009.
- [AHH08] Tomas Akenine-Möller, Eric Haines und Natty Hoffman. *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., 2008. ISBN: 987-1-56881-424-7.
- [AK00] Colin Atkinson und Thomas Kuehne. *Meta-level Independent Modelling*. 2000.
- [BD04] Michael Balzer und Oliver Deussen. „Hierarchy Based 3D Visualization of Large Software Structures“. In: *Proceedings of the conference on Visualization '04. VIS '04*. Washington, DC, USA: IEEE Computer Society, 2004, 598.4-. ISBN: 0-7803-8788-0. DOI: [10.1109/VISUAL.2004.39](https://doi.org/10.1109/VISUAL.2004.39).
- [Bet+08] Stefanie Betz u. a. „3D Representation of Business Process Models.“ In: *MobiIS*. Hrsg. von Peter Loos u. a. Bd. 141. LNI. GI, 2008, S. 73–87. URL: <http://dblp.uni-trier.de/db/conf/mobis/mobis2008.html#BetzEKKL0T08>.
- [bpmn] OMG. "BPMN 2.0". URL: <http://www.omg.org/spec/BPMN/2.0/PDF/>.
- [BRJ99] Grady Booch, James Rumbaugh und Ivar Jacobson. *The Unified Modeling Language user guide*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-57168-4.
- [Bro10] Ross A. Brown. „Conceptual modelling in 3D virtual worlds for process communication“. In: *Proceedings of the Seventh Asia-Pacific Conference on Conceptual Modelling - Volume 110*. APCCM '10. Brisbane, Australia: Australian Computer Society, Inc., 2010, 25–32. ISBN: 978-1-920682-92-7. URL: <http://dl.acm.org/citation.cfm?id=1862330.1862336>.
- [Buc12] Sebastian Buchholz. *i>PM 3D – Ein Prozessmodellierungswerkzeug für drei Dimensionen – Evaluation und Anbindung verschiedener Eingabemethoden mit besonderem Fokus auf Gesteninteraktion und Motion Tracking*. Bachelor-Thesis. Universität Bayreuth. 2012 (eingereicht).

- [collada] *COLLADA Specification 1.5*. URL: http://www.khronos.org/files/collada_spec_1_5.pdf.
- [Cos+02] G. Costagliola u. a. „A Classification Framework to Support the Design of Visual Languages“. In: *Journal of Visual Languages & Computing* 13.6 (2002), S. 573 –600. ISSN: 1045-926X. DOI: [10.1006/jvlc.2002.0234](https://doi.org/10.1006/jvlc.2002.0234).
- [CSD93] Carolina Cruz-Neira, Daniel J. Sandin und Thomas A. DeFanti. „Surround-screen projection-based virtual reality: the design and implementation of the CAVE“. In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '93. Anaheim, CA: ACM, 1993, 135–142. ISBN: 0-89791-601-8. DOI: [10.1145/166117.166134](https://doi.org/10.1145/166117.166134).
- [CSW08] Tony Clark, Paul Sammut und James Willans. *Applied metamodeling: a foundation for language driven development*. Hrsg. von CetevaEditors. Bd. 2005. Ceteva, 2008. URL: <http://eprints.mdx.ac.uk/6060/>.
- [Dam97] Andries van Dam. „Post-WIMP user interfaces“. In: *Commun. ACM* 40.2 (Feb. 1997), 63–67. ISSN: 0001-0782. DOI: [10.1145/253671.253708](https://doi.org/10.1145/253671.253708).
- [Dwy01] Tim Dwyer. „Three dimensional UML using force directed layout“. In: *Proceedings of the 2001 Asia-Pacific symposium on Information visualisation - Volume 9*. APVis '01. Sydney, Australia: Australian Computer Society, Inc., 2001, 77–85. ISBN: 0-909925-87-9. URL: <http://dl.acm.org/citation.cfm?id=564040.564050>.
- [EAT09] Niklas Elmquist, Ulf Assarsson und Philippas Tsigas. „Dynamic Transparency for 3D Visualization: Design and Evaluation“. In: *International Journal of Virtual Reality* 8.1 (2009), S. 65–78.
- [Fis+11] Martin Fischbach u. a. „SiXton's curse - Simulator X demonstration“. In: *Virtual Reality Conference (VR), 2011 IEEE*. 2011, S. 255–256.
- [FLW09] Christian Fröhlich, Marc Erich Latoschik und Ipke Wachsmuth. „Virtuelle Werkstatt – Multimodale Interaktion für intelligente virtuelle Konstruktion“. In: *8. Paderborner Workshop Augmented & Virtual Reality in der Produktentstehung*. Paderborn: HNI, 2009, S. 241–255.
- [GK98] Joseph Gil und Stuart Kent. „Three dimensional software modelling“. In: *Proceedings of the 20th international conference on Software engineering*. ICSE '98. Kyoto, Japan: IEEE Computer Society, 1998, 105–114. ISBN: 0-8186-8368-6. URL: <http://dl.acm.org/citation.cfm?id=302163.302174>.
- [GRR99] Martin Gogolla, Oliver Radfelder und Mark Richters. „Towards three-dimensional representation and animation of UML diagrams“. In: *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard*. UML'99. Fort Collins, CO, USA: Springer-Verlag, 1999, 489–502. ISBN: 3-540-66712-1. URL: <http://dl.acm.org/citation.cfm?id=1767297.1767349>.
- [Hal+08] Harry Halpin u. a. „Exploring Semantic Social Networks Using Virtual Reality“. In: *Proceedings of the 7th International Conference on The Semantic Web*. ISWC '08. Karlsruhe, Germany: Springer-Verlag, 2008, 599–614. ISBN: 978-3-540-88563-4. DOI: [10.1007/978-3-540-88564-1_38](https://doi.org/10.1007/978-3-540-88564-1_38).
- [HNR72] Peter E. Hart, Nils J. Nilsson und Bertram Raphael. „Correction to Ä Formal Basis for the Heuristic Determination of Minimum Cost Paths““. In: *SIGART Bull.* 37 (1972), 28–29. ISSN: 0163-5719. DOI: [10.1145/1056777.1056779](https://doi.org/10.1145/1056777.1056779).
- [HO09] Philipp Haller und Martin Odersky. „Scala Actors: Unifying thread-based and event-based programming“. In: *Theoretical Computer Science* 410.2–3 (2009), S. 202 –220. ISSN: 0304-3975. DOI: [10.1016/j.tcs.2008.09.019](https://doi.org/10.1016/j.tcs.2008.09.019).
- [Hof+03] Arthur ter Hofstede u. a. „Business Process Management: A Survey“. In: *Business Process Management*. Hrsg. von Mathias Weske. Bd. 2678. Lecture

- Notes in Computer Science. Springer Berlin / Heidelberg, 2003, S. 1019–1019. ISBN: 978-3-540-40318-0. DOI: [10.1007/3-540-44895-0_1](https://doi.org/10.1007/3-540-44895-0_1).
- [Hol12] Uli Holtmann. *i>PM 3D – Ein Prozessmodellierungswerkzeug für drei Dimensionen – Konzeption und Implementierung einer multimodalen Bedienschnittstelle sowie Optimierung von Nutzbarkeit und Bedienbarkeit*. Bachelor-Thesis. Universität Bayreuth. 2012 (eingereicht).
- [HW09] Danny Holten und Jarke J. van Wijk. „A user study on visualizing directed edges in graphs“. In: *Proceedings of the 27th international conference on Human factors in computing systems*. CHI ’09. Boston, MA, USA: ACM, 2009, 2299–2308. ISBN: 978-1-60558-246-7. DOI: [10.1145/1518701.1519054](https://doi.org/10.1145/1518701.1519054).
- [ipm] ProDatO GmbH. *i>ProcessManager. Handbuch i>PM Integrated Process Manager*. Version 2.5.0, 2005.
- [JB96] Stefan Jablonski und Christoph Bussler. *Workflow Management: Modeling Concepts, Architecture and Implementation*. International Thomson Computer Press, 1996.
- [JG08] Stefan Jablonski und Manuel Goetz. „Perspective oriented business process visualization“. In: *Proceedings of the 2007 international conference on Business process management*. BPM’07. Brisbane, Australia: Springer-Verlag, 2008, 144–155. ISBN: 3-540-78237-0, 978-3-540-78237-7. URL: <http://dl.acm.org/citation.cfm?id=1793714.1793732>.
- [KN09] Anne-Katrin Krolovitsch und Linda Nilsson. *3D Visualization for Model Comprehension*. Bachelor of Applied Information Technology Thesis. Universität Göteborg. 2009.
- [LT11] Marc Erich Latoschik und Henrik Tramberend. „Simulator X: A Scalable and Concurrent Software Platform for Intelligent Realtime Interactive Systems“. In: *Proceedings of the IEEE VR 2011*. 2011.
- [MHS08] Paul McIntosh, Margaret Hamilton und Ron van Schyndel. „X3D-UML: 3D UML State Machine Diagrams“. In: *Model Driven Engineering Languages and Systems*. Hrsg. von Krzysztof Czarnecki u. a. Bd. 5301. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, S. 264–279. ISBN: 978-3-540-87874-2. DOI: [10.1007/978-3-540-87875-9_19](https://doi.org/10.1007/978-3-540-87875-9_19).
- [opengl] *OpenGL 3.3 Core Profile Specification*. Online. URL: <http://www.opengl.org/registry/doc/glspec33.core.20100311.withchanges.pdf>.
- [OSV11] Martin Odersky, Lex Spoon und Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*. 2nd. USA: Artima Incorporation, 2011. ISBN: 0981531644, 9780981531649.
- [Par09] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. 1st. Pragmatic Bookshelf, 2009. ISBN: 193435645X, 9781934356456.
- [PD08] Jens von Pilgrim und Kristian Duske. „Gef3D: a framework for two-, two-and-a-half-, and three-dimensional graphical editors“. In: *Proceedings of the 4th ACM symposium on Software visualization*. SoftVis ’08. Ammersee, Germany: ACM, 2008, 95–104. ISBN: 978-1-60558-112-5. DOI: [10.1145/1409720.1409737](https://doi.org/10.1145/1409720.1409737).
- [Pho75] Bui Tuong Phong. „Illumination for computer generated pictures“. In: *Commun. ACM* 18.6 (Juni 1975), 311–317. ISSN: 0001-0782. DOI: [10.1145/360825.360839](https://doi.org/10.1145/360825.360839).
- [Por04] Nick Porcino. „Gaming Graphics: The Road to Revolution“. In: *Queue* 2.2 (Apr. 2004), 62–71. ISSN: 1542-7730. DOI: [10.1145/988392.988409](https://doi.org/10.1145/988392.988409).
- [RCL05] Nicolas Ray, Xavier Cavin und Bruno Lévy. *Vector texture maps on the GPU*. Techn. Ber. 2005.

- [Rot11] Bastian Roth. *Konzeption und Implementierung eines generischen Modellierungswerkzeugs zur Unterstützung der domänenspezifischen Prozessmodellierung*. Master-Thesis. Universität Bayreuth. 2011.
- [SA94] Lindsey L. Spratt und Allen L. Ambler. „Using 3D Tubes to Solve the Intersecting Line Representation Problem.“ In: VL'94. 1994, S. 254–261.
- [SBE00] Bastiaan Schönhage, Alex van Ballegooij und Anton Elléns. „3D gadgets for business process visualization – a case study“. In: *Proceedings of the fifth symposium on Virtual reality modeling language (Web3D-VRML)*. VRML '00. Monterey, California, United States: ACM, 2000, 131–138. ISBN: 1-58113-211-5. DOI: [10.1145/330160.330238](https://doi.org/10.1145/330160.330238).
- [Sch10] LLC Schrödinger. „The PyMOL Molecular Graphics System, Version 1.3r1“. Aug. 2010.
- [Sei03] Ed Seidewitz. „What Models Mean“. In: IEEE Softw. 20.5 (Sep. 2003), 26–32. ISSN: 0740-7459. DOI: [10.1109/MS.2003.1231147](https://doi.org/10.1109/MS.2003.1231147).
- [SN00] August-Wilhelm Scheer und Markus Nüttgens. „ARIS Architecture and Reference Models for Business Process Management“. In: *in: Business Process Management – Models, Techniques, and Empirical Studies*. Springer, 2000, 376–389.
- [Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer Vienna, 1973. ISBN: 978-3211811061.
- [Sun11] Kelvin Sung. „Recent Videogame Console Technologies“. In: Computer 44.2 (Feb. 2011), 91–93. ISSN: 0018-9162. DOI: [10.1109/MC.2011.55](https://doi.org/10.1109/MC.2011.55).
- [TC09] Alfredo R. Teyseyre und Marcelo R. Campo. „An Overview of 3D Software Visualization“. In: IEEE Transactions on Visualization and Computer Graphics 15.1 (Jan. 2009), 87–105. ISSN: 1077-2626. DOI: [10.1109/TVCG.2008.86](https://doi.org/10.1109/TVCG.2008.86).
- [Vol11] B. Volz. „Werkzeugunterstützung für Methodenneutrale Metamodellierung“. Diss. Universität Bayreuth, Juli 2011.
- [War04] Colin Ware. *Information Visualization: Perception for Design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN: 1558608192.
- [WF96] Colin Ware und Glenn Franck. „Evaluating stereo and motion cues for visualizing information nets in three dimensions“. In: ACM Trans. Graph. 15.2 (Apr. 1996), 121–140. ISSN: 0730-0301. DOI: [10.1145/234972.234975](https://doi.org/10.1145/234972.234975).
- [WL12] Dennis Wiebusch und Marc Erich Latoschik. „Enhanced Decoupling of Components in Intelligent Realtime Interactive Systems using Ontologies“. In: *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), proceedings of the IEEE Virtual Reality 2012 workshop*. 2012.
- [WM08] Colin Ware und Peter Mitchell. „Visualizing graphs in three dimensions“. In: ACM Trans. Appl. Percept. 5.1 (Jan. 2008), 2:1–2:15. ISSN: 1544-3558. DOI: [10.1145/1279640.1279642](https://doi.org/10.1145/1279640.1279642).
- [Wri+10] Richard S. Wright u.a. *OpenGL SuperBible: Comprehensive Tutorial and Reference*. 5th. Addison-Wesley Professional, 2010. ISBN: 0321712617, 9780321712615.
- [WS08] Ingo Weisemöller und Andy Schürr. „A Comparison of Standard Compliant Ways to Define Domain Specific Languages“. In: *Models in Software Engineering*. Hrsg. von Holger Giese. Bd. 5002. LNCS. Heidelberg: Springer, 2008, 47–58. ISBN: 978-3-540-69069-6. DOI: [10.1007/978-3-540-69073-3_6](https://doi.org/10.1007/978-3-540-69073-3_6).
- [WTS89] Christopher D. Wickens, Steven Todd und Karen Seidler. *Three dimensional displays: Perception, Implementation, Applications*. 1989. URL: <http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA259937>.

WWW-Referenzen

- [wp:bewegungsparallaxe] *Bewegungsparallaxe – Wikipedia*. Version vom 7. April 2012 um 04:57. URL: <http://de.wikipedia.org/w/index.php?title=Bewegungsparallaxe&oldid=101762882>.
- [wpe:cave] *Cave automatic virtual environment – Wikipedia*. Version vom 6 April 2012 um 8:14. URL: http://en.wikipedia.org/w/index.php?title=Cave_automatic_virtual_environment&oldid=485859676.
- [wp:stereoskopie] *Stereoskopie – Wikipedia*. Version vom 13 März 2012 um 3:39. URL: <http://de.wikipedia.org/w/index.php?title=Stereoskopie&oldid=100809900>.
- [www:blender] *Blender*. URL: <http://www.blender.org/> (besucht am 11.04.2012).
- [www:dia] *Dia*: URL: <http://live.gnome.org/Dia> (besucht am 07.04.2012).
- [www:emf] *Eclipse Modeling Framework Project (EMF)*. URL: <http://www.eclipse.org/modeling/emf/> (besucht am 07.04.2012).
- [www:erlang] *Erlang Programming Language*. URL: <http://www.erlang.org/> (besucht am 07.04.2012).
- [www:frustum] *glFrustum*. URL: <http://www.opengl.org/sdk/docs/man/xhtml/glFrustum.xml> (besucht am 07.04.2012).
- [www:gef3d] *GEF3D*. URL: <http://www.eclipse.org/gef3d> (besucht am 07.04.2012).
- [www:gef3ddevblog] *GEF3D Development Blog*. URL: <http://gef3d.blogspot.com> (besucht am 07.04.2012).
- [www:glpipe] *Rendering Pipeline – OpenGL Wiki*. Version vom 6 November 2011 um 18:12. URL: http://www.opengl.org/wiki_132/index.php?title=Rendering_Pipeline_Overview&oldid=4107.
- [www:lwjgl] *LWJGL*. URL: <http://lwjgl.org/> (besucht am 07.04.2012).
- [www:quat] *Quaternions*. URL: <http://mathworld.wolfram.com/Quaternion.html> (besucht am 07.04.2012).
- [www:scala] *The Scala Programming Language*. URL: <http://www.scala-lang.org/> (besucht am 07.04.2012).
- [www:simplex3d] *Simplex3D*. URL: <http://www.simplex3d.org/> (besucht am 07.04.2012).
- [www:visio] *Microsoft Visio*. URL: <http://office.microsoft.com/de-de/visio/> (besucht am 07.04.2012).
- [www:vislib] *Visualization Library*. URL: <http://www.visualizationlibrary.org> (besucht am 07.04.2012).
- [www:xtext] *Xtext*. URL: <http://www.eclipse.org/Xtext/> (besucht am 07.04.2012).

Verwendete Metamodelle

A.1 Editor-Metamodell

A.1.1 Programming-Language-Mapping

```
level D3 {
    package base {
        concept ScalaMapping {
            1..1 string scalaType;
            0..1 string typeConverter;
        }
    }
}
```

A.1.2 Editor-Base-Level

```
level D2 instanceOf EMM.D3 {
    package types {
        import EMM.D3.base.ScalaMapping;
        ScalaMapping Dimension {
            1..1 real x;
            1..1 real y;
            1..1 real z;
            scalaType = "newsimplex3d.math.float.Vec3";
            typeConverter = "mmpe.model.lmm2scala.Simplex3DVectorConverter";
        }

        ScalaMapping Position {
            1..1 real x;
            1..1 real y;
            1..1 real z;
            scalaType = "newsimplex3d.math.float.Vec3";
            typeConverter = "mmpe.model.lmm2scala.Simplex3DVectorConverter";
        }

        ScalaMapping Rotation {
```

```

    1..1 real x0;
    1..1 real x1;
    1..1 real x2;
    1..1 real x3;
    scalaType = "newsimplex3d.math.float.Quat4";
    typeConverter = "mmpe.model.lmm2scala.Simplex3DVectorConverter";
}

ScalaMapping Color {
    1..1 real r;
    1..1 real g;
    1..1 real b;
    1..1 real a;
    scalaType = "java.awt.Color";
    typeConverter = "mmpe.model.lmm2scala.AWTTypeConverter";
}

ScalaMapping Font {
    0..1 string face;
    0..1 integer size;
    0..1 string style;
    scalaType = "java.awt.Font";
    typeConverter = "mmpe.model.lmm2scala.AWTTypeConverter";
}

abstract ScalaMapping PhysicsSettings {
    1..1 real mass;
    1..1 real restitution;
    typeConverter = "mmpe.model.lmm2scala.PhysicsSettingsConverter";
}

concept PhysBox extends PhysicsSettings {
    1..1 concept Dimension halfExtends;
    scalaType = "siris.components.physics.PhysBox";
}

concept PhysSphere extends PhysicsSettings {
    1..1 real radius;
    scalaType = "siris.components.physics.PhysSphere";
}

package figures {
    import EMM.D3.base.ScalaMapping;
    import EMM.D2.types.*;
    abstract ScalaMapping EditorElement {
        1..1 pointer modelElementFQN;
        string toolingIcon = "(none)";
        1..1 boolean interactionAllowed = true;
        0..1 real highlightFactor = 1.3;
    }

    abstract ScalaMapping SceneryObject {
        1..1 string toolingName;
        1..1 string toolingIcon;
        1..1 concept Position pos;
    }
}

```

```

    1..1 concept Dimension dim;
    1..1 concept Rotation rotation;
    0..1 concept PhysicsSettings physics;
}

abstract concept Node extends EditorElement {
    1..1 string toolingAttrib;
    1..1 string toolingTitle;
    1..1 concept Position pos;
    1..1 concept Dimension dim;
    1..1 concept Rotation rotation;
}

abstract concept Edge extends EditorElement {
    1..1 real thickness;
    1..1 string inboundAttrib;
    1..1 string outboundAttrib;
    1..1 string toolingName;
    1..1 concept Node startNode;
    1..1 concept Node endNode;
}

concept ColoredLine extends Edge {
    1..1 concept Color color;
    1..1 concept Color specularColor;
    scalaType = "mmpe.model.figures.ColoredLine";
}

concept TexturedLine extends Edge {
    1..1 string texture;
    1..1 concept Color specularColor;
    1..1 concept Color backgroundColor;
    scalaType = "mmpe.model.figures.TexturedLine";
}

abstract concept TextLabelNode extends Node {
    1..1 string displayAttrib;
    1..1 concept Font font;
    1..1 concept Color fontColor;
    1..1 concept Color backgroundColor;
}

abstract concept TexturedNode extends Node {
    1..1 string texture;
    1..1 concept Color backgroundColor;
}

concept TextDiamond extends TextLabelNode {
    scalaType = "mmpe.model.figures.TextDiamond";
}

concept RoundedTextBox extends TextLabelNode {
    scalaType = "mmpe.model.figures.RoundedTextBox";
}

```

```

concept TextBox extends TextLabelNode {
    scalaType = "mmpe.model.figures.TextBox";
}

concept TextureBox extends TexturedNode {
    scalaType = "mmpe.model.figures.TextureBox";
}

concept TextureDiamond extends TexturedNode {
    scalaType = "mmpe.model.figures.TextureDiamond";
}

concept ColladaSceneryObject extends SceneryObject {
    1..1 string sceneryModel;
    scalaType = "mmpe.collada";
}
}

}

```

A.1.3 Editor-Definition-Level (Auszug)

```

Level D1 instanceOf EMM.D2 {
    package nodeFigures {
        concept ProcessNode instanceOf TextBox {
            modelElementFQN = pointer PM.M2.processLanguage.Process;
            displayAttrib = "function";
            toolingAttrib = "function";
            toolingTitle = "Process";
            font = DefaultFont;
            fontColor = Black;
            backgroundColor = LightYellow;
            dim = UnitDimension;
            pos = DefaultPosition;
            rotation = NullRotation;
        }
    }

    concept AndConnectorNode instanceOf TextDiamond {
        texture = "tex/model_textures/and_symbol.png";
        modelElementFQN = pointer PM.M2.processLanguage.AndConnector;
        toolingAttrib = "name";
        toolingTitle = "AND";
        backgroundColor = Orange;
        dim = UnitDimension;
        pos = DefaultPosition;
        rotation = DiamondRot;
    }
}

package connectionFigures {
    concept ControlFlowEdge instanceOf TexturedLine {
        toolingName = "Control Flow";
        outboundAttrib = "outboundControlFlows";
        inboundAttrib = "inboundControlFlows";
        modelElementFQN = pointer PM.M2.processLanguage.ControlFlow;
        texture = "tex/model_textures/triangle_half_cyan.png";
    }
}

```

```

        specularColor = White;
        backgroundColor = Red;
        thickness = 0.1;
        highlightFactor = 1.7;
    }

concept NodeDataEdge instanceOf ColoredLine {
    toolingName = "Functional-Data Assoc";
    outboundAttrib = "outboundNodeDataConnection";
    inboundAttrib = "inboundNodeDataConnection";
    modelElementFQN = pointer PM.M2.processLanguage.NodeDataConnection;
    color = Blue;
    specularColor = White;
    thickness = 0.03;
    highlightFactor = 2.0;
}
}
}

```

A.2 Prozess-Metamodell

```

model PMM {
    uri "model:/www.ai4.uni-bayreuth.de/ipmap3d/pmm";
    level M2 {
        package processLanguage {
            abstract concept Perspective {
            }

            abstract concept Connection {
            }

            abstract concept DataPerspective extends Perspective {
            }

            abstract concept FunctionalPerspective extends Perspective {
            }

            concept ControlFlow extends Perspective {
                string tag = "(noTag)";
            }

            concept DataFlow extends DataPerspective {
            }

            concept NodeDataConnection extends Connection {
            }

            concept NodeControlFlowLabelConnection extends Connection {
            }

            concept ProcessOrgConnection extends Connection {
            }
        }
    }
}

```

```
concept Node extends FunctionalPerspective {
    0..* concept ControlFlow inboundControlFlows;
    0..* concept ControlFlow outboundControlFlows;
    0..* concept NodeDataConnection outboundNodeDataConnection;
    0..* concept NodeControlFlowLabelConnection outboundNodeControlFlowLabelConnection;
}

concept DataItem extends DataPerspective {
    1..1 string name;
    0..* concept DataFlow inboundDataFlows;
    0..* concept DataFlow outboundDataFlows;
    0..* concept NodeDataConnection inboundNodeDataConnection;
}

concept ControlFlowLabel {
    0..* concept NodeControlFlowLabelConnection inboundNodeControlFlowLabelConnection;
    1..1 string tag;
}

concept DataContainer extends Node extends DataPerspective {
    string name;
}

concept Process extends Node {
    string function;
    string shortFunction;
    real duration;
    integer difficulty;
    boolean isComposite;
}

abstract concept FlowElement extends Node {
    string name;
}

concept StartStopInterface extends FlowElement {}

concept AndConnector extends FlowElement {}

concept OrConnector extends FlowElement {}

concept Decision extends FlowElement {
}

}
```

Systemanforderungen und Inhalt der DVD

B.1 Systemanforderungen von i>PM3D

B.1.1 Hardware

- Grafikchip mit OpenGL 3.3-Unterstützung. Getestet mit Geforce 8400, 9400GS, AMD A-3850.
- mindestens 1,5 GByte freier Arbeitsspeicher
- 200MB Festplattenspeicher
- Minimale Auflösung 1024x768
- Optional: Microsoft Kinect, Nintendo WiiMote

B.1.2 Software

- Betriebssystem Windows (getestet unter Windows 7) oder Linux (getestet unter Ubuntu Linux 11.10 und Gentoo Linux)
- Grafiktreiber mit Unterstützung für OpenGL 3.3
- Java Virtual Machine in einer Version ab 1.6
- (Scala und weitere Libraries sind in die ausführbare jar der Anwendung integriert)

B.2 DVD

B.2.1 Ausführbare Dateien

Die DVD enthält eine ausführbare Version von i>PM3D. Für den Start der Anwendung werden .bat-Dateien (Windows) und sh-Dateien (Linux) für unterschiedliche Konfigurationen der Eingabegeräte angeboten. Weitere Hinweise zur Ausführung des Programms werden auf der DVD in README.txt gegeben. Außerdem ist der Collada2Scala-Compiler enthalten. Hinweise dazu befinden sich in README-collada2scala.txt.

B.2.2 Eclipse-Projekte

Die DVD umfasst den vollständigen Quellcode von i>PM3D.

Die mitgelieferten Scala-Eclipse-Projekte teilen sich wie folgt auf:

- MMPEEditor: Editorkomponente, Modellkomponente / Modellanbindung
- MMPPERenderer: Render-Bibliothek, Collada-Loader, Collada2Scala
- MMPESirisAddons: Benötigte Erweiterungen für Simulator X: Renderkomponente, modifizierte Physikkomponente
- MMPEUtils: Verschiedene Scala-Hilfsklassen und -objekte (Logging, Mathematik)
- MMPEResources: verwendete Texturen, Shader und Collada-Dateien sowie Modelle
- LMM4MMPE: LMMLight-Implementierung, Parser, M2T
- ScalaST4: Anbindung von StringTemplate

(Rückfragen zur Ausführung des Programms oder zur Verwendung und Kompilierung der Projekte in Eclipse oder mit SBT bitte an Tobi.Stenzel@gmx.de¹)

Weitere Hinweise werden auf der DVD in README.txt gegeben.

B.2.3 Beilagen

- Video BPMN-Prozess am Flughafen: brown-airport.flv
- PDF (Anzeigeversion) der Bachelorarbeit: IPM3D-Tobias-Stenzel.pdf
- PDF (Druckversion): IPM3D-Tobias-Stenzel-print.pdf
- In der Arbeit gezeigte Screenshots von i>PM3D: screenshots
- Diagramme (.dia-Format): diagrams
- In der Arbeit gezeigte Abbildungen: ext-pics

¹Tobi.Stenzel@gmx.de