Brandenburgische Technische Universität Cottbus
Fakultät für Mathematik, Naturwissenschaften und Informatik
Institut für Informatik, Informations- und Medientechnik
Lehrstuhl Praktische Informatik / Grafische Systeme

Konzeption eines Subsystems für die GroIMP-Plattform sowie eines zugrunde liegenden XML-Datenformats zum Austausch graphbasierter, multiskalierter Strukturen

Diplomarbeit von Sören Schneider

Erstgutachter und Betreuer:	Prof. Dr. Winfried Kurth
Zweitgutachter:	Prof. Dr. Gerd Wagner
Betreuer:	Dipl. Phys. Ole Kniemeyer

Cottbus, 30. Oktober 2006

Eidesstattliche Erklärung

Die vorliegende Diplomarbeit wurde von mir selbstständig verfasst. Die verwendeten Quellen und Hilfsmittel sind im Literaturverzeichnis vollständig angegeben.

Cottbus, 30. Oktober 2006

Sören Schneider

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Bei der Zusammenarbeit von Wissenschaftlern an gemeinsamen Projekten spielt der Austausch von Daten eine wichtige Rolle. Dabei gibt es oft Probleme, weil die verschiedenen Arbeitsgruppen aufgrund unterschiedlicher fachlicher Vokabulare, unterschiedlicher verwendeter Modelle und nicht zuletzt unterschiedlicher verwendeter Software kein gemeinsames Datenaustauschformat besitzen.

Um die unterschiedlichen Modelle, die auf ein Objekt anwendbar sind, zu beschreiben, reicht ein einzelner Graph nicht aus. Um dies zu verdeutlichen, wird im Kapitel 2 ein ausführliches Beispiel (siehe Bsp. 2.1) vorgestellt, das die Verschiedenartigkeit von Modellen am Beispiel von Pflanzen zeigen soll. Das darin enthaltene beispielhafte Modell eines Baumes mit fünf unterschiedlichen Skalen wird die gesamte Arbeit begleiten. Das Modell ist bewusst einfach gehalten und zielt nicht auf botanische Genauigkeit. Die Anzahl der Knoten wurde möglichst gering gehalten, um die Übersichtlichkeit zu wahren.

Die Beschreibung eines Objektes innerhalb einer gewählten Skale erfolgt anhand von Modulen, die logischen Einheiten eines fachspezifischen Modells entsprechen. Diesen Modulen sollten sowohl die Attribute des beschreibenden Modells, als auch andersartige Attribute zugeordnet werden können. Denn die Zusammenarbeit zwischen den Anwendern der verschiedenen Modelle kann durch die Verwendung von Datenformaten eingeschränkt sein, die nur die Erfassung modellspezifischer Parameter zulassen.

Ein Format, welches die Speicherung eines Objektes in verschiedenen Modellen zulässt und außerdem die Möglichkeit bietet, modellspezifische, domänenbasierte Parameter zu definieren, kann neue Perspektiven und das Erkennen von Zusammenhängen zwischen Modellen ermöglichen.

Dabei soll das Format lediglich die Möglichkeit zur Erfassung von Objekten auf verschiedenen Modellebenen bieten. Zusammenhänge zwischen den Modellen/Skalen zu modellieren bleibt Aufgabe des Benutzers bzw. der Software, welche das Format verwenden.

Deshalb soll das Konzept der multiskalierten Beschreibung eines Objektes auf Anwendbarkeit untersucht werden. So soll jedes Modell als eigenständige Skale eines Objektes beschrieben werden. Die Vielzahl an Beschreibungsmöglichkeiten auf verschiedenen Ebenen führt zu der Notwendigkeit der Entwicklung eines multiskalierten Formates zur Beschreibung eines Objektes. Es gibt in verschiedenen Bereichen bereits Anwendungen solcher Multiskalierungskonzepte (z. B. multiskalierte Beschreibungen von Objekten in der Computergrafik durch unterschiedliche Detaillierungsgrade (siehe Abschnitt 2.1.2) und multiskalierte Modellierung und Simulation in der Physik). Es muss also untersucht werden, ob bereits Datenformate existieren, die multiskalierte Strukturen beschreiben können und im Rahmen der Aufgabenstellung erweitert werden können.

Die Erfassung der Grundstruktur von Objekten soll auf der Basis eines sie beschreibenden Graphen erfolgen. Da die Definition einer vorgegebenen Menge an Attributen zur Beschreibung der Eigenschaften der Objekte weder alle Bereiche der Wissenschaft abdecken kann, noch den notwendigen Freiraum lässt, um flexibel auf die Anforderungen neuer Entwicklungen einzugehen, soll die Definition der verwendeten Attribute den Anwendern überlassen werden. Der zur Beschreibung der Objekte von den Arbeitsgruppen verwendete Satz von Attributen soll ihnen zugeordnet werden können und von ihnen selbst dokumentiert werden. Arbeitsgruppen, die an gemeinsamen Projekten arbeiten, sollen sich anhand der öffentlich verfügbaren Dokumentation einen Überblick über die verwendeten Attribute der anderen Seite verschaffen können, damit die Konvertierung der fremden Attribute in die eigenen Attribute erleichtert wird.

XML[1] gilt als Standard bei der Definition offener Dateiformate. Eine Vielzahl an existierenden Werkzeugen zur Modellierung, Validierung und Verarbeitung XML-basierter Daten bietet umfangreiche Möglichkeiten zur Handhabung von Daten. Diese bilden die Grundlage dieser Arbeit und werden in Abschnitt 3.3 näher erläutert.

Dabei bietet zum Beispiel XSLT[2] umfangreiche Möglichkeiten zur Umwandlung von XML-Daten in andere Formate. Damit können also beispielsweise die verschiedenen Attribute konvertiert werden.

Namensräume[3] sind geeignet, die Grundlage eines Konzeptes der Kennzeichnung der Zugehörigkeit von Attributen zu bilden.

Ziel der Untersuchung wird es sein, ob mithilfe von XML multiskalierte Strukturen erfasst werden können, und gegebenenfalls ein solches XML-Format zu entwickeln.

---

[1] eXtensible Markup Language

[2] eXtensible Stylesheet Language Transformations: XML-basierte Sprache zur Transformation von XML-Dokumenten (siehe Abschnitt 3.3.4)

[3] siehe Abschnitt 3.3.3

1.1 Begriffsklärung

1.1.1 multiskaliert

Ein *multiskaliertes* Objekt (Def. 2.1) besitzt mehrere *Skalen*. Das Konzept der Multiskalierung dient der Beschreibung eines Objektes durch verschiedene Modelle. Eine ausführliche Erklärung erfolgt in Kapitel 2.

1.1.2 graphbasiert

Eine *graphbasierte* Objektbeschreibung beschreibt die Struktur eines Objektes mithilfe eines gerichteten, schleifenfreien Graphen mit Mehrfachkanten. Ein Graph besteht aus einer Menge von Knoten und Kanten. Dabei beschreiben die Knoten die Teilelemente des Objekts und die Kanten die Beziehungen zwischen den Knoten. Es können sowohl gerichtete als auch ungerichtete Kanten vorkommen. Dabei sind Schleifen innerhalb des Graphen ausdrücklich nicht erlaubt. Mehrfachkanten sind hingegen erlaubt. Insbesondere können ungerichtet Kanten also durch zwei gerichtete Kanten ausgedrückt werden.

1.1.3 Skale

Eine *Skale* (Def. 2.6) ist eine Beschreibung eines Objektes anhand modellspezifischer *Module* und ihrer Beziehungen (siehe Abschnitt 2.2) zueinander. Eine Skale kann eine *Verfeinerung* (Def. 2.8) einer anderen Skale darstellen, wenn die Elemente der verfeinerten Skale durch Elemente der verfeinernden Skale beschrieben werden können. Gilt das nicht, *überlagern* (Def. 2.9) sich die Skalen. Ein Modul (Def. 2.7) ist innerhalb einer Skale eine nicht weiter zerlegbare Einheit, die ein Bestandteil des Modells des Objektes ist. Je zwei Module innerhalb einer Skale können durch verschiedene Relationen (siehe Abschnitt 2.2) miteinander in Beziehung stehen.

1.2 Allgemeine Problemstellung und Motivation

Ziel dieser Diplomarbeit ist es, die am Lehrstuhl Grafische Systeme der BTU Cottbus in Entwicklung befindliche 3D-Plattform GroIMP[4] um ein Subsystem zu erweitern, das dem bidirektionalen Datenaustausch zu existierenden Datenformaten dreidimensionaler, multiskalierter Strukturen, sowie allgemeiner graphbasierter Strukturen dient. Die Basis dieses Subsystems soll ein geeignetes, erweiterbares

[4]GroIMP [Kni06] (=Growth Grammar-related Interactive Modelling Platform) ist eine 3D-Modellierungsplattform, welche sich durch die Möglichkeit auszeichnet, Modelle mithilfe von Wachstumsgrammatiken zu erstellen. Dazu steht mit XL [Kni04] eine eigene Modellierungssprache zur Verfügung.

XML-Datenformat sein. Dieses Dateiformat soll den einheitlichen, Fachdisziplinen übergreifenden Austausch struktureller Daten zwischen Anwendungen ermöglichen.

Die Motivation für die Aufgabenstellung ergibt sich aus der Nützlichkeit für wissenschaftliche Belange in der Pflanzenmodellierung und computergrafische Belange der 3D-Modellierung. Die Wissenschaftler, welche die dreidimensionale Struktur von Vegetation untersuchen (in Biologie, Ackerbau, Forstwirtschaft, ...), haben in den letzten Jahren mittels manueller sowie digitaler, 3D-Scanner-gestützter Messungen große Datenbestände angesammelt. Diese liegen aber in zahlreichen unterschiedlichen Formaten und Datenbanken vor. Ein Vorschlag für ein einheitliches Datenformat, unter Verwendung des XML-Standards, würde den Austausch dieser Daten und die Evaluierung von Pflanzensimulationsmodellen sehr fördern. Ebenso haben sich im Bereich der allgemeinen 3D-Modellierung zahlreiche unterschiedliche Formate etabliert. Es besteht jedoch ein Mangel an brauchbaren Werkzeugen zum Austausch zwischen den verschiedenen Formaten. Da GroIMP mit dem Ziel einer universellen Modellierplattform entwickelt wurde und wird, gibt es einen Bedarf für ein flexibles Subsystem zum bidirektionalen Datenaustausch solcher Modelle.

Kapitel 2

Multiskalierung

2.1 Was ist ein multiskaliertes Objekt?

Ein Objekt kann oftmals in verschiedene Teilobjekte zerlegt werden. Dabei unterliegt die Wahrnehmung eines Objektes dem subjektiven Eindruck des Betrachters. So gibt es zahlreiche Faktoren, welche die Zerlegung eines Objektes in Teilobjekte beeinflussen. Das Wissen des Betrachters über die Struktur des Objektes hat dabei Einfluss auf die Zerlegung des Objektes in verschiedene Module (Def. 2.7) und Skalen (Def. 2.6). Weiterhin ist die Wahrnehmung eines Objektes und seiner Bestandteile abhängig von der Beobachtungsentfernung. Die Trennungsmerkmale zwischen den Teilobjekten verschwinden und die Bestandteile eines Objektes scheinen bei zunehmender Entfernung und der damit verbundenen Auflösungsverschlechterung zu verschmelzen.

Definition 2.1 *Ein* ***multiskaliertes Objekt*** *ist ein Objekt, welches durch verschiedene Modelle beschrieben wird. Dabei entspricht jede* Skale *einem Modell.*

Ein Objekt kann Multiskalierung auf verschiedenen Ebenen aufweisen. Die folgenden drei Unterabschnitte erläutern die unterschiedlichen Formen von Skalierung, die einem Objekt zugeordnet werden können.

2.1.1 Skalierung auf Modellebene

Ein Objekt kann abhängig vom wissenschaftlichen Fachgebiet, von dem aus es betrachtet wird, durch verschiedene Modelle beschrieben werden. Physikalische, biologische, chemische oder mathematische Beschreibungen des gleichen Objektes erfolgen anhand unterschiedlichster Modelle. Aber auch innerhalb der Fachgebiete gibt es verschiedene Modelle zur Beschreibung von Objekten.

Beispiel 2.1 *So gibt es nach [GC98] verschiedene Strukturen einer Pflanze, die mit zahlreichen Modellen beschrieben werden können. Im Folgenden werden einige davon aufgeführt:*

1. räumliche Struktur*: Verteilung der Pflanzenbestandteile im 3D-Raum mit Orts- und Richtungsangaben für jeden Bestandteil,*
2. geometrische Struktur*: Formen der Pflanzenbestandteile,*
3. mechanische Struktur*: Gültigkeit von mechanischen Bedingungen*[1] *der Pflanzenbestandteile,*
4. hydraulische Struktur*: Gefäßsystem und Flüssigkeitshaushalt der Pflanzen,*
5. topologische Struktur*: Zerlegung der Pflanze in elementare Bestandteile und Beschreibung der Zusammenhangsbeziehungen der Bestandteile*

Einige der darstellbaren Modelle zur Abbildung der Struktur oder der Funktion einer Pflanze betreffen zum Beispiel die Kohlenstoffverteilung, den Wasserhaushalt, das Wurzelwachstum, den architektonischen Aufbau, die Interaktion mit der Mikroumgebung oder die mechanischen Eigenschaften. So gibt es nach [God00] zwei Sorten von Pflanzendarstellungen:

- ***globale Darstellungen****: Sie beschreiben eine Pflanze oder deren Funktionen im Ganzen. Zwei Arten von globalen Darstellungen seien genannt:*
  - geometrische Darstellungen*: Eine einfache Art der Repräsentation von Pflanzengeometrien kann durch parametrische Darstellungen erreicht werden. Kugeln oder Ellipsen werden zum Beispiel benutzt, um das Abfangen von Licht durch Baumkronen zu modellieren. Zylinder, Kegel, Kegelstümpfe oder Paraboloide werden benutzt, um die mechanischen Eigenschaften oder, speziell in forstwirtschaftlichen Anwendungen, um Wurzeln oder Baumkronen von Pflanzen zu modellieren. Um ein breiteres Spektrum von Formen zu erhalten, müssen diesen einfachen geometrischen Grundformen nur zusätzliche Parameter hinzugefügt werden.*
  - kompartimentbasierte Darstellung*: Kompartiment*[2]*-basierte Ansätze dienen dazu, den Austausch von Substanzen innerhalb von Pflanzen zu modellieren. Die Pflanzen werden dabei in zwei oder mehr Kompartimente zerlegt, die Quellen und Senken beim Transfer von Substanzen innerhalb der Pflanze oder an der Schnittstelle zwischen der Pflanze und ihrer Umgebung darstellen. Kompartimentbasierte Darstellungen können*

[1] engl. constraints
[2] engl. compartment

als topologische Beschreibung von Pflanzenarchitekturen betrachtet werden. Ein Kompartiment kann zum Beispiel eine Menge an Blättern, Wurzeln, Früchten oder Hölzern darstellen, die miteinander verbunden sind. In diesen Mengen wird zwischen den einzelnen Elementen nicht mehr unterschieden. Sie werden als Biomasse angesehen mit bestimmten globalen Eigenschaften (z. B. fotosynthetische Effizienz, Masse, Temperatur, Übertragungsraten, usw.). Gleichermaßen wird vorgegangen, um den Transport von Wasser in Pflanzen zu modellieren. Die Pflanze wird dann als eine Reihe von Kompartimenten an der Schnittstelle zwischen dem Boden und der Luft dargestellt. Jedes Kompartiment hat einen spezifischen hydraulischen Leitwert. Der Fluss des Wassers durch die Pflanze ergibt sich aus den unterschiedlichen Wasserpotenzialen zwischen der Oberfläche der Blätter und den Wurzeln.

- ***modulare Darstellung**: Pflanzen bestehen aus Wiederholungen bestimmter Typen von Komponenten. Modulare Darstellungen bestehen aus der Beschreibung dieser wiederholten Komponenten. Zwei grundlegende Zerlegungstypen von Pflanzenarchitekturen in Module sind*

  - *die* räumliche *und*
  - *die* organbasierte *Zerlegung.*

  Bei räumlichen Zerlegungen *wird die Verteilung von Pflanzenmodulen im dreidimensionalen Raum angenähert durch die Aufteilung des dreidimensionalen Raumes in Zellen mit einfachen aber gleichen Formen und die Markierung der Zellen, die Pflanzenmodule enthalten.*

  Die organbasierten Zerlegungen *von Pflanzenmodellen können in zwei Klassen eingeteilt werden. In*

  - geometrische Zerlegungen*, wobei nur die geometrischen Aspekte der Pflanzenorgane und ihre räumlichen Positionen benutzt werden, und*
  - topologische Darstellungen*, bei denen die Verbindungen zwischen den Pflanzenorganen betrachtet werden.*

2.1.2 Skalierung bei der grafischen Darstellung

Ein anderer Einsatzbereich von Skalen bietet sich in der grafischen Darstellung von Objekten. Der Detaillierungsgrad (engl. Level of Detail [Con04a][Wat01]) eines Objektes variiert in Abhängigkeit von der Entfernung zum Betrachter.

Für diesen Zweck können für ein Objekt mehrere Skalierungen angegeben werden, die von einer detailreichen Darstellung des Objektes aus vielen grafischen

Primitivobjekten[3] bis hin zu einer extrem vereinfachten Darstellung aus nur wenigen oder gar einem grafischen Primitivobjekt reichen.

2.1.3 Skalierung bei der zeitlichen Entwicklung

Ein Objekt kann im Laufe seiner Existenz verschiedene Entwicklungsstadien durchlaufen, die mit Form-, Farb- oder andersartigen Veränderungen einhergehen können. Auch diese Effekte können durch eine Modellierung eines Objektes in verschiedenen Skalen zum Ausdruck gebracht werden.

Beispiel 2.2 *Die Frucht eines Baumes kann in einer Skale im Stadium der Blüte, in einer anderen Skale als ausgewachsener Fruchtkörper modelliert werden.*

2.2 Multiskaliertes Modell zur Pflanzenbeschreibung

In [GC98] wird ein multiskalierter Graph zur Modellierung der statischen, topologischen Struktur von Pflanzen vorgestellt. Dieses Prinzip und eine Auswahl der dortigen Definitionen wird in diesem Abschnitt, verkürzt und in deutsche Sprache übersetzt, wiedergegeben und dient gleichzeitig als Einführung in das Thema Multiskalierung.

Dieses Modell zur Darstellung statischer, pflanzlicher, topologischer Strukturen benutzt Baumgraphen, um topologische Strukturen innerhalb einer gegebenen Modularität zu modellieren. Das Modell wird dann mehr und mehr verallgemeinert, um Aufteilungsgraphen, multiskalierte Baumgraphen und schließlich verallgemeinerte multiskalierte Baumgraphen abbilden zu können. In einem weiteren Schritt wird noch ein Verfahren zur Beschreibung dynamischer, multiskalierter Baumgraphen vorgestellt. Dies ist aber für die Aufgabenstellung nicht relevant und wird nicht betrachtet.

[3] zwei- oder dreidimensionale geometrische Grundformen, aus denen sich komplexere Formen zusammensetzen lassen, wie z. B. Kugel, Zylinder, Kegel, Dreieck, Box, ...

Definition 2.2 *Nach [GC98](Def.1) ist ein* ***Graph*** *h ein Tripel (V, E, <.>), wobei*

- *V eine endliche Menge von Knoten ist, bezeichnet*[4] *mit* $\vartheta(h)$,
- *E eine endliche Menge von Kanten ist, bezeichnet mit* $\varepsilon(h)$, *und*
- *<.> eine Abbildung von E nach* $V \times V$ *ist, bezeichnet mit* $<.>_h$. *Diese auch als* Inzidenzfunktion *von h bezeichnete Abbildung besitzt zwei Teilfunktionen, bezeichnet mit <.| und |.>. Beispiel: <e>=(x, y), <e| = x und |e> = y, d. h. Kante e ist inzident zu den Knoten x und y. Dabei bezeichnet man x als Ausgangspunkt und y als Endpunkt der Kante e.*

Definition 2.3 *Ein* ***Wurzelknoten*** *ist ein Knoten, der keine eingehenden Kanten besitzt.*

Definition 2.4 *Ein* ***Baumgraph***[5] *ist ein endlicher, zyklenfreier, zusammenhängender und gerichteter Graph, der genau einen Wurzelknoten besitzt. Alle anderen Knoten eines Baumgraphen besitzen genau eine eingehende Kante.*

Die durch einen Baumgraphen beschriebene topologische Struktur entspricht einer ausgewählten Modularität einer Pflanze.

Pflanzen sind modulare Organismen. Sie können in Mengen von Bestandteilen mit ähnlichen Merkmalen zerlegt werden.

Definition 2.5 *Eine* ***Modularität***[6] *ist eine modulare Zerlegung einer Pflanze. Modularitäten können für unterschiedliche Typen von Modulen definiert werden.*

Definition 2.6 *Eine* ***Skale***[7] *ist die Modellierung eines Objekts anhand einer ausgewählten Modularität.*

Die Begriffe Skale und Modularität werden synonym benutzt. Allerdings entspricht die höchste Skale dabei der feinsten Modularität und die niedrigste Skale der gröbsten Modularität.

Definition 2.7 *Ein* ***Modul***[8] *ist ein einzelner Bestandteil einer modularen Zerlegung einer Pflanze.*

[4] Diese alternativen Bezeichner sollen in einem Kontext mit mehreren Graphen zur eindeutigen Bezeichnung einer Menge eines bestimmten Graphen (Bezeichner des Graphen in den Klammern) dienen.

[5] engl. tree graph

[6] engl. modularity

[7] engl. scale

[8] engl. module

Die topologische Struktur einer modularen Zerlegung besteht aus der Beschreibung der Verbindungen zwischen den Modulen. Diese Verbindungen werden durch Kanten in einem Graphen dargestellt. In den Beispielen in [GC98] und den Beispielen dieser Arbeit werden, anhand der Eigenschaften pflanzlichen Wachstums, zwei Kantentypen definiert:

- ***Nachfolgerkanten***[9], gekennzeichnet mit dem Symbol $<$ und
- ***Verzweigungskanten***[10], gekennzeichnet mit dem Symbol $+$.

Diese beiden Kantentypen sind kein Bestandteil des Modells, sondern dienen nur innerhalb der Beispiele zur Modellierung der Beziehungen zwischen Modulen von Pflanzen. Allgemein können beliebige Kantentypen definiert werden. Ein Kantentyp, der notwendiger Bestandteil des Modells ist, wird zur Modellierung der Verfeinerungsrelationen zwischen Skalen oder Modulen benutzt:

- ***Verfeinerungskanten***[11], gekennzeichnet mit dem Symbol /.

Definition 2.8 *Eine Modularität ist eine* ***Verfeinerung*** *einer zweiten Modularität, wenn jedes Modul der zweiten Modularität in eine Menge von Modulen der ersten Modularität zerlegt werden kann und umgekehrt, jedes Modul der ersten Modularität Teil eines Moduls der zweiten Modularität ist.*

Das bedeutet, es existiert eine Verfeinerungsrelation zwischen den Modularitäten (siehe Abb. 2.1). Damit ist es möglich, beide in einer hierarchischen Struktur zu erfassen.

Definition 2.9 *Eine Modularität* ***überlappt/überlagert*** *sich mit einer zweiten Modularität, wenn mindestens ein Modul einer Modularität einen gemeinsamen Teil mit einem Modul einer anderen Modularität besitzt, wobei keines der beiden Module ein Teil des anderen Moduls ist.*

Das bedeutet, es existiert keine Verfeinerungsrelation zwischen den Modularitäten. Es ist also nicht möglich, beide in einer hierarchischen Struktur zu erfassen (siehe Abb. 2.2).

Definition 2.10 *Modularitäten sind* ***vergleichbar****, wenn sie* Verfeinerungen *voneinander sind.*

Definition 2.11 *Modularitäten sind* ***nicht vergleichbar****, wenn sie sich* überlappen/überlagern.

[9] engl. successor, bildet die Entwicklung einer Pflanze ab, die in einer Richtung erfolgt
[10] engl. branch, bildet die Entwicklung einer Pflanze ab, die zu Verzweigungen führt
[11] engl. refinement

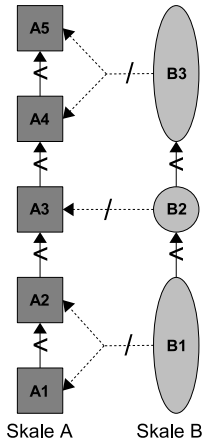


Abbildung 2.1: Beispiel zweier vergleichbarer Skalen/Modularitäten.
Dabei ist Skale A eine Verfeinerung (/) von Skale B. Jeder Knoten aus Skale A kann eindeutig einem Knoten aus Skale B zugeordnet werden.

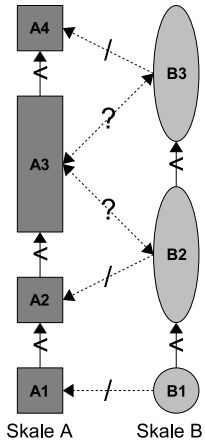


Abbildung 2.2: Beispiel zweier nicht vergleichbarer, sich überlagernder Skalen/Modularitäten.
Keine der beiden Skalen stellt eine Verfeinerung (/) der anderen dar.

Bemerkung 2.1 *Meiner Meinung nach könnte man den Vorgang, in Anlehnung an Abb. 2.3(b), bei vergleichbaren Modularitäten auch* vertikale Skalierung *und bei nicht vergleichbaren Modularitäten* horizontale Skalierung *benennen. Dabei drückt die vertikale Skalierung die zunehmende Verfeinerung von einer niedrigen zu einer hohen Skale aus. Die horizontale Skalierung hingegen drückt das Fehlen einer Verfeinerung aus.*

Wenn man nun eine vergleichbare, gröbere Modularität betrachtet, dann kann jeder Bestandteil der gröberen Modularität in eine Menge von Bestandteilen der feineren Modularität zerlegt werden. Umgekehrt ist jeder Bestandteil der feineren Modularität ein Teil eines Bestandteils der gröberen Modularität.

Diese Aufteilungsrelation zwischen den Bestandteilen kann durch eine Abbildung von der Menge der Bestandteile der feinen Modularität in die Menge der Bestandteile der gröberen Modularität modelliert werden.

Zwei vergleichbare Modularitäten können durch einen Aufteilungsgraphen dargestellt werden.

Definition 2.12 *Nach [GC98](Def.2) ist ein* ***Aufteilungsgraph***[12] *g ein Tripel (h, V, π), wobei*

- *h ein einfacher Graph ist, genannt* ***Trägergraph***[13] *von g, bezeichnet mit $\psi(g)$,*
- *V eine endliche Menge von Knoten ist, bezeichnet mit $\vartheta(g)$, und*
- *π eine surjektive*[14] *Abbildung von $\vartheta(h)$ nach V ist.*

Dabei repräsentiert der Graph h die topologische Struktur einer feinen Modularität, V repräsentiert die Bestandteile einer groben Modularität und die Abbildung π repräsentiert die Aufteilungsrelation. Die Aufteilungsrelation legt fest, wie jeder Bestandteil der groben Modularität in eine Menge von Bestandteilen der feinen Modularität aufgeteilt werden kann. Dabei wird der Knoten $\pi(x)$ der **Komplex** von x und umgekehrt x eine **Komponente** von $\pi(x)$ genannt.

In Def. 2.12 gibt es keine Informationen darüber, wie die ganze Struktur des Trägergraphen h, mit Knoten und Kanten, von der Aufteilungsfunktion π transformiert werden soll. Von außen betrachtet geht der Wechsel von der feinen zu der groben Modularität einher mit dem Entfernen von Grenzen zwischen den Bestandteilen der feinen Modularität. Deshalb wird π als binäre Relation $\Re$ zwischen den Kanten der feinen Modularität interpretiert. Das heißt, zwei Knoten von h sind

[12] auch: faktorisierter Graph, engl. quotiented graph
[13] engl. support graph
[14] engl. onto

in Relation $\Re$ zueinander, wenn sie nach einem Skalenwechsel nicht mehr vom Betrachter unterschieden werden können. Das bedeutet, dass die daraus resultierende makroskopische topologische Struktur aus der Knotenmenge V besteht, welche die Menge der mit $\Re$ assoziierten Äquivalenzklassen[15] repräsentiert, und der Menge von Kanten, die noch zwischen den erkennbaren Grenzen dieser Klassen nach einem Skalenwechsel existieren. Dieser entstandene einfache Graph wird Projektion eines Aufteilungsgraphen genannt.

Definition 2.13 *Nach [GC98](Def.3) ist die* ***Projektion*** *eines Aufteilungsgraphen $g=(h, V, \pi)$ der einfache Graph $\phi(g)=(V, E, <.>)$, wobei*

- *V die Knotenmenge $\vartheta(g)$ bezeichnet,*
- *E eine Teilmenge von $\varepsilon(h)$ ist, die folgendermaßen definiert ist:*
  $\forall e \in \varepsilon(h), <e>_h = (x, y) : e \in E \Leftrightarrow \pi(x) \neq \pi(y)$
- *die Inzidenzfunktion $<.>_{\phi(g)}$ ist, die folgendermaßen definiert ist:*
  $\forall e \in E : <e>_h = (x, y) \Rightarrow <e>_{\phi(g)} = (\pi(x), \pi(y))$.

Definition 2.14 *Nach [GC98](Def.4) ist ein* ***Aufteilungsbaumgraph***[16] *ein Aufteilungsgraph, dessen Träger- und Projektionsgraph Baumgraphen sind.*

Ohne auf die zur Herleitung durchgeführten Zwischenschritte einzugehen, die in [GC98] ausführlich beschrieben sind, werden im Folgenden die multiskalierten Baumgraphen beschrieben.

Aufteilungsgraphen können verallgemeinert werden, indem ausgehend von einem Startgraphen ständig neue Aufteilungsoperationen auf die nachfolgenden Aufteilungsgraphen angewendet werden. Der multiskalierte Graph soll deshalb eine rekursive Struktur haben, die diese Eigenschaft wiedergeben kann.

Die Darstellung eines Objekts in verschiedenen Skalen führt dadurch zu einer pyramidenähnlichen Struktur, deren Basis aus den Knoten der feinstmöglichen Skale besteht und deren Spitze aus dem einzelnen Knoten der gröbsten möglichen Skale besteht, nämlich dem Knoten, der die Pflanze als Gesamtheit repräsentiert. In dieser Struktur ist das Objekt in jeder Skale vollständig beschrieben. Das Modell soll aber auch unvollständige Beschreibungen des Objektes erfassen können.

Dies führt zu der in [GC98](Def.10) aufgeführten Definition eines multiskalierten Graphen:

[15] Klassen von Knoten, die nach einem Wechsel zu einer gröberen Skale nicht mehr voneinander unterschieden werden können

[16] engl. quotiented tree graph

Definition 2.15 *Ein einfacher Graph g ist ein* ***multiskalierter Graph***[17]*. Die Knotenmenge des einfachen Graphen g definiert die Knotenmenge des multiskalierten Graphen, genannt* $\vartheta(g)$*.*
Wenn h ein multiskalierter Graph ist, dann ist g ebenfalls ein multiskalierter Graph. Dabei ist $g=(h,\ V,\ \pi,\ \delta V,\ \delta E,\ <.>)$ *und:*

- *V ist eine Menge von Knoten, bezeichnet mit* $\bar{\vartheta}(g)$*,*
- π *ist eine surjektive Abbildung von* $\vartheta(h)$ *nach* $\bar{\vartheta}(g)$*, bezeichnet mit* π_g
- δV *ist eine Menge von Knoten, bezeichnet mit* $\delta\vartheta(g)$*, und* $\vartheta(g)$ *ist die Menge* $\bar{\vartheta}(g) \cup \delta\vartheta(g)$*,*
- δE *ist eine Menge von Kanten, bezeichnet mit* $\delta\varepsilon(g)$*,*
- $<.>$ *ist eine Funktion von* δE *nach* $(V \cup \delta V)^2$*, bezeichnet mit* $<.>_g$

Wenn g ein einfacher Graph ist, dann wird er *abschließender multiskalierter Graph*[18] genannt, sonst heißt er *rekursiver multiskalierter Graph*[19]. Bei einem rekursiven multiskalierten Graph $g=(h,\ V,\ \pi,\ \delta V,\ \delta E,\ <.>)$ heißt h der Trägergraph von g. Die ersten drei Komponenten von g $(h,\ V,\ \pi)$ entsprechen den Komponenten eines Aufteilungsgraphen, außer dass h jetzt, statt eines einfachen Graphen, etwas allgemeiner ein multiskalierter Graph ist. Das zeigt, dass multiskalierte Graphen eine rekursive Struktur besitzen. Die letzten drei Komponenten von g $(\delta V,\ \delta E,\ <.>)$ dienen dazu, den Fall zu modellieren, dass in einer Skale nicht alle Knoten in eine Menge feinerer Knoten aufgeteilt werden. Die Knoten aus $\bar{\vartheta}(g)$ werden zerlegt, die aus δV nicht. Letztere sind auf der feinsten Skale nicht sichtbar.

Definition 2.16 *Nach [GC98](Def.12) ist ein* ***multiskalierter Baumgraph***[20] *ein multiskalierter Graph, dessen Trägergraph ein multiskalierter Baumgraph und dessen Projektion ein Baumgraph ist.*

Multiskalierte Graphen wurden definiert, um topologische Strukturen von Pflanzen darzustellen, die aus vergleichbaren Modularitäten bestehen. Solche topologischen Strukturen werden durch einen einzigen multiskalierten Baumgraphen *g1* erfasst, der aus einer Serie von multiskalierten Baumgraphen $g_1,\ g_2, \ldots, g_m$ besteht, die untereinander in den in [GC98] Abschnitt 3.3 beschriebenen Beziehungen stehen. Das bedeutet, dass g_m der Trägergraph von g_{m-1} ist. Der Projektionsgraph jedes multiskalierten Graphen g_i stellt dabei die topologische Struktur einer Pflanze in einer der vergleichbaren Modularitäten dar.

[17] engl. multiscale Graph
[18] engl. terminal multiscale graph
[19] engl. recursive multiscale graph
[20] engl. multiscale tree graph

Dieses Modell kann erweitert werden, um die topologischen Strukturen von nicht vergleichbaren, sich überlagernden Modularitäten (siehe Def. 2.11 und 2.9) abzubilden.

Zwei sich überlagernde Modularitäten C und D (siehe Abb. 2.3(b)) seien also nicht vergleichbare Darstellungen einer Pflanze. So sei jede der Modularitäten für sich jedoch mit einer dritten, feineren Modularität E vergleichbar. Diese Situation kann dadurch modelliert werden, dass zwei verschiedene rekursive multiskalierte Graphen h_C und h_D mit dem gleichen Trägergraph k_E erstellt werden.

$h_C = (k_E, V_C, \pi_C, \delta V_C, \delta E_C, <.>_C)$

$h_D = (k_E, V_D, \pi_D, \delta V_D, \delta E_D, <.>_D)$

In dieser Erweiterung wird die topologische Struktur der Pflanze also nicht durch einen einzelnen Graphen beschrieben, sondern durch eine Menge von (in diesem Fall zwei) multiskalierten Graphen (h_C, h_D). Der Projektionsgraph des multiskalierten Graphen k_E ist eine Verfeinerung der Projektionsgraphen der beiden multiskalierten Graphen h_C und h_D. Die Projektionsgraphen der beiden multiskalierten Graphen h_C und h_D sind aber nicht vergleichbar. Es können also zwischen nicht vergleichbaren Modularitäten keine Beziehungen modelliert werden.

Dieses erweiterte Konzept wird durch verallgemeinerte multiskalierte Graphen beschrieben.

Definition 2.17 *Nach [GC98](Def.13) ist ein* ***verallgemeinerter multiskalierter Graph***[21] *eine indizierte Menge* $(g_1, g_2, \ldots, g_m)$ *von nicht vergleichbaren multiskalierten Graphen.*

2.3 Modellierung multiskalierter Objekte mit Graphen

Das in Abschnitt 2.2 beschriebene Prinzip bildet die Grundlage für das geplante Datenformat. Die darin beschriebene Art der Modellierung von multiskalierten Strukturen durch Graphen wird benutzt, um die durch Graphen beschriebenen Skalen eines multiskalierten Objektes in einem XML-Format abzulegen. Dafür wird das eigentlich auf die Abbildung topologischer Strukturen von Pflanzen spezialisierte Konzept nun für allgemeine Objekte genutzt. Die im Modell in Abschnitt 2.2 beschriebene Einschränkung auf baumartige Graphstrukturen bleibt bestehen. Es gibt keine Beschränkung der Anzahl an Typen von Relationen (Kantenarten). Da die Art der Verfeinerungsbeziehungen zwischen den vergleichbaren Skalen aber hierarchisch ist, dürfen die zur Modellierung der Verfeinerungsbeziehungen definierten Kanten keine Zyklen im Graphen erzeugen. Die gleiche Einschränkung gilt zur Zeit auch für

[21] engl. generalized multiscale graph

alle anderen Kanten, aufgrund der Einschränkung auf baumartige Graphstrukturen. Das Modell wird um das Konzept von multiskalierten Modulen erweitert. Diese werden mithilfe von multiskalierten Knoten realisiert. Darauf wird in Abschnitt refsec:schachtelungmultiskalierterobjekte eingegangen.

Der in Abb. 2.3(b) dargestellte Graph dient dazu, die Zusammenhänge zwischen den verschiedenen Skalen eines Objektes zu modellieren. Vergleichbare Skalen werden dabei mit Verfeinerungskanten (/) verbunden, nicht vergleichbare Skalen jedoch nicht. Dieses Vorgehen wird später auf das Dateiformat übertragen, indem die Knoten vergleichbarer Module in den Skalen miteinander über Verfeinerungskanten verbunden werden.

Ein Beispiel des Graphen eines multiskalierten Objektes ist in Abb. 2.3(a) zu sehen. Dort sind die fünf Skalen des multiskalierten Objekts durch verschiedenfarbig eingefärbte Graphen dargestellt. Um die Übersichtlichkeit zu wahren, sind nur die Kanten auf der Ebene der feinsten Skale E vorhanden. Die Kanten auf den Ebenen der anderen Skalen können, nach dem in Abschnitt 2.4.1 geschilderten Prinzip, welches sich aus Def. 2.13 ergibt, hergeleitet werden. Die Verfeinerungsrelationen sind für die Skalen in Abb. 2.3(b) und für die einzelnen Module anhand des Enthaltenseins ineinander dargestellt.

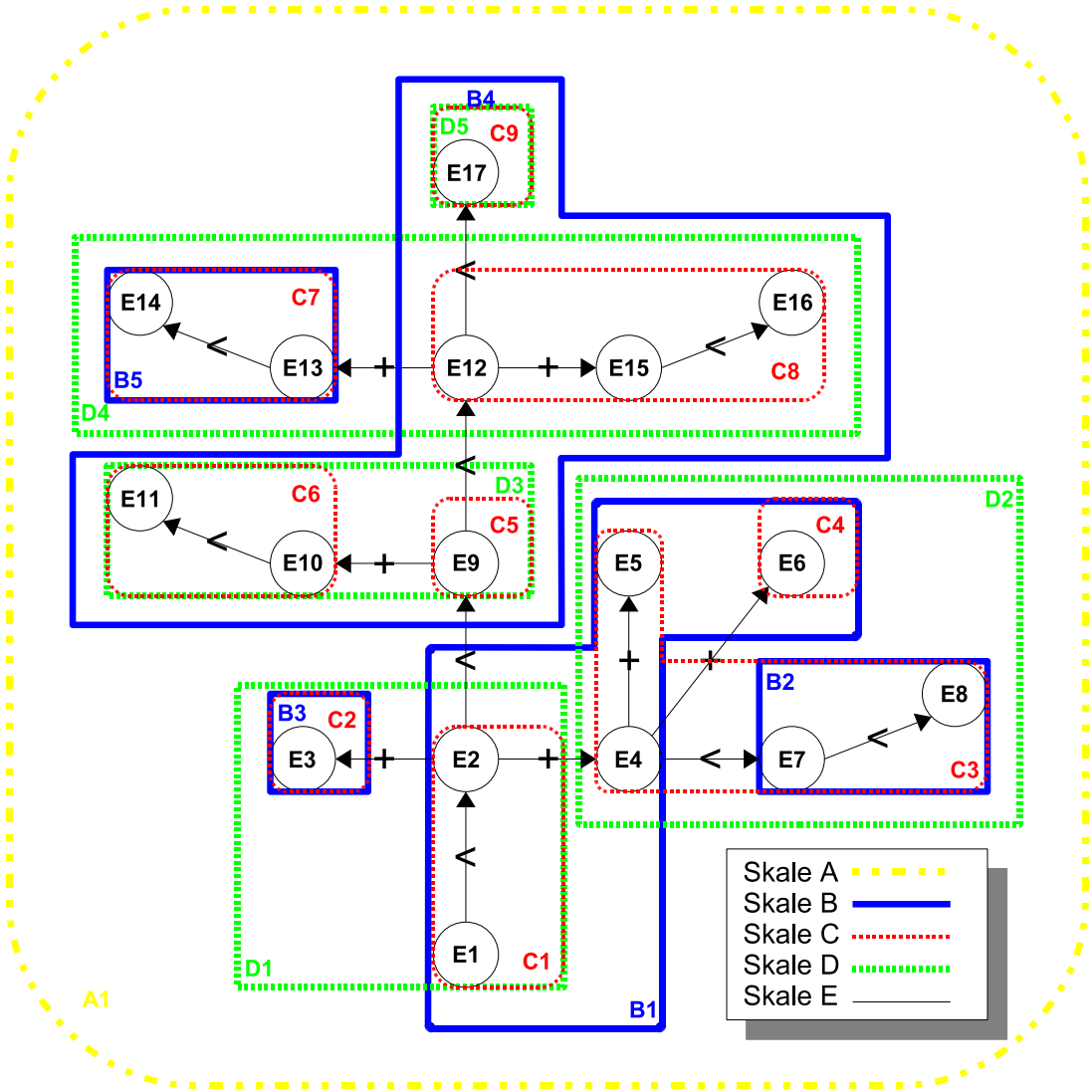
2.4 Probleme beim Umgang mit multiskalierten Objekten

In den folgenden Unterabschnitten werden Problemstellungen beim Umgang mit multiskalierten Objekten beschrieben.

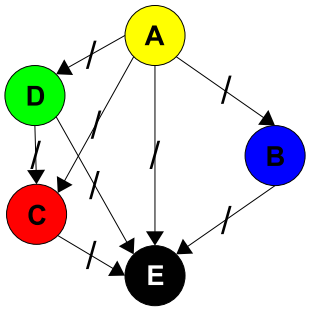
2.4.1 Zusammenhang zwischen den Modulen verschiedener Skalen

Im Allgemeinen ist es Aufgabe des Nutzers, den Zusammenhang zwischen den einzelnen Modulen innerhalb einer Skale eines multiskalierten Objektes anhand des von ihm gewählten Modells zu beschreiben. Zwischen vergleichbaren Skalen gibt es aber auch die Möglichkeit, bei einer vorhandenen Beschreibung der Beziehungen der Module der feineren Skale und vorhandenen Verfeinerungsbeziehungen zwischen den Skalen, die Beziehungen der Module auf der gröberen Skale automatisch herzuleiten. Das Prinzip lässt sich in Abb. 2.1 erkennen. Dabei wird, als Reaktion auf die Feststellungen in dem Absatz (vor Def. 2.13) über die Projektion von Aufteilungsgraphen, an die Relationen zwischen den Modulen die Forderung gestellt, dass nur die folgenden beiden Fälle auftreten:

1. Die Relationen zwischen Modulen der feineren Skale, die Verfeinerungen des



(a) Beispiel eines multiskalierten Graphen



(b) Graph der Verfeinerungsrelationen (/) zwischen den Skalen

Abbildung 2.3: Multiskalierter Graph eines Objekts mit 5 Skalen mit Verzweigungs- (+) und Nachfolgerkanten (<) in Skale E.

gleichen Moduls der gröberen Skale darstellen, werden ignoriert. (Zum Beispiel sind in Abb. 2.1 die Module A1 und A2 beide Verfeinerungen des Moduls B1. Das heißt, es ist keine Kante innerhalb von B1 nötig.)

2. Die Relationen zwischen Modulen der feineren Skale, die Verfeinerungen von unterschiedlichen Modulen der gröberen Skale darstellen, werden übernommen. Dabei bleibt der Typ der Beziehung gleich. (Zum Beispiel ist in Abb. 2.1 das Modul A2 eine Verfeinerung von B1, aber Modul A3 ist eine Verfeinerung von B2. Das heißt, es gibt eine Nachfolger-Kante von B1 nach B2.)

Diese sinnvolle Konvention ist für die Aufgabenstellung nicht relevant, kann aber für zukünftige Werkzeuge zur Handhabung, Erstellung oder Validierung multiskalierter Strukturen benutzt werden.

2.4.2 Schachtelung von multiskalierten Objekten

Ein multiskaliertes Objekt kann innerhalb einer Skale Module besitzen, die ebenfalls durch ein multiskaliertes Objekt beschrieben werden können.

Beispiel 2.3 *Innerhalb einer detaillierten Skale einer Pflanze können die Blütenmodule ebenfalls als multiskalierte Objekte dargestellt werden.*

Ein multiskaliertes Objekt O habe eine Skale S. Wenn Skale S, neben den normalen Modulen, auch ein Modul M besitzt, das ebenfalls multiskaliert ist, dann kann man die Skale S von O durch die Anzahl von Skalen zu ersetzen, die der Anzahl an Skalen entspricht, die M enthält. Dies führt aber bei einer größeren Anzahl von unterschiedlichen multiskalierten Modulen zu einer großen Anzahl von Skalen mit ähnlichen Inhalten und ist ineffizient und unhandlich.

Die bessere Möglichkeit ist, multiskalierte Objekte auch als Knoten im Graphen zuzulassen. Diese werden im Folgenden ***multiskalierte Knoten*** genannt.

Da in Abschnitt 2.3 eine Einschränkung der Graphen einer Skale auf Baumgraphen erfolgt ist, gilt diese Beschränkung auch für die durch multiskalierte Knoten darstellbaren Module. Weiterhin wird festgelegt, dass das Ziel einer Kante von einem Knoten zu einem multiskalierten Knoten nur die Wurzel des Graphen der Skale des multiskalierten Knoten sein darf.

Ein Problem ergibt sich für die nachfolgenden Knoten eines solchen multiskalierten Knoten im Graphen des multiskalierten Objekts. Der multiskalierte Knoten besitzt ebenfalls pro Skale einen Graph, der ihn beschreibt. Beim Verbinden der Knoten eines Graphen des multiskalierten Objekts mit einem multiskalierten Knoten ist aber nicht eindeutig, von welchem Unterknoten des multiskalierten Knoten die ausgehenden Kanten starten, bzw. in welchem Unterknoten die eingehenden Kanten

enden. Deshalb muss bei Kanten innerhalb eines Graphen, deren Ausgangspunkt ein multiskalierter Knoten ist, für jede Skale des multiskalierten Knotens eine separate Kante angegeben werden. Deren Ausgangspunkt ist der Unterknoten der jeweiligen Skale des multiskalierten Knotens, der als Endpunkt innerhalb der Skale des multiskalierten Knoten festgelegt wurde. Solche Kanten müssen gekennzeichnet[22] werden, da sie den Übergang zwischen zwei Knoten des Graphen darstellen. Dieser Übergang ist aber beim Traversieren des Graphen des multiskalierten Knoten nicht feststellbar, da innerhalb des einbettenden Graphen und innerhalb des Graphen eines multiskalierten Knotens die gleichen Kantentypen (z. B. Verzweigung (+), Nachfolger (<)) vorkommen. Die Knoten des einbettenden Graphen, welche die Nachfolger des multiskalierten Knoten sind, würden als Teilknoten des multiskalierten Knoten erkannt werden.

Das gleiche Problem würde auch für eingehende Kanten in multiskalierte Knoten gelten wenn, entgegen der weiter oben gemachten Einschränkungen,

- mindestens einer der Graphen der Skalen eines multiskalierten Knoten kein Baumgraph, sondern ein allgemeiner Graph ohne ein ausgewiesenes Wurzelelement wäre bzw.

- eingehende Kanten in multiskalierte Knoten zugelassen würden, die einen beliebigen Knoten innerhalb des Skalen eines multiskalierten Knotens als Endpunkt haben dürften.

Diese Fälle werden aber im weiteren Verlauf aufgrund der weiter oben in diesem Abschnitt gemachten Einschränkungen nicht berücksichtigt.

Multiskalierte Knoten arbeiten also wie eine Weiche, wobei beim Traversieren des Graphen der Weg eingeschlagen wird, der durch die gewählte Skale des multiskalierten Knotens vorgegeben ist. In Abb. 6.4 wird der Sachverhalt noch einmal grafisch dargestellt.

2.4.3 Strukturveränderungen während des Lebenszyklus multiskalierter Objekte

Die Zugehörigkeit einer Komponente zu einem bestimmten Modul einer Skale kann sich im Laufe der Entwicklung eines Objektes ändern.

Beispiel 2.4 *Ein Ast eines Baumes konkurriert mit dem Stamm und wird irgendwann zum Stamm. Der bisherige Teil des Stammes wird zum Ast. Das bedeutet eine Änderung der Modultypen von Ast und Stamm.*

[22] In GroIMP geschieht diese Kennzeichnung durch einen eigenen Kantentyp, der solchen Kanten zusätzlich zugeordnet wird.

Eine Komponente kann aber auch völlig aus dem Modell verschwinden.

Beispiel 2.5 *Ein Baum verliert seine Blätter.*

Auch dieser Anwendungsfall ist für die Aufgabenstellung nicht relevant, kann aber für zukünftige Werkzeuge zur Handhabung und Erstellung multiskalierter Strukturen wichtig sein.

In [GC98] Abschnitt 4 wird eine zeitlich veränderliche Variante der multiskalierten Graphen besprochen. Diese kann eine Lösung für dieses Problem darstellen.

2.4.4 Synchronisieren des Objektzustandes in verschiedenen Skalen

Änderungen innerhalb einer Skale eines multiskalierten Objektes können Änderungen in den anderen Skalen des Objektes notwendig machen.

Beispiel 2.6 *Ein Baum wächst und erhält innerhalb einer entsprechenden Skale neue Äste, Zweige und Blätter. Wie wirkt sich das auf die anderen Skalen aus, zum Beispiel auf eine Skale, in welcher der Kronenumfang durch eine Kugel dargestellt wird?*

Dabei existieren zum einen die Probleme bei der Vergleichbarkeit von Skalen. Andererseits ist ohne ein konkretes Regelsystem, was den Zusammenhang zwischen den Skalen beschreibt, ein automatisches Abgleichen des Zustandes zweier Skalen undenkbar. Es bleibt also vorerst dem Benutzer und dessen Wissen über die verschiedenen Modelle überlassen, die Zustände der verschiedenen Skalen konsistent zu halten, was bei Skalen mit einer großen Anzahl an Knoten kaum praktikabel erscheint. Dieses Problem zu lösen, ist aber nicht Aufgabe eines Datenformats und sei deshalb in diesem Zusammenhang nur erwähnt.

Kapitel 3

Datenformate

Ein Modell, was die Erfassung multiskalierter Strukturen ermöglicht, ist das des *MTG*[1] [GCS99]. Das im Zusammenhang mit der *AMAPMOD*-Software [GG97] verwendete textbasierte Format zur Codierung von *MTG*s ist allerdings nicht *XML*-basiert und außerdem in der Anwendung auf die Erfassung von Pflanzenstrukturen spezialisiert. Während *MTG* nur ein Modul zur topologischen Beschreibung multiskalierter Graphstrukturen ist, gibt es innerhalb der *AMAPMOD*-Software auch noch das *GEOM*-Modul, das die Beschreibung einer Szene im dreidimensionalen Raum ermöglicht, indem geometrische und das Aussehen beschreibende Eigenschaften dem *MTG* zugeordnet werden.

3.1 Allgemeine Anforderungen für ein Datenaustauschformat

In [HSSW05](Abschnitt 2.3) werden Anforderungen an ein Standardaustauschformat aufgeführt. Diese Anforderungen werden in den folgenden Unterabschnitten aufgeführt und erläutert werden.

3.1.1 Allgemeingültigkeit

Ein Datenaustauschformat soll den Austausch von Daten für die verschiedensten Zwecke ermöglichen, um den Austausch zwischen verschiedenen Modellen auf unterschiedlichen Abstraktionsebenen zu erlauben.

[1] Multiscaled Tree Graph

3.1.2 Typisierbarkeit

Ein Datenaustauschformat soll typisiert sein. Das Kennen der verwendeten Datentypen erleichtert die Interpretation der ausgetauschten Daten. Typisierte Datenaustauschformate erlauben auch die Validierung[2] der ausgetauschten Daten. Das Definieren von einheitlichen Datentypen und ihrer Abhängigkeiten untereinander hilft bei der Standardisierung anwendungsspezifischer Modelle.

3.1.3 Flexibilität

Ein Datenaustauschformat soll flexibel sein. Es soll in der Lage sein, verschiedenste anwendungsspezifische Daten zu erfassen, um eine breite Nutzungsbasis zu gewährleisten.

3.1.4 Einfachheit

Ein Datenaustauschformat soll so entworfen sein, dass seine Unterstützung durch Programme einfach zu implementieren ist. Solche Programme sind zum Beispiel Im- und Exportfilter, Übersetzer von und nach anderen Formaten und Validierungsprogramme, welche die Integrität der auszutauschenden Daten überprüfen.

3.1.5 Skalierbarkeit

Ein Datenaustauschformat soll mit unterschiedlich komplexen Modellen umgehen können. Deshalb muss es in der Lage sein, effizient auch mit großen Datenmengen umgehen zu können.

Bei einem multiskalierten Ansatz kommt hinzu, dass ein Objekt in mehreren Modellen/Darstellungsweisen erfasst wird, wodurch sich die zu handhabende Datenmenge vervielfacht und deshalb der Effizienzgedanke besondere Bedeutung gewinnt.

3.1.6 Modularisierbarkeit

Ein Datenaustauschformat soll den modularen und schrittweisen Datenaustausch unterstützen, um Daten trennen, verstecken und verteilen zu können. Das heißt, Daten sollen in Teilen, als Subsysteme oder in unterschiedlichen Dokumenten ausgetauscht werden können. Dazu gehört die Fähigkeit des Formates, mit externen Datenquellen umgehen zu können. Dafür muss unter Umständen auch die Notwendigkeit einer Verwaltung von Zugriffsrechten untersucht werden, z. B. die Einschränkung von Zugriffsrechten auf externe Quellen.

[2] Überprüfung auf Gültigkeit von Daten, im Hinblick auf die Syntax und Semantik des verwendeten Formates

3.1.7 Erweiterbarkeit

Ein Datenaustauschformat soll die Möglichkeit bieten, die Modellierungskonzepte in speziellen Versionen des Datenaustauschformats erweitern zu können. Die Erweiterbarkeit eines Datenaustauschformats bestimmt über das Ausmaß an Verwendungsmöglichkeiten in anderen Anwendungsbereichen, durch das Hinzufügen neuer Elemente oder die Benutzung des Formates als Basis oder Teil eines neuen Datenaustauschformats.

3.2 Anforderungen zur Erfassung multiskalierter Objekte

Nach [God00] enthält die Beschreibung einer Pflanzenarchitektur mindestens eine der folgenden Informationen. Ich verallgemeinere das und gehe davon aus, dass diese Informationen auch für die Architekturen von Objekten anderer Art vorhanden sind und demzufolge durch ein Format erfasst werden sollen:

- ***Zerlegungsinformationen***, die beschreiben, wie das Objekt aus den einzelnen Komponenten aufgebaut ist und welche Typen von Komponenten verwendet werden.
- ***Geometrische Informationen***, welche die Form und die räumliche Lage der Komponenten beschreiben. Dies entfällt bei nicht-räumlichen Objekten.
- ***Topologische Informationen***, die beschreiben, welche Komponenten miteinander verbunden sind und in welcher Weise.

Für die Erfassung mehrerer Objekte in einer Gruppe, zum Beispiel zur Beschreibung einer Szene, die mehrere Objekte enthält, müssen Gruppierungselemente verfügbar sein, die eine klare Abgrenzung der einzelnen Objekte voneinander erlauben.

Den Objekten, Gruppen von Objekten und Einzelelementen innerhalb der Objekte müssen Metadaten[3] zugeordnet werden können. Die Metadaten sollen flexibel durch den Benutzer gehandhabt und definiert werden können und nicht auf wenige primitive Datentypen beschränkt sein.

Die Beziehungen zwischen den Objekten müssen beschrieben werden. Auch dort müssen Metadaten zugeordnet werden können, um die Art der Beziehung flexibel zu beschreiben.

Es muss die Möglichkeit bestehen, Kanten zwischen Knoten von verschiedenen Stufen eines hierarchischen Graphen zu erstellen, um Kanten zwischen Elementen verschiedener Skalen eines Objekts modellieren zu können.

[3]Daten, welche die Eigenschaften eines Objektes beschreiben

Es soll eine Möglichkeit bestehen, in externen Dateien definierte Objekte einzubinden bzw. vordefinierte Objekte mehrmals innerhalb einer Datei zu verwenden. Das dient dazu, das Format kompakter und übersichtlicher zu gestalten, aber ebenfalls dazu, einen verteilten Datenbestand und somit ein verteiltes Arbeiten mehrerer Gruppen am Datenbestand zu ermöglichen.

Eine Möglichkeit zum Anlegen einer Bibliothek von vordefinierten Elementen und Metadaten zur Verwendung innerhalb des Formates soll ebenso gegeben sein.

3.3 Grundlagen zu XML

Dieser Abschnitt befasst sich mit den Möglichkeiten der verschiedenen XML-Techniken, die im Rahmen dieses Projekts von Bedeutung sind.

3.3.1 XML

XML (eXtensible Markup Language [BPSM+06]) ist ein vom W3C[4] vorgeschlagener Standard zur Beschreibung strukturierter Daten. XML ist eine Metasprache zur Beschreibung von Markup-Sprachen und ein wichtiges Werkzeug zum Datenaustausch. Ein XML-Dokument besitzt eine hierarchische Struktur mit genau einem Wurzelelement (deshalb auch Baumstruktur genannt). Dieses Wurzelelement kann beliebig viele Unterelemente besitzen.

Beispiel 3.1 *Dies ist ein einfaches Beispiel, welches den Aufbau einer XML-Datei zeigt. Dabei sieht man die* Elemente *(baum, stamm, ast, radius) und die* Attribute *(alter, radius). Man erkennt auch die Möglichkeit, die Eigenschaft eines Elements sowohl als Unterelement (< radius >)als auch als Attribut (radius) zu modellieren.*

```
<baum alter="2">
    <stamm radius="50">
        <ast>
            <radius>15</radius>
        </ast>
        <ast radius="10"/>
    </stamm>
</baum>
```

Elemente bestehen aus Start- (< baum >) und Endmarken (< /baum >), bzw. aus nur einer Marke[5]*(< ast/ >) bei leeren Elementen(=Elemente ohne Unterelemente).*

[4] World Wide Web Consortium: `http://www.w3.org`, letzter Zugriff am 30.11.2006

[5] engl. tag

XML erleichtert den Austausch und die Konvertierung von Daten, ist erweiterbar (z. B. durch die Möglichkeit zur Definition eigener Elemente und Attribute), plattformunabhängig, lizenzfrei und unterstützt internationale Zeichensätze (z. B. Unicode) und Namensräume.

Namensräume kennzeichnen die Zugehörigkeit von Elementen zu Wissensdomänen. Mehr dazu in Abschnitt 3.3.3.

XML kann mithilfe von XML-Schemata oder DTDs auf Gültigkeit überprüft werden. Diese Form der Überprüfung einer XML-Datei setzt voraus, dass diese wohlgeformt[6] ist und dem in den XML-Schemata oder DTDs definierten Format entspricht. Mehr dazu in Abschnitt 3.3.2.

Einer der wenigen Nachteile ist die Dateigröße von in XML formulierten Daten im Vergleich zu binär erfassten Datenrepräsentationen. Da XML ein textbasiertes Format ist und Tags[7] verwendet, um die Daten abzugrenzen, sind XML-Dateien fast immer größer als vergleichbare binäre Formate. Dies wurde aber bewusst in Kauf genommen, da dieser Nachteil meist an anderer Stelle ausgeglichen werden kann. Speicherplatz ist nicht mehr so teuer wie früher, und Programme wie *zip* und *gzip* können Dateien gut und schnell komprimieren. Außerdem erlauben aktuelle Kommunikationsprotokolle Daten automatisch zu komprimieren und damit ebenso effektiv Bandbreite zu sparen wie ein binäres Format. Mittlerweile gibt es zahlreiche XML-basierte Sprachen aus allen Bereichen. Eine kleine Auswahl an Beispielen von XML-Anwendungen sei im Folgenden genannt:

- *XHTML*[8]: eine Sprache zur Beschreibung von Internetseiten und Nachfolger von HTML
- *MathML*[9]: eine Sprache zur Beschreibung mathematischer Formeln und komplexer Ausdrücke.
- *SVG*[10]: eine Sprache zur Beschreibung skalierbarer zweidimensionaler Vektorgrafiken
- *RDF*[11]: eine Sprache zur Beschreibung von Metadaten
- *CellML*[12]: eine Sprache zur Beschreibung biologischer Modelle
- *X3D*[13]: eine Sprache zur Beschreibung dreidimensionaler Objekte, virtueller Welten und Szenengraphen

[6] wird definiert in Abschnitt 2.1 von [BPSM+06]

[7] XML-Elemente, die Text, oder andere Tags (auch in Form von Attributen) enthalten und mit ihrem Namen selbst zur Beschreibung von Daten dienen können. z. B. $< jahr > 2006 < /jahr >$ oder $< jahr\ value = "2006" / >$

[8] extensible Hypertext Markup Language [W3C06]

[9] Mathematical Markup Language [CIM+03]

[10] Scalable Vector Graphics [FJ03]

[11] Resource Description Framework [HSB06]

[12] Cell Modelling Language [Ne]

[13] Extensible 3D [Con06]

3.3.2 XML-Schema

XML-Schema [TBMM04] ist eine vom W3C empfohlene Sprache zur Definition von XML-Dokumentstrukturen. Ein Schema kann anhand benutzerdefinierter Tags und Datentypen den Aufbau eines XML-Dokuments beschreiben. Durch einen Parser können diese Daten ausgelesen werden.

Im Gegensatz zu einer DTD[14] besteht XML-Schema nur aus XML. Außerdem bietet es im Gegensatz zur DTD einen breiteren Umfang an Datentypen [BM04], ein komplexeres Typensystem und das Konzept der Namensräume (siehe Abschnitt 3.3.3). So ist es möglich, einfache oder komplexe Typen zu definieren, Typen von anderen Typen abzuleiten (indem man existierende Typen erweitert oder einschränkt) oder Elemente mit beliebigem Typ zuzulassen. Ebenso kann man bestehende XML-Schemata importieren und redefinieren.

Ist für ein XML-Dokument ein XML-Schema verfügbar, kann man die Gültigkeit des Dokuments überprüfen. Dabei erstreckt sich die Überprüfung nicht nur auf die Struktur (Anordnung, Anzahl und Reihenfolge von Elementen und Attributen), wie bei den DTDs, sondern, aufgrund des komplexeren Typensystems, auch auf die Typen. Diesen Vorgang nennt man Validierung.

3.3.3 Namensräume

Das Konzept der Namensräume[15] [BHLT06] dient in XML zur Kennzeichnung der Herkunft der Sprachelemente bzw. der Zugehörigkeit zu einem Sprachvokabular. Es ist eine flexible Methode zur eineindeutigen Benennung von XML-Elementen und Attributen ohne die Notwendigkeit einer zentralen Registrierung. So können Konflikte zwischen gleichlautenden Elementen oder Attributen aus verschiedenen Kontexten gelöst werden.

Ein Namensraum wird mit einem Namensraumpräfix versehen, mit dem alle ihm zugehörigen Elemente gekennzeichnet werden. Der Namensraum wird mit einer RFC2396-konformen URI[16] identifiziert. Dabei gelten nur Namensräume als identisch, deren URIs Zeichen für Zeichen identisch sind.

Beispiel 3.2 *Für X3D wird im Rahmen dieses Projektes der mit dem Präfix* ***x3d:*** *versehene Namensraum* `http://www.web3d.org/specifications` *benutzt. Alle GroIMP-spezifischen Attribute besitzen den mit dem Präfix* ***g:*** *versehenen Namensraum* `http://grogra.de/msml/datatypes/groimp`*. In Bsp. 4.1 kann man den Konflikt zweier gleichnamiger XML-Elemente sehen (Appearance), welcher mithilfe von Namensräumen aufgelöst wurde.*

[14] Dokumenttypdefinition (engl. Document Type Definition, DTD) ist eine nicht XML-basierte Sprache zur Definition von XML-Dokumentstrukturen.

[15] engl. Namespace

[16] Uniform Resource Identifier [BFM05], zur Begriffsklärung zwischen URI und URL siehe [URI01]

3.3.4 XSLT

XSLT (eXtensible Stylesheet Language Transformations) ist eine XML-basierte Transformationssprache für XML-basierte Dokumente. Diese kann die Struktur und den Inhalt eines Dokuments verändern.

XSLT erfordert einen XSLT-Prozessor. Das ist eine Software, die ein XSLT-Stylesheet auf ein Quelldokument anwendet und als Resultat das neue, transformierte Dokument erstellt.

Nach [Kay06] beschreibt eine in XSLT 2.0 formulierte Transformation Regeln zur Transformation von 0 oder mehr XML-basierten Ausgangsbäumen in einen oder mehrere Ergebnisbäume. Die Struktur dieser Bäume ist in [WFMe06] beschrieben. Die Transformation wird durch eine Menge von Regeln, in Form von Schablonen[17], beschrieben. Eine solche Regel definiert das Muster, welches eine Menge von Knoten im Ausgangsdokument beschreibt, mithilfe eines Sequenzkonstruktors. In manchen Fällen führt die Ausführung des Sequenzkonstruktors dazu, dass neue Knoten erstellt werden, die einen Teil des Ergebnisbaums bilden. Der Aufbau des Ergebnisbaums kann eine völlig andere Struktur haben als der Ausgangsbaum. Beim Erstellen des Ergebnisbaumes können die Knoten des Ausgangsbaumes gefiltert und geordnet oder auch beliebige Strukturen hinzugefügt werden. Durch dieses Prinzip ist eine XSLT-Transformation auf eine breite Klasse von Dokumenten mit gleicher Ausgangsbaumstruktur anwendbar.

XSLT-Stylesheets können flexibel eingesetzt werden. So können diese zum Beispiel in einer Kommandozeilenversion zur Anwendung gebracht werden und arbeiten plattformunabhängig.

Da XSLT aber eine funktionale Transformationssprache ist, besitzt es auch die Nachteile funktionaler Sprachen. So wird man beim Ausdrücken von Schleifen oder Variablen (die nach ihrer Initialisierung nicht mehr verändert werden können und daher eher Konstanten entsprechen) meist auf den Umweg über Rekursion gezwungen, mit all den damit einhergehenden Problemen.

Auch kann ein XSLT-Stylesheet selbst nicht auf seine Ausgabe zugreifen, was mir insbesondere beim Einbinden von Elementen externer XML-Dateien Probleme bereitet hat.

Diese Nachteile sind der Tatsache geschuldet, dass XSLT-Funktionen und XSLT-Schablonen keine Seiteneffekte verschulden dürfen. Das bedeutet, Funktionen liefern bei jedem Aufruf die gleichen Ergebnisse und sind nicht von der Reihenfolge oder der Anzahl ihrer Aufrufe abhängig.

Es gibt mittlerweile neben XSLT 1.0 [Cla99] auch eine Empfehlung für XSLT 2.0 [Kay06]. Darin sind eine Menge von neu- und weiterentwickelten Konzepten [Kay04] enthalten, welche die Handhabung deutlich vereinfachen und zu zahlreichen Verbesserungen geführt haben, auf die in Abschnitt 5.1 näher eingegangen wird.

[17] engl. Templates

3.3.5 Einbinden von Elementen externer XML-Quellen

In Abschnitt 3.2 wurde die Notwendigkeit einer Möglichkeit zum Einbinden von Informationen aus externen XML-Datenquellen genannt. In den folgenden Unterabschnitten sollen einige der aktuell existierenden oder noch in der Entwicklung befindlichen Methoden zum Einbinden externer XML-Daten in eine XML-Datei untersucht werden.

Dabei soll auch geprüft werden, inwieweit die untersuchten Methoden bereits ein automatisches Einbinden der externen Ressourcen durch den XML-Prozessor realisieren.

XLink

XLink (XML Linking Language) [DMO01] ist eine Empfehlung des W3C und beschreibt eine Standardmethode, um Hyperlinks zu XML Dateien hinzuzufügen.

Man unterscheidet zwischen einfachen und erweiterten XLinks. Dabei entsprechen die einfachen XLinks den heute in HTML üblichen unidirektionalen Hyperlinks zwischen zwei Ressourcen. Die erweiterten XLinks können:

- Verbindungen zwischen mehr als zwei Ressourcen darstellen,
- Traversierungsregeln formulieren,
- zwischen lokalen und entfernten Ressourcen unterscheiden,
- zwischen eingehenden, ausgehenden und Verbindungen zwischen nicht-lokalen Ressourcen[18] unterscheiden und
- für Menschen lesbare Bezeichnungen für einen Link enthalten.

Den eigentlichen Verweis enthält das Attribut xlink:href="URI", wobei URI durch einen Uniform Resource Identifier zu ersetzen ist, der ebenso ein XPointer-Ausdruck (siehe Abschnitt 3.3.5) für interne Verweise sein kann.

XInclude

XInclude (XML Inclusion) [MO04] ist eine Empfehlung des W3C, die das Verweisen auf Teile von XML- oder Textdateien ermöglicht. Damit ist das Zusammenfügen mehrerer nicht XML-basierter Textdokumente, wohlgeformter XML-Dokumente oder Teilen davon zu einer großen XML-Datei möglich.

Das Anwenden von XInclude-Anweisungen wird aber nicht von XML-Parsern vorgenommen. Erforderlich ist ein eigener XInclude-Prozessor, welcher die

[18] sogenannten Drittanbieterkanten: Verbindung zwischen Ressourcen, die nicht in der lokalen Datei enthalten sind

xi:include-Elemente durch die Dokumentteile ersetzt, auf die sie verweisen. Man kann dies auch durch einen SAX-Filter erreichen, der die XInclude-Anweisungen auflöst. Es existieren für JAVA mittlerweile einige Anwendungen, die XInclude-Anweisungen auflösen können.

XPointer

XPointer ist ebenfalls eine Empfehlung des W3C und beschreibt ein Framework, das es ermöglicht mit URIs und XPath[19]-Ausdrücken auf Fragmente von XML-Dokumenten zu verweisen.

Das Framework ist als Basis für alle Bezeichner von XML-Fragmenten vorgesehen, deren MIME-Typ einem der Folgenden entspricht:

- text/xml,
- application/xml,
- text/xml-external-parsed-entity oder
- application/xml-external-parsed-entity.

Benutzer anderer XML-basierender MIME-Typen sind angehalten, ebenfalls dieses Framework zu benutzen, um Sprachen zu entwickeln, die ihre eigenen Fragmentbezeichner unterstützen.

Das Framework wurde in ein Basisschema [GEJN03b], welches die Grundlagen der XPointer-Sprache enthält und drei zusätzliche Schemata aufgeteilt:

- das *element()*-Schema [GEJN03a], um die grundlegende Adressierung von XML-Elementen zu ermöglichen,
- das *xmlns()*-Schema [DREJ03], um die Benutzung von Namensräumen bei der Adressierung zu ermöglichen und
- das *xpointer()*-Schema [DRE02], für die Unterstützung einer vollständig auf XPath basierenden Adressierung.

Auch für die Anwendung der XPointer sind eigenständige XPointer-Prozessoren oder entsprechende Filter notwendig.

[19] die XML Path Language [CD99][Bon04] ist eine Sprache für die Adressierung von Elementen und Attributen eines XML-Dokuments

Schlussfolgerung

Keine der untersuchten Methoden ist so weit fortgeschritten, dass ein automatisches Einbinden externer Elemente durch den XML-Prozessor möglich ist. Weder XPointer noch XInclude sind ohne XPointer-, XInclude-Prozessor oder vom Anwender anzuwendende Filter für die Auflösung von Verweisinstruktionen einzusetzen. In Abschnitt 5.2 wird ein solcher Filter auf Basis von XSLT für die Einbettung externer Elemente realisiert. Dabei orientiert sich die Syntax des in MSML verwendeten Verweiselements (msml:extSrc) an Teilen der XLink-Spezifikation. Eine vollständige Benutzung von XLink für Kanten (msml:edge) und Verweiselemente (msml:extSrc) wurde aus Zeitgründen nicht realisiert.

Diese Lösung wurde gewählt, da in keiner der untersuchten Methoden Ansätze zur Lösung der in Abschnitt 5.2.1 und 5.2.2 beschriebenen Probleme zu erkennen waren.

Keine der Methoden ist außerdem zur Validierung einer Datei mit Verweisen auf externe Quellen zu gebrauchen, da das Ergebnis einer solchen Validierung davon abhängt, ob die Verweise vor oder nach dem Validieren aufgelöst wurden. Da dieses Auflösen aber nicht automatisch durch den XML-Prozessor geschieht, sondern manuell durch den Anwender geschehen muss, ist das Ergebnis einer solchen Validierung nicht aussagekräftig.

3.4 Untersuchung XML-basierter Datenformate für Graphen

Ein auf *XML* basierendes Datenformat zur Modellierung von Pflanzen wird offenbar in einem Artikel [YL01] beschrieben, dessen Titel „A XML-Based Plant Modeling Language“ lautet. Leider war es nicht möglich, diesen Artikel in irgendeiner Form aufzufinden. Es wäre interessant, ob und in welcher Form darin mit dem Thema Multiskalierung umgegangen wurde.

Im Folgenden sollen einige existierende, auf *XML* basierende Datenformate zur Erfassung von Graphen dahingehend untersucht werden, ob sie sich als Grundlage zur Erfassung multiskalierter Objekte eignen und eventuell als Basis für das zu entwickelnde Format dienen können. Dabei gelten die in den Abschnitten 3.1 und 3.2 genannten Kriterien. Zusätzlich werden auch noch andere Kriterien gelten, wie die Akzeptanz des Formates bei den Anwendern, der Umfang der Unterstützung durch Programme und die Verfügbarkeit von Dokumentationen und Quellen.

Die Akzeptanz eines Formates ist unter anderem an der Verfügbarkeit einer aussagekräftigen Internetseite, auf der aktuelle Dokumentationen, *XML-Schemata* oder *DTD*s abzurufen sind, zu erkennen. Auch eventuell unterschiedliche Versionen der verfügbaren Schemata, die Existenz von lebendigen Mailinglisten, Foren oder Kon-

taktadressen zu den Entwicklern weisen darauf hin, ob ein Format in der Praxis verwendet wird und der Entwicklungsprozess lebendig ist, und damit auf den Grad der Akzeptanz und Verwendung durch die Nutzer. Die Schemata dienen zum einen der Dokumentation der Formate, zum anderen ist mit ihrer Hilfe die Validierung der erstellten *XML*-Daten möglich.

3.4.1 XGMML

Die *eXtensible Graph Markup and Modeling Language* [Pun01] ist eine auf *GML*[20] basierende *XML*-Sprache, die unter anderem dazu genutzt wird, die Struktur der Graphen von Internetseiten oder von *RDF*-Daten [21] zu beschreiben.

Die wesentlichen Eigenschaften sind die Unterstützung von

- gerichteten, ungerichteten und gemischten Graphen,
- hierarchischen Graphen (Subgraphen),
- grafischen 2D-Darstellungen des Graphen,
- Referenzen auf externe Daten und
- anwendungsspezifischen Attributen.

Eine *DTD* und ein *XML-Schema* der Sprache sind ebenso verfügbar, wie Programme zur Umwandlung von und nach *GML*. *XGMML* verwendet $< graph >$, $< node >$ und $< edge >$-Elemente zur Beschreibung des Graphen, wobei zusätzlich durch Angabe von $< att >$-Elementen Metainformationen zu den zugehörigen Oberelementen definiert werden können. $< att >$-Elemente innerhalb von $< graph >$-Elementen können Subgraphen beschreiben. Ein $< graphics >$-Element dient zur Beschreibung der grafischen Erscheinung eines Elements, beschränkt sich aber auf zweidimensionale Grafiken und Elemente. *XGMML* ist zur Beschreibung von topologischen Strukturen konzipiert. Um dreidimensionale Objekte damit zu beschreiben, müsste das $< graphics >$-Element um entsprechende geometrische Typen erweitert werden. Die Möglichkeit, *RDF*-Syntax innerhalb der $< att >$-Elemente zu verwenden, ermöglicht eine flexible Verwendung von selbst definierten Attributen. Mittels eines $< locator >$-Elements können Elemente aus externen Quellen benutzt werden.

[20] Die nicht *XML*-basierte *Graph Modeling Language* [Him97] war das Ergebnis einer Initiative, die auf den „Graph Drawing“-Symposien 1995 in Passau begann und 1996 in Berkeley endete. *GML* wird von einigen Systemen zur Erzeugung von Graphen unterstützt.

[21] Resource Description Framework [HSB06]: dient zur Beschreibung von Metadaten

3.4.2 GraphXML

GraphXML wurde beim „Graph Drawing“-Symposium 2000 in Williamsburg vorgestellt und seit April 1999 benutzt. Es existiert allerdings nur ein Dokument [IH00], aber keine offizielle Webseite. Nur die *DTD* des Formats ist als Anhang in der Dokumentation verfügbar. Das Projekt scheint nicht mehr aktiv betreut zu werden. Benutzt wird es offenbar nur von *Royere/GVF*[22].

Die wesentlichen Eigenschaften sind die Unterstützung von

- gerichteten, ungerichteten und gemischten Graphen,
- hierarchischen Graphen (Subgraphen),
- Kanten zwischen Knoten von verschiedenen Stufen hierarchischer Graphen,
- dynamischen Graphen,
- grafischen 2D-Darstellungen von Graphen,
- Referenzen auf externe Daten,
- anwendungsspezifischen Attributen,
- der Übergabe von Parametern an festgelegte Anwendungen und
- Erweiterungen der Sprachsyntax durch interne *DTD*s.

Als Einziges der vier untersuchten Formate bietet es die Möglichkeit zum Beschreiben der dynamischen Entwicklung eines Graphen. Dazu können Aktionen wie das Hinzufügen, Entfernen und Ändern eines Knotens in das Format eingefügt werden.

GraphXML verwendet $< graph >$, $< node >$ und $< edge >$-Elemente zur Beschreibung des Graphen. Zur Verwendung anwendungsspezifischer Attribute gibt es $< label >$-Elemente, $< data >$-Elemente und $< dataref >$-Elemente.

$< data >$-Elemente können beliebige *XML*-Sprachkonstrukte enthalten. Dies lässt dem Anwender freie Hand bei der Definition von anwendungsspezifischen Attributen. Dies setzt aber auch eine präzise Dokumentation der definierten Attribute vonseiten des Anwenders voraus. Eine Validierung dieser selbst definierten Attribute ist ohne die Möglichkeit zur Angabe eines Schemas oder einer *DTD* für die Attribute nicht möglich.

$< dataref >$-Elemente dienen dazu, mittels *XLink* (siehe Abschnitt 3.3.5) auf Attribute von externen Quellen zuzugreifen.

Ebenso können die geometrischen Positionen der Knoten und Kanten gespeichert werden.

[22] Graph Visualization Framework [Mar]

3.4.3 GraphML

Die *Graph Modeling Language* [Gra04] ist ein umfangreicher und einfach zu benutzender, auf *XML* basierender Dateiformatstandard für Graphen, der bei den „Graph Drawing"-Symposien 2000 in Williamsburg und 2001 in Wien erstellt wurde. Einer der Vorgänger von *GraphML* ist *GML*. Neben der Dokumentation durch [BEL04][Gra04][BEH+02], dem *XML-Schema* und der *DTD* sind auch noch Beispiele und ein paar Programme zum Konvertieren verfügbar. Allerdings konnte nur eine Anwendung gefunden werden, die in der Lage war, auf *GraphML* basierende Dateien einzulesen und darzustellen. Der *yEd - Java Graph Editor* [yEd06] der Firma yWorks implementiert jedoch das Referenzieren externer Daten noch nicht.

GraphML besteht aus einem Sprachkern zur Beschreibung der strukturellen Eigenschaften eines Graphen und eines flexiblen Erweiterungsverfahrens zum Hinzufügen von anwendungsspezifischen Daten durch Erweiterungen des *XML-Schemas* durch vom Anwender zu definierende Module[23]. Die wesentlichen Eigenschaften sind die Unterstützung von

- gerichteten, ungerichteten und gemischten Graphen,
- Hypergraphen[24],
- hierarchischen Graphen (Subgraphen),
- Kanten zwischen Knoten von verschiedenen Stufen hierarchischer Graphen,
- grafischen 2D-Darstellungen des Graphen,
- Referenzen auf externe Daten und
- anwendungsspezifischen Attributen.

Auch *GraphML* verwendet $< graph >$, $< node >$ und $< edge >$-Elemente zur Beschreibung des Graphen. Zur Verwendung anwendungsspezifischer Attribute gibt es $< data >$-Elemente. Diese Attribute müssen aber vor der ersten Benutzung durch ein $< key >$-Element definiert werden, wodurch die Identifikationsnummer[25], der Name, der Typ und der Gültigkeitsbereich für das Attribut festgelegt werden. Die ID muss eindeutig sein und wird zur Referenzierung des Attributes im Dokument verwendet. Der Typ eines Attributes kann entweder ein primitiver Datentyp (boolean, int, float, double oder string) sein oder auch strukturierte Daten (z. B. einen komplexen *XML*-Ausdruck) enthalten, wenn das *XML-Schema* von GraphML um

[23] ähnlich dem in *XHTML* verwendeten modularen Aufbau [ABD+01]

[24] ist eine verallgemeinerte Art von Graph, die Hyperkanten enthält. Hyperkanten können mehr als nur zwei Knoten verbinden.

[25] wird im Folgenden mit der Kurzform ID benannt. Sie dient der eindeutigen Identifizierung eines Objektes innerhalb einer XML-Datei

entsprechende komplexe Typen erweitert wurde. Der Gültigkeitsbereich eines Attributes kann sich nur auf Graphen, Knoten, Kanten oder auf alle Elemente erstrecken. Zusätzlich erlaubt *GraphML* innerhalb eines $< key >$-Elements ein $< default >$-Element zur Festlegung von Standardwerten für $< graph >$-Elemente, die kein $< data >$-Element besitzen.

Externe Elemente werden mithilfe eines $< locator >$-Elements mit einem *XLink*-Attribut, wie in [BEH+02] beschrieben, eingebettet.

Des Weiteren wurde in [BEL04] die Möglichkeit beschrieben, unter Benutzung des Erweiterungsverfahrens des *XML-Schemas* zum Hinzufügen anwendungsspezifischer Daten, die ebenfalls *XML*-basierte Sprache *SVG*[26] einzubinden, um eine einheitliche Beschreibungsgrundlage für die visuellen Attribute der Graphelemente zu erhalten.

Bemerkung 3.1 *Das Beispiel der Verbindung von* SVG *und* GraphML *brachte mich auf die Idee, eine Verknüpfung von* X3D *und* GraphML *zur Beschreibung dreidimensionaler Objekte zu realisieren. Dabei stieß ich auf zahlreiche Probleme.*

Zum einen gibt es für X3D *noch keinen standardisierten Namensraum. Dieser ist aber notwendig, um innerhalb eines erweiterten* GraphML-*Schemas die zu importierenden* X3D-*Elemente eindeutig zuordnen zu können. Deshalb wählte ich als Namensraum zur Kennzeichnung der* X3D-*Elemente die Adresse* `http://www.web3d.org/specifications`*, an der sich die* DTD *und das* XML-*Schema von* X3D *befinden. Diese Vorgehensweise wird in einigen Foren derzeit vorgeschlagen, um trotz des Fehlens einer offiziellen Empfehlung eine einigermaßen einheitliche Kennzeichnung von* X3D-*Elementen zu erreichen.*

Weiterhin stieß ich auf einen Fehler bei den Namensräumen in den XML-*Schemata von* GraphML*, welcher aber nach Kontaktaufnahme mit einem auf der* GraphML-*Internetseite angegebenen Mitarbeiter korrigiert wurde. Ebenfalls auf Probleme traf ich bei der Validierung der (nach den Vorgaben im* GraphML-*Primer [BEL04] zum Einbinden von* SVG*) erstellten Schemaerweiterungen zum Einbinden von* X3D *mit Altova's XMLSpy 2006. Eine Validierung mit dem Schemavalidator [TT04] des W3C funktionierte hingegen.*

3.4.4 GXL

Die *Graph eXchange Language* [HSSW02] ist ein auf *XML* basierendes Austauschformat für Graphen. Eine *DTD* und ein *XML-Schema* der Sprache sind ebenso verfügbar, wie zahlreiche Beispiele, Dokumentationen[27] und Programme zur Visua-

[26] Scalable Vector Graphics [FJ03], beschreibt zweidimensionale Grafiken in *XML*

[27] `http://www.gupro.de/GXL/Introduction/intro.html`, letzter Zugriff am 26.10.2006

lisierung, Konvertierung und Editierung von *GXL*-Dateien. Die wesentlichen Eigenschaften sind die Unterstützung von

- gerichteten, ungerichteten und gemischten Graphen,
- Hypergraphen,
- hierarchischen Graphen (Subgraphen),
- Kanten zwischen Knoten von verschiedenen Stufen hierarchischer Graphen,
- Graphschemata und Instanzgraphen[28],
- Referenzen auf externe Daten und
- anwendungsspezifischen Attributen.

Ein Unterschied zu all den anderen Formaten besteht in der Möglichkeit, Graphschemata zu erstellen und Instanzgraphen gemeinsam mit den zugehörigen Schemata in einem einheitlichen Format auszutauschen. In verschiedenen Anwendungsbereichen ist es üblich, die Form eines Graphen festzulegen, zum Beispiel durch die Einschränkung der verfügbaren Knotentypen. Ein Schema bietet die Möglichkeit, eine Klasse von Graphen zu beschreiben. Darin wird festgelegt,

- welche Arten von Knoten, Kanten und Hyperkanten benutzt werden dürfen,
- welche Beziehungen zwischen Knoten, Kanten und Hyperkanten der verschiedenen Arten existieren dürfen,
- welche Attribute den Knoten, Kanten und Hyperkanten zugeordnet werden können,
- welche Hierarchien in den Graphen erlaubt sind und
- welche zusätzlichen Einschränkungen gelten sollen.

Diese Einschränkungen spezialisieren einen Graphen, um den gewünschten Teil eines Modells abzubilden.

Auch *GXL* verwendet $< graph >$, $< node >$ und $< edge >$-Elemente zur Beschreibung des Graphen. Die $< node >$ und $< edge >$-Elemente können $< attr >$-Elemente enthalten, die den Namen und den Wert eines Attributes beschreiben. Wegen der Kompatibilität mit Programmen, die typisierte Attribute benutzen, stellt

[28]Graphen, die Instanzen dieser Graphschemata sind (ähnlich dem Prinzip der Klassen und Instanzen von Klassen bei der Objektorientierung)

auch *GXL* typisierte Attribute zur Verfügung. Diese Informationen werden meistens im Schema einer Graphklasse definiert. Da aber *GXL* die Verwendung von Graphschemata nicht voraussetzt, können die Attributtypen auch in den Dokumenten der Instanzgraphen durch entsprechende Bezeichner angegeben werden. *GXL* unterstützt $< bool >, < int >, < float >$ *und* $< string >$ Attribute, sowie Aufzählungstypen $< enum >$, URI-Referenzen auf externe Objekte $< locator >$, sowie zusammengesetzte Attribute wie Sequenzen $< seq >$, Mengen $< set >$, Multimengen $< bag >$ und Tupel $< tup >$. $< attr >$-Elemente enthalten nur ein Datenelement, z. B. $< int >$ oder $< set >$. Aber sie können auch wieder $< attr >$-Elemente enthalten, um Attribute von Attributen zu beschreiben.

3.4.5 Zusammenfassung

Als Zusammenfassung meiner Untersuchung der vier Formate unter Einbeziehung der Ergebnisse aus [HSSW05] und [BLP05] komme ich zu dem Schluss, dass sich *XGMML*, *GraphML*, *GraphXML* und *GXL* nur in einigen Merkmalen und Eigenschaften unterscheiden. Dies hängt vermutlich damit zusammen, dass alle Formate gemeinsame Wurzeln im *GML*-Format haben.

Das Prinzip, nach dem in allen vier Formaten der Graph erfasst wird, als auch die Syntax (abgesehen von den Attributnamen) sind identisch. Die Knoten werden als ungeordnete Liste von $< node >$-Elementen erfasst, wobei jedes einen eindeutigen Bezeichner erhält. Die Kanten zwischen den Knoten werden durch $< edge >$-Elemente mit Verweisen auf die Bezeichner der Quell- und Zielknoten dargestellt. Kanten können als gerichtet oder ungerichtet modelliert werden. Diese Kantentypen können auch gemeinsam verwendet werden. Eines der wichtigsten Entwurfsziele von *GraphML* und *GXL* war die Möglichkeit zur Unterstützung anwendungsspezifischer Erweiterungen. Dafür können ihre *DTD*s oder *XML Schemata* verändert und durch individuelle Elemente und Attribute erweitert werden, was für Anwendungen von Interesse ist, die, anstatt der primitiven Datentypen, komplexe *XML*-Unterbäume als Datentypen benötigen.

Manchmal reichen konventionelle Graphen nicht aus, um bestimmte Modelle zu beschreiben. Deshalb bieten *GraphML* und *GXL* die Möglichkeit, Hyperkanten zu definieren, die eine Verallgemeinerung der konventionellen Kanten darstellen und die Beziehungen zu einer beliebigen Anzahl von Knoten beschreiben können.

Das Konzept der verschachtelten (hierarchischen) Graphen kann in allen vier Sprachen modelliert werden. Danach können Knoten wiederum Graphen enthalten. Syntaktisch wird dies, außer in *XGMML*, dadurch erreicht, indem $< node >$ und $< graph >$-Elemente ineinander verschachtelt werden. Dies kann benutzt werden, wenn ein Graph verschiedene Abstraktionsebenen besitzt.

Da alle vier Formate eine Möglichkeit bieten, Kanten zwischen Knoten von verschiedenen Stufen eines hierarchischen Graphen zu erstellen, und damit Kanten

zwischen Elementen verschiedener Skalen eines Objekts modellieren zu können, ist eine Grundvoraussetzung zur Modellierung multiskalierter Objekte durch alle vier Formate erfüllt. Somit können Beziehungen zwischen Elementen verschiedener, vergleichbarer Skalen beschrieben werden.

Am deutlichsten werden die Unterschiede bei *GXL*, das als einziges Format Schemata von Graphen und Instanzen davon definieren kann. In einem Vergleich von *GXL*, *GraphML* und *GraphXML* in [HSSW05] kommt man zu dem Schluss, dass *GraphML* erfolgreicher ist als *GraphXML* (dessen Entwicklung anscheinend zugunsten von *GraphML* aufgegeben wurde, das wiederum anscheinend in den Entwicklungsprozess von *GXL* integriert werden soll). *GraphML* biete dabei ein ähnliches Kernmodell eines Graphen wie *GXL*. Während die Anpassungsfähigkeit von *GXL* auf der Metamodellierungstechnologie der Graphschemata basiert, beruht die Anpassungsfähigkeit von *GraphML* auf der Erweiterung der *GraphML-DTD* bzw. des *GraphML-XML-Schemas*. Deshalb benutzen *GraphML*-Dokumente verschiedene *anwendungsabhängige* (domänenspezifische) *DTDs/XML-Schemata* mit einem gemeinsamen Sprachkern. Im Gegensatz dazu benutzt *GXL* eine gemeinsame, *anwendungsunabhängige* Dokumentendefinition. Beide Formate gelten als kompatibel, und es existieren entsprechende Filter, um *GraphML* und *GXL* ineinander zu konvertieren [BLP05]. *GXL* scheint, wenn man die Liste[29] der Programme betrachtet, die es verwenden, das verbreitetere *XML*-basierte Format zur Beschreibung von Graphen zu sein.

Alle vier Formate bieten eine Möglichkeit, mithilfe eines $<$ *locator* $>$-Elements Verweise auf interne oder externe Elemente zu realisieren.

Meine Bedenken hinsichtlich der mangelhaften Unterstützung des Einbindens externer Quellen durch die vier untersuchten Formate konnten jedoch nicht vollständig entkräftet werden. Mithilfe von XLink und den meist vorhandenen Locator-Elementen lässt sich auf externe Elemente verweisen. Die entsprechenden Sprachelemente sind aber teilweise unzureichend dokumentiert und werden deshalb kaum von Anwendungen implementiert. Damit ist ein reibungsloser Austausch mit anderen Nutzern dieser Formate nicht gewährleistet und ein Argument für die Nutzung der Formate ist entkräftet.

Ein Hauptargument gegen die Verwendung besteht in der mangelnden Übersichtlichkeit der Formate bei einer Modellierung verschachtelter Strukturen. Da in allen Formaten zur Modellierung multiskalierter Graphen nur die Elemente $<$ *graph* $>$ und $<$ *node* $>$ zur Verfügung stehen, würde die Übersichtlichkeit komplexerer multiskalierter Beschreibungen schnell in schier endlosen Kaskaden von $<$ *graph* $>$- und $<$ *node* $>$-Elementen verloren gehen. Eine Unterscheidung von Gruppen, Objekten, Skalen und Knoten ist anhand der Elementbezeichner nicht möglich und selbst die Struktur der verschachtelten Elemente führt nicht immer zu

[29] `http://www.gupro.de/GXL/tools/tools.html`, letzter Zugriff am 26.10.2006

eindeutigen Schlüssen. Die einzelnen $< graph > / < node >$-Elemente müssten ihre Rollen anhand zusätzlicher Attribute zugewiesen bekommen, was nicht nur den Umfang des Formates vergrößern, sondern auch bei der späteren Verarbeitung des Formates zu zusätzlichem Aufwand führen würde. Das hätte der Effizienz des Formates geschadet.

Außerdem ist der Aufbau einer Bibliothek und die Verwendung von den in dieser Bibliothek definierten Attributen in den meisten der vorgestellten Formate nicht möglich. Dies hängt zum Beispiel damit zusammen, dass *GXL* eine Strategie der Trennung von Struktur und visuellen Eigenschaften eines Graphen verfolgt. Deshalb ist eine Möglichkeit der Definition und Wiederverwendung von Eigenschaftselementen in diesem Format nicht vorgesehen. Einzig *GraphML* bietet mit der key/data-Methode eine Möglichkeit, globale Attribute zu definieren, mit deren Hilfe man eine Bibliothek realisieren kann.

Im Nachhinein betrachtet sind zwar alle vier untersuchten Formate als Basis für ein multiskaliertes Datenformat anwendbar, jedoch ist dies nur mit den eben genannten Einschränkungen möglich.

Aus diesem Grund definierte ich ein eigenes *XML*-Format. Dieses ähnelt syntaktisch den untersuchten Formaten. Es enthält zusätzliche Elemente zur besseren Verständlichkeit der strukturellen Beschreibung von multiskalierten Objekten und Objektgruppen.

Kapitel 4

MSML - Multiscale Modeling Language

MSML ist ein XML-basiertes Datenformat zur Erfassung graphbasierter, multiskalierter Objekte und Gruppen von Objekten. Damit lassen sich sowohl dreidimensionale Szenen aus mehreren Objekten, als auch einfache graphbasierte Strukturen beschreiben. Es bietet die Möglichkeit, beliebige Eigenschaften für die Elemente zu definieren. Eine Auflistung aller MSML-Sprachelemente und deren Erläuterung findet sich in Anhang A. Eine schematische Darstellung des XML-Schemas von MSML ist in Abb. 4.1 zu sehen. Beispiele von MSML-Quelltext finden sich in Anhang B.

4.1 Erfassung einzelner Objekte

Das < *msobject* >-Element (siehe Anhang A.5) dient zum Erfassen einzelner Objekte. Dabei spielt es keine Rolle, ob das Objekt multiskaliert ist oder nicht. Ist es nicht multiskaliert, wird es als multiskaliertes Objekt mit nur einer Skale beschrieben.

4.2 Erfassung von Gruppen von Objekten

Das < *group* >-Element (siehe Anhang A.4) dient zum Gruppieren von multiskalierten Objekten und anderen Gruppen.

4.3 Beschreibung eines Objektes

Ein Objekt besteht aus einer Anzahl von < *scale* >-Elementen (siehe Anhang A.6). Jedes < *scale* >-Element enthält einen Graphen, der die Struktur des Objekts in dieser Skale beschreibt. Der derzeitige Stand der Implementierung setzt voraus, dass

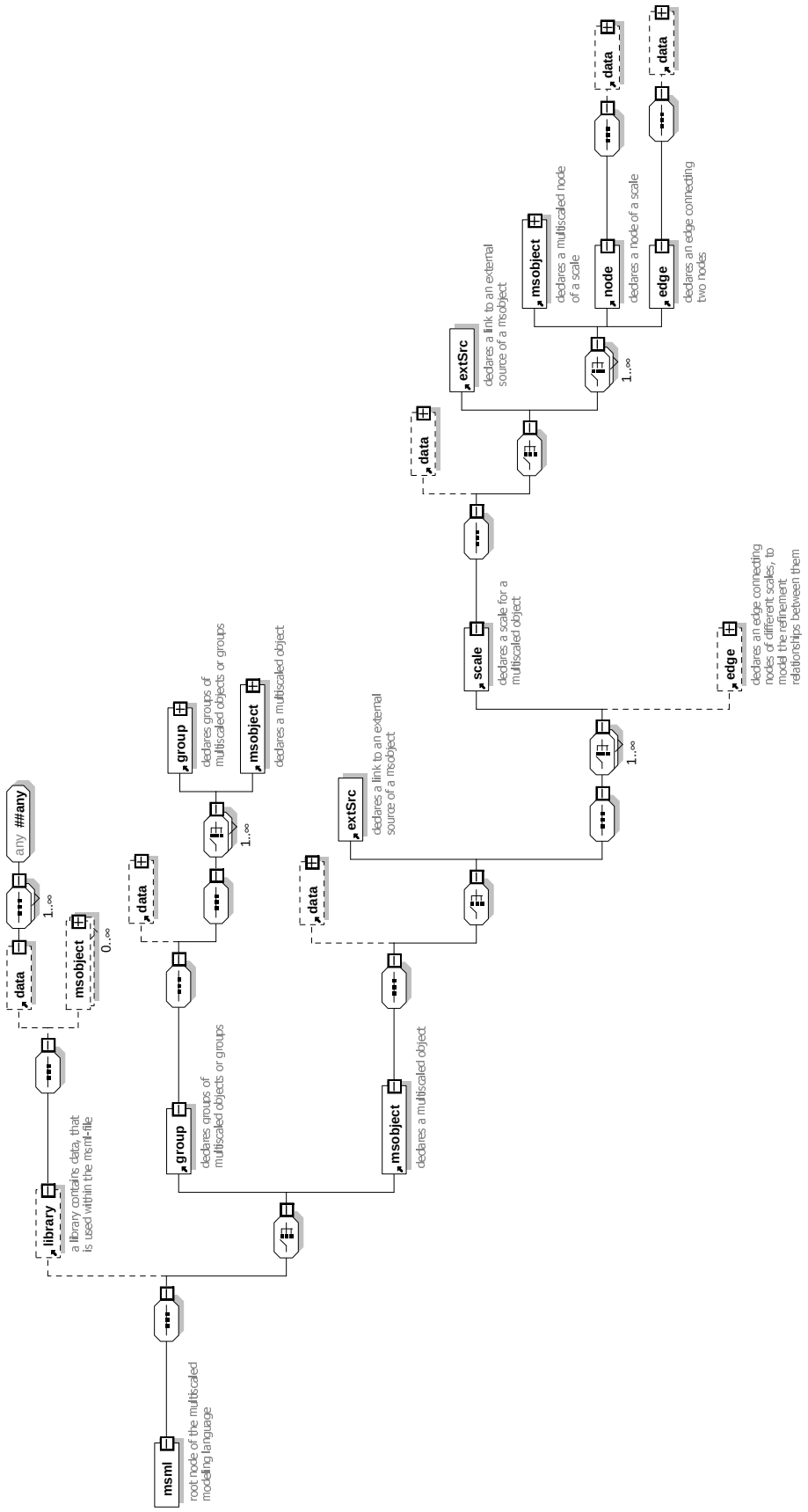


Abbildung 4.1: Aufbau des XML-Schemas von MSML

dieser Graph ein Baumgraph ist. Die einzigen Elemente, die innerhalb des Graphen vorkommen dürfen, sind Knoten (< *node* >), multiskalierte Knoten (< *msobject* >), Kanten (< *edge* >) und Eigenschaftselemente (< *data* >). Um die Verfeinerung von Knoten einer Skale durch Knoten einer anderen Skale zu modellieren, werden Kanten auf der gleichen Hierarchieebene wie der < *scale* >-Elemente benutzt. Die Verfeinerungskanten, vom Typ *refinement*, besitzen als Ausgangspunkt den zu verfeinernden Knoten und als Endpunkt einen der verfeinernden Knoten. Es müssen so viele Verfeinerungskanten definiert werden, wie es verfeinernde Knoten für einen zu verfeinernden Knoten gibt. Die Beziehungen zwischen den Knoten, die einen Knoten verfeinern, werden durch die Beziehungskanten in der Skale dieser Knoten beschrieben. Deshalb muss nur die Menge der verfeinernden Knoten für einen zu verfeinernden Knoten angegeben werden, aber nicht deren Reihenfolge oder Beziehung untereinander. Dieses Prinzip wurde in Def. 2.13 und dem dieser Definition vorausgehenden Textabschnitt beschrieben.

4.4 Beschreibung von Kanten

Eine durch ein < *edge* >-Element (siehe Anhang A.8) beschriebene Kante ist eine gerichtete Kante. Ungerichtete Kanten werden durch zwei < *edge* >-Elemente beschrieben.

In MSML existieren momentan vier vordefinierte Kantentypen.

- *successor* (siehe Abschnitt 2.2)
- *branch* (siehe Abschnitt 2.2)
- *refinement* (siehe Abschnitt 2.2)
- *userdefined* dient zur Kennzeichnung benutzerdefinierter Kantentypen. Ein < *edge* >-Element vom Typ *userdefined* kann in seinem < *data* >-Element Metadaten zur Beschreibung des Kantentyps enthalten.

Generell kann jedes < *edge* >-Element in seinem < *data* >-Element kantenspezifische Metadaten enthalten, z. B. Informationen über Kantengewichte, grafische Darstellungsattribute, alternative Bezeichnungen, Kantennamen und vieles mehr.

4.5 Definition von Metadaten

Das < *data* >-Element (siehe Anhang A.3) kennzeichnet den Bereich, in dem Metadaten[1] des Elternelements mithilfe von Eigenschaftselementen (siehe Anhang A.10)

[1] Daten, welche Eigenschaften beschreiben

definiert werden. Das < *data* >-Element enthält beliebige gültige XML-Elemente, die durch Namensräume gekennzeichnet sind. Diese verweisen auf die Quellen der Definitionen und die Herkunft der Elemente.

Beispiel 4.1 *Das mit dem Präfix g: gekennzeichnete GroIMP-spezifische Attribut (siehe Anhang A.10.1) in diesem Beispiel, dient der Beschreibung des Aussehens eines Objektes. Dabei verweist g: auf den Namensraum* `http://grogra.de/msml/datatypes/groimp`. *Diese Internetadresse soll Informationen über den Aufbau und die Bedeutung des Elementes und der dadurch beschriebenen Eigenschaft, sowie Möglichkeiten zur Kontaktaufnahme mit dem Autor enthalten.*

```
<g:Appearance>
    <x3d:Appearance>
        <x3d:Material diffuseColor="1 1 0"/>
    </x3d:Appearance>
</g:Appearance>
```

Wie man sehen kann, enthält das Element g:Appearance *selbst keine Informationen über das zu beschreibende Aussehen. Dafür benutzt es das durch den Namensraumpräfix x3d: gekennzeichnete Element* x3d:Appearance*, welches der Sprache X3D entstammt. Das Element* g:Appearance *hat also in diesem Zusammenhang nur den Zweck, die Herkunft dieses Attributes zu kennzeichnen, nämlich dass es im Zusammenhang mit der GroIMP-Software definiert wurde und sich unter* `http://grogra.de/msml/datatypes/groimp` *weitere Informationen zu diesem Element finden lassen.*

4.6 Definition von Bibliotheken

MSML erlaubt mithilfe des < *library* >-Elements (siehe Anhang A.2) die Definition einer Bibliothek von Objekten und Attributen. Derzeit gilt, dass jeder Name eines Bibliothekselements, unabhängig vom Typ des Elements, eindeutig sein muss. Man kann dies aber auch dahingehend ändern, dass der Name eines Bibliothekselements nur unter den Elementen gleichen Typs (g:Appearance in Bsp. 4.2) oder Elementen des gleichen Namensraumes eindeutig sein muss.

Beispiel 4.2 *Das Beispiel zeigt Definition und Benutzung eines Bibliothekselements. Die Syntax der DEF- und USE-Attribute ist identisch mit der von X3D. Im Unterschied zu X3D, wo Elemente überall in der Datei definiert werden dürfen, müssen in MSML vordefinierte Elemente innerhalb des < library >-Elements definiert werden.*

```
...
<library>
    <data>
        <g:Appearance DEF="blau">
            ...
        </g:Appearance>
    </data>
</library>
...
    <node id="n1" name="Knoten1">
        <data>
            <g:Appearance USE="blau" />
        </data>
    </node>
...
```

Kapitel 5

Das GroIMP-Subsystem

Ein Teilziel der Diplomarbeit war, ein Subsystem für GroIMP zu entwickeln, das auf der Basis eines einheitlichen Formats den Im- und Export von Daten ermöglicht. Ein Vorschlag für ein solches Format ist MSML. Um den Datenaustausch in beliebige Datenformate zu ermöglichen, müssen Datenformate von und nach MSML konvertiert werden können.

Ein bereits in GroIMP existierendes, am Architekturmuster „Pipes & Filter“ [Bus00] orientiertes Filtersystem erlaubt die Erstellung von Filtern, die es im Zusammenspiel miteinander ermöglichen, Daten eines bestimmten Formats in ein anderes zu konvertieren. Diese Daten können dadurch in die GroIMP-interne graphbasierte Datenstruktur eingelesen oder aus dieser heraus exportiert werden.

Eine Möglichkeit um Änderungen an XML-basierten Daten innerhalb eines solchen Filters vorzunehmen, ist die Verwendung von XSLT-Stylesheets. Eine andere Form der Bearbeitung von XML-Daten innerhalb eines Filters kann mithilfe der DOM[1]- oder SAX[2]-Technologien erfolgen.

Das „Document Object Model (DOM)“ ist eine standardisierte, plattform- und programmiersprachenunabhängige Schnittstelle vom W3C, die den Zugriff und die Veränderung von Inhalt, Struktur und Stil eines Dokuments ermöglicht.

Unabhängig von der internen Datenstruktur der jeweiligen DOM-Implementation werden die Dokumente nach außen hin objektorientiert repräsentiert, wobei alle Objekte des Dokuments in eine hierarchische Baumstruktur eingegliedert sind. Im Gegensatz zu SAX wird hier im Speicher eine Baumstruktur aufgebaut, über die traversiert werden kann. Damit ist eine Manipulation oder Neuordnung der Elemente möglich. Teilbäume können extrahiert oder verändert werden. Das DOM-Modell besteht aus Knoten, welche Dokumente, Elemente und Attribute enthalten, die in einer Baumstruktur durch Zeiger miteinander verbunden

[1]Document Object Model - `http://www.w3.org/DOM`, letzter Zugriff am 26.10.2006
[2]Simple API for XML - `http://www.saxproject.org`, letzter Zugriff am 26.10.2006

sind. Dieser Ansatz ist mächtiger als in SAX, aber auch speicherintensiver durch die persistente Abbildung des Baumes im Speicher. Das Parsen dauert hier ebenfalls länger als bei SAX.

Die „Simple API for XML (SAX)“ ist eine standardisierte, ereignisorientierte, plattform- und programmiersprachenunabhängige Schnittstelle zum Parsen von XML-Daten. Die XML-Daten werden von einem SAX-Parser als sequenzieller Datenstrom eingelesen. Beim Eintreten definierter Ereignisse (z. B. Anfang oder Ende eines XML-Elements) werden vorgegebene Funktionen aufgerufen und die geparsten Daten können ausgewertet werden.

Im Gegensatz zum DOM ist SAX zustandslos. Das bedeutet, dass der freie Zugriff auf die Inhalte eines XML-Dokumentes nicht möglich ist. SAX ist wegen seines geringeren Speicherverbrauchs (da hier keine interne Baumstruktur im Speicher aufgebaut werden muss) und aufgrund der Tatsache, dass die Verarbeitung seriell in einem Durchlauf erfolgt, schneller und für die Verarbeitung großer Datenmengen deshalb geeigneter. Die serielle Verarbeitung hat den Vorteil, dass man Dokumente schnell nach Schlüsselwörtern durchsuchen kann oder die Inhalte in der Reihenfolge ihres Erscheinens ausgeben kann. Die serielle Verarbeitung beinhaltet aber auch Nachteile. So ist es mit SAX nicht möglich, die Elemente im Dokument neu zu ordnen und Querverweise innerhalb des Dokuments oder auf externe Dokumente aufzulösen. Ein weiterer Nachteil ist, dass die ausgelösten Ereignisse genauso schnell wieder vergessen werden, wie sie gefunden wurden.

Im Rahmen dieser Arbeit wurden die in Abb. 5.1 sichtbaren, mit durchgezogenen Linien mit Pfeilen verbundenen Filter implementiert. Der durch gestrichelte Linien markierte Filter zum Im- und Export des MTG-Formats konnte aus Zeitgründen nicht implementiert werden. In den folgenden Abschnitten sollen die einzelnen Filter besprochen werden.

5.1 Gründe für die Verwendung von XSLT 2.0

Der Umfang an Funktionen in XSLT 2.0 wurde stark erweitert. Funktionen, die bislang nur durch den Einsatz von Zusatzprojekten[3] oder prozessorspezifischen Erweiterungen verfügbar waren, sind nun Bestandteil der Sprache. Diese verkürzen einige bislang in XSLT 1.0 nötige und umständliche Sprachkonstrukte, z. B. den Umgang mit Gruppen und Zeichenketten, deutlich oder bieten sogar neue Funktionen, die bislang mit XSLT 1.0 so nicht realisierbar sind, zum Beispiel den Umgang mit regulären Ausdrücken. Außerdem besteht in XSLT 2.0 nun die Möglichkeit, eigene Funktionen [DuC03] zu schreiben.

Ein weiteres in XSLT 2.0 eingeführtes Konstrukt sind die temporären Bäume[4].

[3] z. B. EXSLT - `http://www.exslt.org`, letzter Zugriff am 26.10.2006

[4] engl. temporary trees

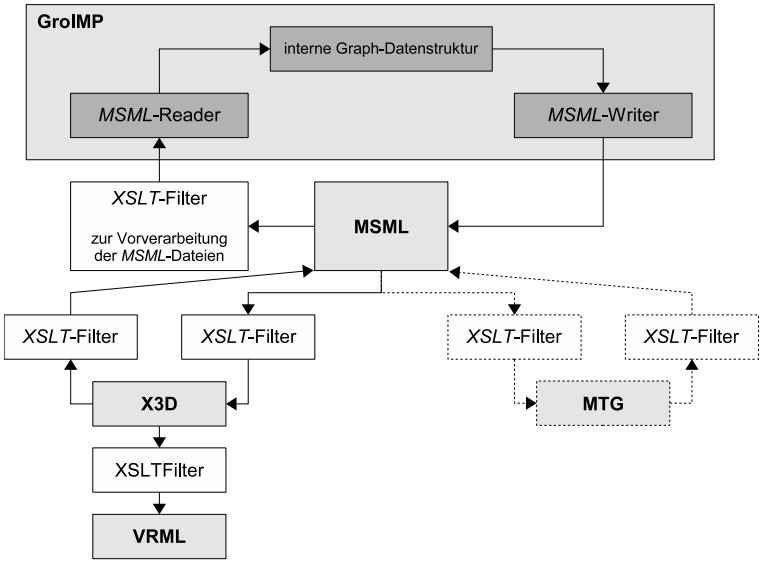


Abbildung 5.1: Das GroIMP-Subsystem zum Im- und Export von Daten auf der Basis von Filtern

Statt wie bisher die Ergebnisse der XSL-Transformationen und XSL-Variablen als Zeichenketten darzustellen, werden die von $< xsl : variable >$, $< xsl : param >$ oder $< xsl : with - param >$ Elementen erzeugten Variablen oder Parameter als eine Menge von Dokumentknoten dargestellt, die temporäre Bäume genannt werden. Diese können als Ergebnis eines Teils der Transformation an den nächsten Teil übergeben werden. Mit diesen temporären Bäumen kann man den Inhalt einer Variable oder eines Parameters mithilfe von XPath-Ausdrücken auswerten und das Abarbeiten des XSL-Baumes modularisieren. Das bedeutet, man kann komplexe Transformationen in verschiedene Module aufteilen und diese iterativ auf das XML-Dokument anwenden.

Nach [Kay04] erweitert XSLT 2.0 die Möglichkeiten des Umgangs mit textbasiertem Input ohne die Notwendigkeit, Konvertierungscode in Java oder einer anderen prozeduralen Programmiersprache zu erstellen. Deshalb kann XSLT 2.0 nicht nur für XML-zu-XML– und XML-zu-Text[5]– Anwendungen, sondern auch für Text-zu-XML–Umwandlungen benutzt werden.

Da der Saxon-Prozessor [Kay] derzeit als Einziger XSLT 2.0, XQuery 1.0[6] und XPath 2.0 vollständig implementiert, wurde er im Rahmen dieses Projektes in GroIMP verwendet.

Es ist sicherlich möglich, die in diesem Projekt notwendigen XSLT-

[5]textbasierte Formate

[6]die XML Query Language [BCF+06] ist derzeit eine Empfehlung des W3C für eine Abfragesprache für XML-Dokumente

Transformationen auch mit den eingeschränkten Möglichkeiten von XSLT 1.0 zu realisieren. Um aber die Komplexität der Stylesheets gering zu halten, und auch einfach aus Gründen des Komforts, fiel die Entscheidung zugunsten von XSLT 2.0. Dabei ziehe ich rückblickend den Schluss, dass ich das Potenzial dieser Sprache noch nicht vollständig genutzt habe, um die gewünschten Ergebnisse zu erzielen. Grund dafür sind unter anderem der Mangel an Entwurfsmustern[7] [GHJ95] [Bus00] für den Entwurf von XSLT-Stylesheets, wie sie in der objektorientierten Programmierung existieren. Entwurfsmuster sind allgemein akzeptierte und effiziente Lösungen für wiederkehrende Problemstellungen in einem Fachgebiet. Sie erleichtern den Einstieg in eine Technologie und verbessern die Qualität der Ergebnisse, da der Prozess, der zur Lösung des Problems bereits einmal durchlaufen wurde, nicht wiederholt werden muss.

In [Kay04] und [BKK01] sind zwar ein paar wenige solcher Entwurfsmuster für XSLT-Stylesheets aufgeführt, ich bin aber der Meinung, dass auf diesem Gebiet noch ein größerer Bedarf besteht. Letztendlich wird sich die Akzeptanz und Verbreitung von XSLT 2.0 auch daran zeigen, ob und in welcher Qualität solche Muster entwickelt und genutzt werden.

5.2 XSLT-Filter zur Vorverarbeitung der MSML-Dateien

Die Hauptaufgabe dieses Filters ist das Einbinden externer Quellen. Das Einbinden externer Elemente in MSML erfolgt mithilfe des *msml:extSrc*-Elements. Das Auflösen dieser externen Referenzen wird, im Subsystem von GroIMP, von einem XSLT-Stylesheet übernommen. Mit diesem werden sämtliche Bibliothekseinträge externer MSML-Dateien und die referenzierten, externen Objekte selbst in einer MSML-Datei zusammengefasst. Das Auflösen der *msml:extSrc*-Elemente erfolgt, indem die externen Elemente mithilfe der XSLT-Funktion *document()* eingelesen, mit neuen IDs versehen und in die XSLT-Ausgabe geschrieben werden. Diese wird dann an den MSML-Reader zur weiteren Bearbeitung übergeben.

Das XSLT-Stylesheet kann, auch unabhängig von GroIMP, zur Zusammenfassung von Informationen benutzt werden, welche über mehrere MSML-Dateien verteilt sind.

Auf Probleme, die im Zusammenhang mit dem Einbinden der externen Quellen auftreten, wird im Folgenden eingegangen.

[7] engl. Design Patterns

5.2.1 Vergabe neuer IDs

Beim mehrfachen Einbinden eines externen Objektes sind die ID des Objektes und seiner Unterelemente nicht mehr eindeutig. Es müssen also neue IDs erstellt werden und alle Referenzen auf diese IDs entsprechend geändert werden.

Neue IDs werden nach folgendem Schema erstellt: Da die ID des einbindenden Elements innerhalb der MSML-Datei eindeutig ist, wird diese einfach als Präfix vor die IDs der Elemente des einzubindenden Elements eingefügt.

Beispiel 5.1 *In diesem Beispiel wird das Einbinden eines externen multiskalierten Objektes, mit den einhergehenden Änderungen der IDs, aus einer MSML-Datei mit dem Namen „externalFile.xml“ demonstriert.*
Ausschnitt aus „externalFile.xml“:

```
<msobject id="ox777" name="Objekt 777">
    <scale id="s1" name="Skale 1">
        <node id="n1" name="Kn1"/>
        <node id="n2" name="Kn2"/>
        <edge source="n1" target="n2"/>
    </scale>
    <scale id="s2" name="Skale 2">
        ...
    </scale>
    <edge source="s1" target="s2" type="refinement"/>
</msobject>
```

Der folgende Teil zeigt den MSML-Quelltext zum Einbinden der oben gezeigten MSML-Datei. Die ID „o1“ wird als Präfix für alle Unterelemente verwendet werden.

```
<msobject id="o1" name="Objekt 1">
    <extSrc src="externalFile.xml"/>
</msobject>
```

Nach dem Einbinden des externen Objekts durch das XSLT-Stylesheet erhalten die Unterelemente neue IDs. Die resultierende MSML-Datei und die geänderten IDs aller Unterelemente seien im Folgenden veranschaulicht:

```
<msobject id="o1" name="Objekt 1">
    <scale id="o1s1" name="Skale 1">
        <node id="o1n1" name="Kn1"/>
        <node id="o1n2" name="Kn2"/>
        <edge source="o1n1" target="o1n2"/>
    </scale>
    <scale id="o1s2" name="Skale 2">
        ...
    </scale>
    <edge source="o1s1" target="o1s2" type="refinement"/>
</msobject>
```

Diese Vorgehensweise hat sich auch bei mehrfach ineinander verschachtelten Einbindungen externer Quellen bewährt.

5.2.2 Probleme bei identischen Einträgen in externen Bibliotheken

Im Zusammenhang mit dem Einbinden von MSML-Elementen aus einer externen Datei, die Eigenschaftselemente der Bibliothek dieser externen Datei verwenden, existiert folgendes Problem.

Haben die verwendeten Eigenschaftselemente dieser externen Bibliothek die gleichen Namen wie in der aktuellen MSML-Datei, dann gibt es verschiedene Möglichkeiten, diesen Konflikt aufzulösen.
Die folgenden Vorgehensweisen sind möglich:

1. Man überprüft die Typen und Inhalte der gleichnamigen Eigenschaftselemente. Bei Unterschieden werden das einzubindende Eigenschaftselement und seine Referenzen umbenannt. Das Problem ist hierbei, die Gleich- oder Verschiedenartigkeit von Eigenschaftselementen mithilfe von XSLT festzustellen. Außerdem konnte bislang keine befriedigende Lösung für das Ändern der Namen in den Bibliothekselementen und vor allem in den Referenzen darauf gefunden werden, da XSLT zustandslos arbeitet und die Zwischenspeicherung solcher Informationen nicht direkt möglich ist.

2. Man legt Prioritäten für die Bibliothekselemente fest, die der Schachtelungstiefe der Einbettungsaufrufe (msml:extSrc-Elemente) entsprechen. Dabei besitzen Bibliothekselemente der aktuellen MSML-Datei oberste Priorität. Gleichnamige Bibliothekselemente aus den Bibliotheken externer Dateien sind durch

diese zu ersetzen. Problem hierbei ist der Verlust der Prioritäten bereits eingebundener Bibliothekselemente. Dadurch ist beim mehrfach rekursiven Einbinden von Bibliothekselementen, ein Einbinden von Bibliothekselementen unterster Priorität bereits erfolgt. Nach dem Aufsteigen aus der Rekursion ist, beim Einbinden weiterer Bibliothekselemente höherer Priorität, ein Vergleich der Prioritäten zwischen bereits eingebundenen und einzubindenden Bibliothekselementen nicht mehr möglich.

3. Man benutzt das gleiche Schema wie zur Umbenennung der IDs der Elemente von eingebundenen Objekten (siehe Abschnitt 5.2.1). Die ID des einbindenden Elements wird als Präfix vor der ID benutzt. Der Nachteil dieser Lösung liegt darin, dass beim mehrfachen Einbinden von Elementen einer externen Datei für jedes Einbinden neue Namen für die Bibliothekselemente entstehen, obwohl es sich um die gleichen Elemente handelt.

4. Man löst die Bibliotheksreferenzen vor dem Import auf, indem die Referenzen auf Bibliothekselemente durch die Inhalte der Bibliothekselemente ersetzt werden. Der Nachteil dieser Lösung liegt darin, dass der Bibliothekseintrag verloren geht und jedes Objekt eine eigenständige Eigenschaftsinstanz erhält, statt nur einer Referenz auf eine gemeinsame Eigenschaft. Das hat einerseits zur Folge, dass Änderungen an der Eigenschaft eines Objektes sich nicht mehr automatisch auf die anderen übertragen. Andererseits wirkt sich dies bei einer großen Anzahl von Objekten mit gleichen Eigenschaften negativ auf die Leistungsanforderungen (z. B. Speicher) aus. Grund dafür ist, dass statt einer großen Anzahl von Referenzen auf eine Eigenschaft nun eine große Anzahl von eigenständigen Instanzen dieser Eigenschaft nötig ist.

Da keine der aufgeführten Methoden eine befriedigende Lösung darstellt und aufgrund von Zeitmangel ist in der aktuellen Version des XSLT-Filters zur Vorverarbeitung keine der aufgeführten, sondern nur eine provisorische Lösung implementiert. Dabei werden die Verweise auf externe Quellen rekursiv bei ihrem Auftreten durchlaufen und alle Bibliothekseinträge in einem temporären Ergebnisbaum gesammelt. Anschließend werden die Einträge nach ihren Namen sortiert und jeweils das erste Bibliothekselement jedes Namens in die Ergebnisbibliothek aufgenommen.

5.3 MSML-Reader und MSML-Writer

Der MSML-Reader bekommt vom vorverarbeitenden XSLT-Filter eine MSML-Datei als DOM-Baum übergeben, in der alle von ihr referenzierten externen Quellen bereits eingebunden sind. Dieser wird traversiert und die entsprechenden GroIMP-Datenstrukturen werden erstellt.

Der MSML-Writer traversiert den GroIMP-internen Graphen und erstellt einen DOM-Baum der entsprechenden MSML-Repräsentation, der dann als XML-Datei ausgegeben wird.

5.4 Umwandlung zwischen MSML und X3D

Die Umwandlung der Grundstrukturen von MSML nach X3D ist relativ einfach, jedoch das Umwandeln der GroIMP-spezifischen Eigenschaftselemente enthält ein paar Probleme, auf die im Folgenden eingegangen wird. Obwohl die GroIMP-spezifischen Eigenschaftselemente X3D-Syntax benutzen und deshalb die Umwandlung einfach sein sollte, gibt es Probleme. So wird in Anhang A.10.1 die Erweiterung des GroIMP-spezifischen Eigenschaftselements *g:Shape* um zusätzliche Parameter beschrieben, um die, in GroIMP mögliche, Beschreibung unterschiedlicher Ausgangslagen der 3D-Primitive ausdrücken zu können (siehe Abb. A.1). Dies stellt bei der Konvertierung von X3D nach MSML und den GroIMP-spezifischen Attributen kein Problem dar, da die zusätzlichen Parameter einfach mit den in X3D gültigen Werten belegt werden. Beim umgekehrten Fall, der Konvertierung von MSML nach X3D, können diese zusätzlichen Parameter aber nicht äquivalent nach X3D übertragen werden. Das bedeutet, es muss ein zusätzlicher Transformationsknoten eingefügt werden, der das lokale Koordinatensystem eines 3D-Primitivs, vor der eigentlichen Transformation, in die Position bringen, an der das 3D-Primitiv startet (entspricht dem Parameter startpos (siehe Anhang A.10.1)). Diese Transformation kann allerdings mit der eigentlichen Transformation zusammengefasst werden. Ein weiterer Transformationsknoten muss nach der eigentlichen Transformation erstellt werden, um das 3D-Primitiv in die Position zu bringen, von der aus das nachfolgende 3D-Primitiv startet (entspricht dem Parameter endpos (siehe Anhang A.10.1)). Um die Struktur des in X3D verwendeten Szenengraphen zu erstellen, muss bei der Umwandlung von MSML anders vorgegangen werden.

Bei dem hierarchischen Szenengraphen, wie in X3D (siehe Bsp. 5.2), werden Kanten, die einen Knoten mit anderen Knoten verbinden, dadurch modelliert, dass die nachfolgenden Knoten als Kinder des Knotens innerhalb des Knotenelements dargestellt werden.

In MSML befinden sich alle Knoten eines Graphen auf der gleichen Ebene. Die Kanten werden nicht durch die Struktur innerhalb der XML-Datei modelliert, sondern explizit als Elemente angegeben (siehe Bsp. 5.2). Deshalb muss bei der Konvertierung zuerst der Wurzelknoten eines Graphen in MSML gefunden werden und von diesem ausgehend, den Kanten folgend, die hierarchische Struktur des Graphen erstellt werden.

Beispiel 5.2 *MSML-Graphstruktur:*

```
<msobject>
    <scale>
        <node id="Kn1">
            <data>
                <g:Shape>
                    <x3d:Cylinder />
                </g:Shape>
            </data>
        </node>
        <node id="Kn2">
            <data>
                <g:Shape>
                    <x3d:Cylinder />
                </g:Shape>
            </data>
        </node>
        <node id="Kn3">
            <data>
                <g:Shape>
                    <x3d:Cylinder />
                </g:Shape>
            </data>
        </node>
        <edge source="Kn1" target="Kn2" type="successor"/>
        <edge source="Kn2" target="Kn3" type="successor"/>
        <edge source="Kn1" target="Kn4" type="branch"/>
    </scale>
</msobject>
```

X3D-Graphstruktur:

```
<Transform>                 // entspricht Kn1
    <Transform>             // entspricht Kn2
        <Transform>         // entspricht Kn3
            <Shape>
                <Cylinder />
            </Shape>
        </Transform>
        <Shape>
            <Cylinder />
```

```
        </Shape>
    </Transform>
    <Transform>              // entspricht Kn4
        <Shape>
            <Cylinder />
        </Shape>
    </Transform>
    <Shape>
        <Cylinder />
    </Shape>
</Transform>
```

In der momentanen Implementierung wird beim Konvertieren eines multiskalierten Objektes von MSML nach X3D nur die durch das *showScale*-Attribut des *msobject*-Elements festgelegte Skale nach X3D exportiert. Ist kein *showScale*-Attribut vorhanden, wird einfach die erste definierte Skale exportiert. Denkbar ist auch, das *x3d:LOD*-Element zu benutzen, um die verschiedenen Skalen eines Objektes, abhängig von der Entfernung zum Objekt, darzustellen. Möglich ist aber auch das *x3d:Switch*-Element zu benutzen, um dem Betrachter einer X3D-Szene die Auswahl einer Skale zu überlassen.

Die Umwandlung von X3D nach MSML wird im Rahmen dieser Arbeit nur rudimentär, für eine Auswahl an X3D-Sprachelementen, implementiert. Ziel ist es,

1. die Struktur der, in einer X3D-Datei beschriebenen Szenen oder Objekte nach MSML zu übertragen, sowie

2. die in X3D beschriebenen Informationen zum Aussehen von Objekten durch den vom GroIMP-spezifischen MSML-Attribut *g:Appearance* darstellbaren Teil nach MSML zu übertragen.

Dabei ist man beim derzeitigen Stand der Implementierung von GroIMP und MSML auf den von den MSML-Importfiltern einlesbaren Teil von GroIMP-spezifischen Attributen eingeschränkt. Eine Erweiterung dieser Attribute zur Beschreibung von Formen und dem Aussehen von Objekten würde natürlich auch den von X3D nach MSML abbildbaren Teil der X3D-Sprachspezifikation erweitern.

5.5 Umwandlung zwischen X3D und VRML

Für die Konvertierung von X3D nach VRML existiert bereits eine XSLT-Vorlage (*X3dToVrml97.xsl*), die im Paket der Software *X3DEdit* [Bru06] enthalten ist. Diese kann einfach in das GroIMP-Filtersystem integriert werden, da alle *X3DEdit*-Dateien der „Open-Source und Public-Domain“Lizenz unterliegen.

Der *VRML97 to X3D Translator*[8] ist ein Java-Programm zur Übersetzung zwischen dem VRML97- und X3D-Format. Die Übersetzung basiert auf der Spezifikation der „*x3d-compact.dtd*“und „*X3dToVrml97.xsl*“. Dieses Programm kann die Basis für die Entwicklung eines GroIMP-Filters zur Umwandlung von VRML97-Dateien nach X3D bilden.

Da VRML97 nicht XML-basiert ist, ist der Einsatz von XSLT zur direkten Transformation bislang nicht möglich. Mit den erweiterten Möglichkeiten von XSLT 2.0 gibt es, wie bereits in Abschnitt 5.1 angedeutet, Beispiele dafür, dass auch nicht XML-basierte Formate mit XSLT 2.0 nach XML transformiert werden können. Ein Beispiel dazu findet sich auf Seite 703 in [Kay04].

5.6 Umwandlung zwischen MSML und MTG

Wie schon angesprochen, konnte dieser Teil aus Zeitgründen nicht implementiert werden. Da MTG eines der wenigen Formate, wenn nicht das bislang Einzige ist, das die Erfassung multiskalierter Strukturen ermöglicht, ist eine Umwandlung von und nach MSML interessant. Da MTG, wie auch VRML, nicht XML-basiert ist, kann eine Umwandlung von MSML nach MTG durch ein XSLT-Stylesheet realisiert werden. Der umgekehrte Weg von MTG nach MSML kann nach dem, in Abschnitt 5.5 zitierten Prinzip erfolgen.

[8] `http://ovrt.nist.gov/v2_x3d.html`, letzter Zugriff am 26.10.2006

Kapitel 6

MSML-Beispiele in GroIMP

In diesem Kapitel sind Abbildungen der GroIMP-Benutzeroberfläche aufgeführt, die an verschiedenen Stellen im Text der Diplomarbeit referenziert werden.
Dabei gelten in der in GroIMP verfügbaren Graphenansicht folgende Konventionen:

- Eine durchgezogene Linie mit Pfeil stellt eine Nachfolgerkante vom Typ *successor* dar.
- Eine Strich-Punkt-Linie mit Pfeil stellt eine Verzweigungskante vom Typ *branch* dar.
- Eine gestrichelte Linie mit Pfeil stellt eine Verfeinerungskante vom Typ *refinement* dar und wird zusätzlich mit einem / gekennzeichnet.
- Kanten mit mehreren Attributen werden durch eine kommagetrennte Aufzählung von Symbolen der Attribute dargestellt. Die in den Abbildungen vorkommenden Symbole werden im Folgenden aufgeführt:
  - A ... markiert eine Kante, die den Knoten eines multiskalierten Knoten oder Objekts mit dem Knoten einer seiner *verfügbaren Skalen* verbindet.
  - B ... markiert eine Kante, die den internen Endknoten eines multiskalierten Knoten mit dem Knoten verbindet, der über eine Kante mit dem multiskalierten Knoten verbunden ist.
  - > ... markiert die Kante, welche die Verbindung zwischen einem multiskalierten Knoten oder Objekt mit dem Knoten der *gewählten Skale* verbindet.
- Knoten mit hellen (beige) Kästchen besitzen eine grafische Darstellung.
- Knoten mit dunklen (ocker) Kästchen besitzen keine grafische Darstellung.

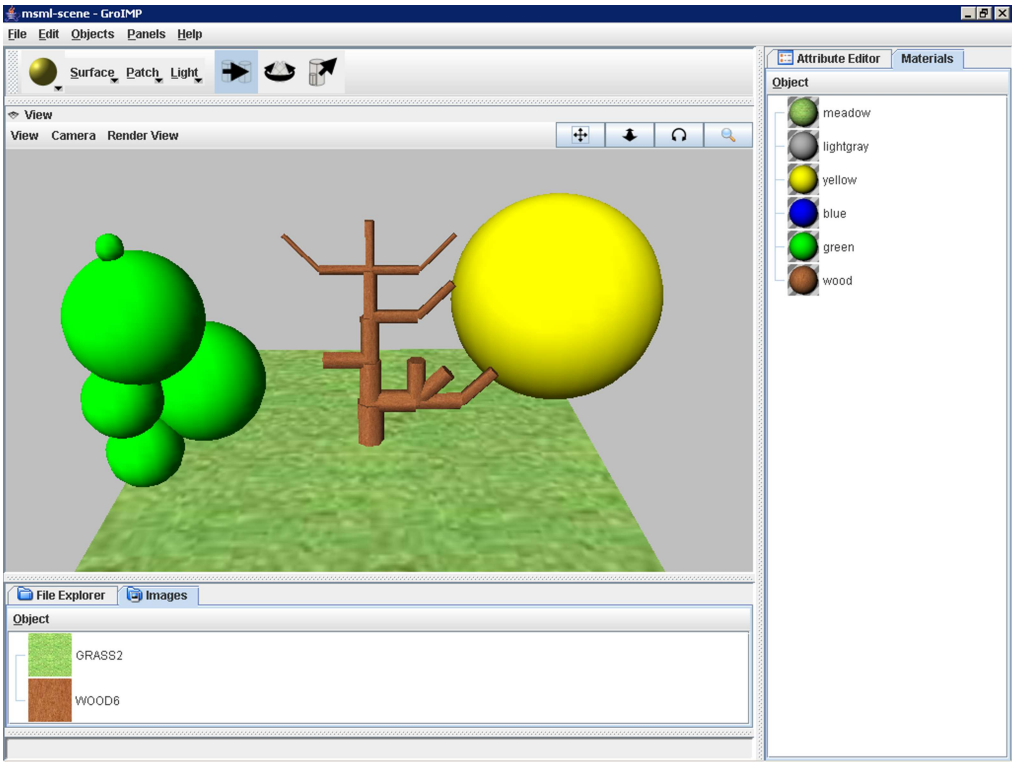
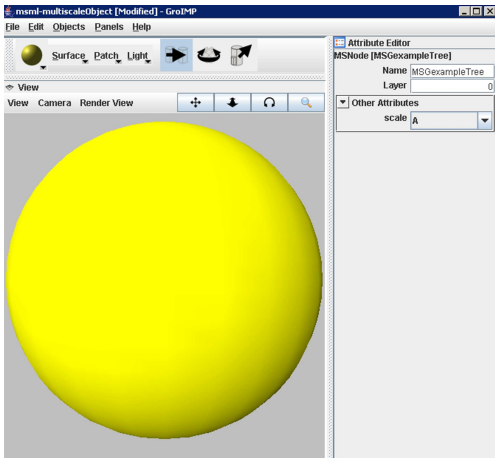
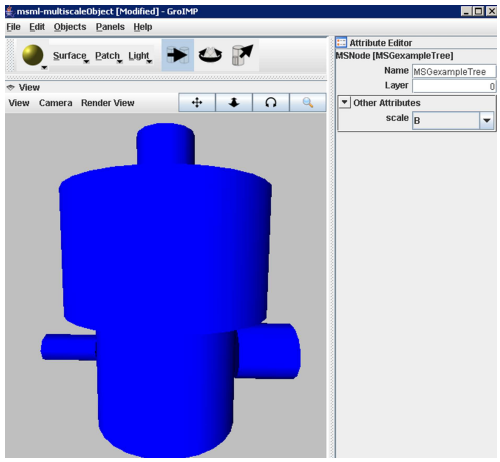


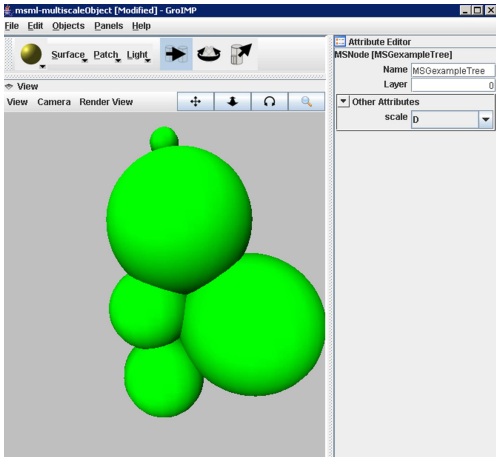
Abbildung 6.1: Darstellung der Szene aus Quelltext B.3 in GroIMP. Sie besteht aus zwei multiskalierten (siehe Abb. 6.2) und einem nicht multiskalierten Objekt (baumartige Struktur mit Holztextur), sowie einem Bodenelement mit Grastextur.
Im „Materials“-Menü (rechtes Menü in der Abbildung) von GroIMP sind die aus der Bibliothek der MSML-Datei importierten Materialien zu sehen.
Im „Images“-Menü (unteres Menü in der Abbildung) sind die von den Objekten verwendeten Texturen zu sehen.



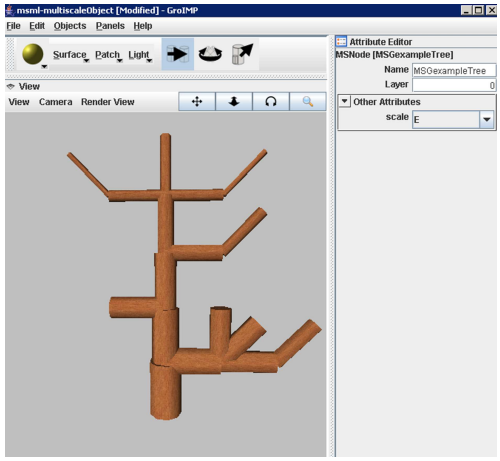
(a) Skale A



(b) Skale B



(c) Skale D



(d) Skale E

Abbildung 6.2: Darstellung der verschiedenen Skalen des multiskalierten Objektes aus Quelltext B.2 in GroIMP. Der multiskalierte Graph, der die Beziehungen zwischen den Modulen der verschiedenen Skalen darstellt, ist in Abb. 2.3 zu sehen.
Im „Attribute Editor“ (rechtes Menü in den Abbildungen) von GroIMP sieht man in der Sektion „Other Attributes“ die Schaltfläche zur Auswahl der aktuellen Skale.

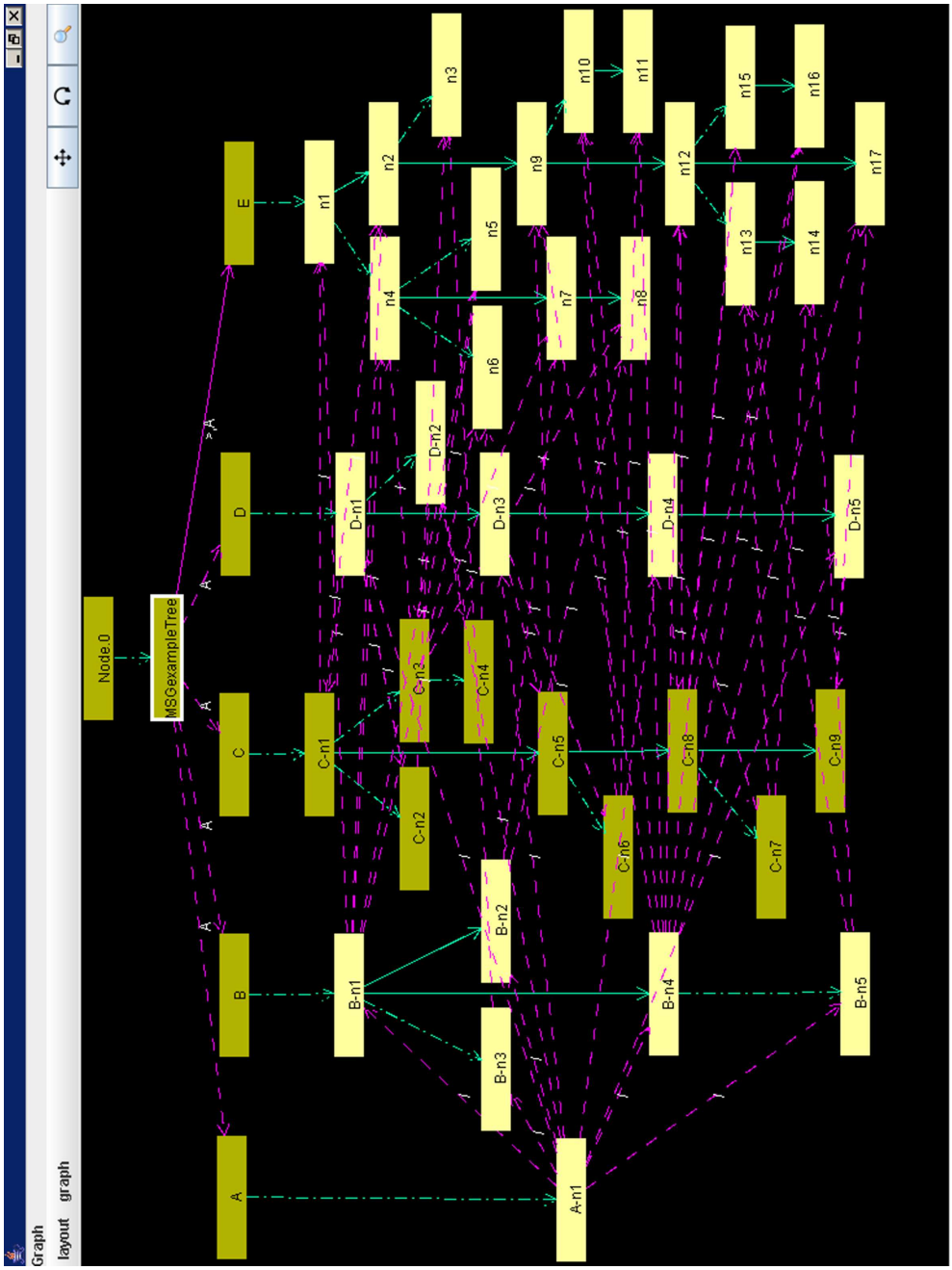
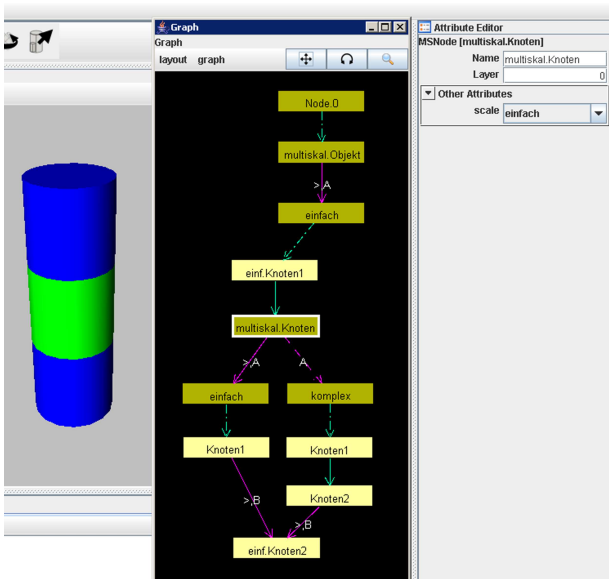
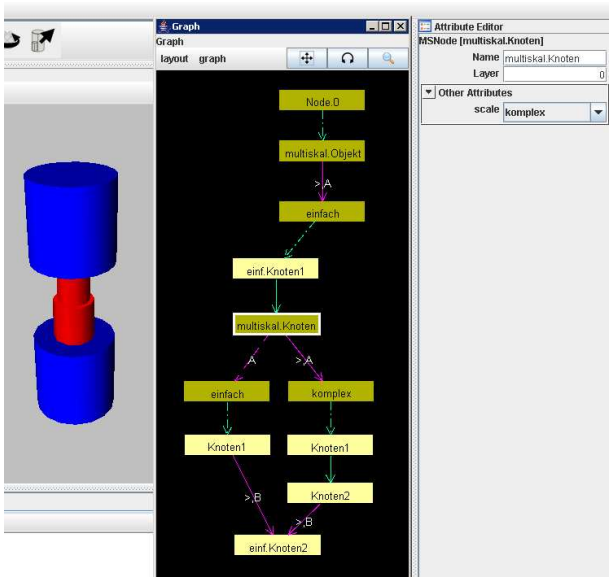


Abbildung 6.3: Darstellung der internen GroIMP-Graphrepräsentation des multiskalierten Objekts aus Abb. 6.2 und Quelltext B.2. *Node.0* ist der Wurzelknoten der gesamten Szene. *MSGexampleTree* ist der Knoten, der das multiskalierte Objekt repräsentiert. Dieser ist über die mit einem **A** gekennzeichneten Kanten mit den Knoten seiner *verfügbaren Skalen* (*A, B, C, D* und *E*) verbunden. Die Kante, welche die Verbindung zur aktuell *gewählten Skale* (*E*) darstellt, ist mit einem > gekennzeichnet. Die Skalenknoten sind mit dem Knoten verbunden, der die Wurzel des Baumgraphen darstellt, der das multiskalierte Objekt in der jeweiligen Skale beschreibt. Für eine Beschreibung der Kantentypen und Farben siehe an den Anfang dieses Kapitels.



(a) einfache Skale



(b) komplexe Skale

Abbildung 6.4: Funktionsweise multiskalierter Knoten aus Quelltext B.4 in GroIMP. Gezeigt wird ein aus drei Knoten bestehendes Objekt (*multiskal.Objekt*). Dabei handelt es sich um zwei einfache Knoten (*einf.Knoten1, einf.Knoten2*) und einen multiskalierten Knoten (*multiskal.Knoten*). Die zwei einfachen Knoten sind durch jeweils einen blauen Zylinder dargestellt. Der multiskalierte Knoten wird in einer einfachen Skale (ein grüner Zylinder) und in einer komplexen Skale (zwei rote Zylinder) dargestellt. In der Graphdarstellung erkennt man die mit **A** gekennzeichneten Kanten, welche die Verbindung zu den *verfügbaren Skalen* darstellen. Die zur aktuell *gewählten Skale* verlaufende Kante, ist mit einem > gekennzeichnet.
Die mit **B** gekennzeichneten Kanten markieren den Übergang vom multiskalierten Knoten zum nachfolgenden einfachen Knoten.

Kapitel 7

Fazit

In dieser Arbeit wurde mit MSML ein Entwurf für ein XML-basiertes Datenformat zur Erfassung multiskalierter Graphstrukturen erarbeitet. Es wurde außerdem eine Möglichkeit geschaffen, Daten von GroIMP in dieses Datenformat zu im- und exportieren. Weiterhin wurden XSLT-Filter für eine eingeschränkte Konvertierung von MSML nach X3D und umgekehrt geschaffen.

Mit MSML wurde ein vielseitig einsetzbares Datenformat geschaffen, das die Beschreibung von Objekten aus unterschiedlichen Perspektiven und Modellen zulässt. Damit geht das Format einen Schritt weiter als bisherige Formate zur Abbildung hierarchischer Strukturen.

Zudem wird den Anwendern die Möglichkeit geboten, Metadaten selbst zu definieren. Diese Eigenschaft scheint besonders wichtig, da sich aufgrund räumlicher, zeitlicher oder sprachlicher Trennung von Institutionen bei der Arbeit an wissenschaftlichen Themen unterschiedliche Vokabulare für ähnliche Eigenschaften herausbilden können. Infolge dessen gestaltet sich ein Datenaustausch oft schwierig. Die Kennzeichnung der Herkunft des Vokabulars und der Definition der Eigenschaften durch Namensräume erleichtert das Auffinden der zugehörigen Dokumentation und ermöglicht eine einfachere Konvertierung von Vokabularen. Die Verwendung von XML führt außerdem dazu, dass eine Konvertierung von Vokabularen und Eigenschaften durch z. B. XSLT einfach zu realisieren ist.

Die Möglichkeit der Verwendung von Elementen aus externen Quellen innerhalb einer MSML-Datei ermöglicht eine dezentrale Datenhaltung und die Erstellung komplexer Szenen mit mehreren Objekten bei geringer Dateigröße und besserer Übersicht. So kann z. B. ein Katalog von MSML-Dateien verschiedener Objekte in unterschiedlichen Skalen in einem Netzwerk vorhanden sein. Ein Benutzer kann nun eine komplexe Szene bestehend aus mehreren dieser Objekte aufbauen, indem er lediglich auf die Elemente des Katalogs innerhalb der MSML-Datei verweist (siehe z. B. Quelltext B.3). Diese Datei ist wesentlich kompakter und übersichtlicher, als wenn sich die umfangreichen Beschreibungen der Objekte ebenfalls in der Datei be-

finden würden.
Die folgenden Punkte führen die Bereiche auf, an denen weiter gearbeitet werden muss, um MSML weiter zu entwickeln.

- ***Überarbeitung der GroIMP-spezifischen Eigenschaftselemente***
  Bei der Konvertierung der grafischen Beschreibungen von Objekten zwischen GroIMP, MSML und X3D wurde deutlich, dass zwar die Möglichkeit besteht, Objekte unterschiedlich zu beschreiben, dies aber nicht von der Notwendigkeit einer Standardisierung abhalten darf, da einige Eigenschaften nicht verlustfrei zwischen den verschiedenen existierenden Formaten konvertiert werden können.

  Um also als zentraler Bestandteil eines GroIMP-Subsystems zur Konvertierung zwischen verschiedenen Datenformaten eines Anwendungstyps (z. B. von 3D-Grafikformaten) einsetzbar zu sein, muss MSML eine Menge von Eigenschaftselementen besitzen, welche die Eigenschaften der zu konvertierenden Formate abbilden können. Das für die Beschreibung der GroIMP-spezifischen Eigenschaftselemente von mir ausgewählte X3D erweist sich dafür in bestimmten Bereichen (z. B. der Beschreibung des Aussehens und der Lage von Objekten) derzeit als nicht mächtig genug (siehe die Bemerkungen über die Abbildung von Shadern in Abschnitt A.10.1 und die Probleme bei der unterschiedlichen Definition der Ausgangsposition von 3D-Objekten und deren Nachfolgern in Abschnitt A.10.1). So bleibt auch hier nur die Möglichkeit, einen Vorschlag für eine eigene XML-Syntax zur flexiblen Beschreibung des Aussehens von Objekten zu entwickeln bzw. die weitere Entwicklung im Bereich der XML-basierten Sprachen aus dem Bereich der 3D-Grafik abzuwarten.

  Bei der Definition von Eigenschaftselementen ist auch die Möglichkeit zu untersuchen, RDF [HSB06][1] zur Beschreibung der Eigenschaften von Objekten zu benutzen.

- ***Implementierung zusätzlicher Konvertierungsfilter***
  Um eine Verbreitung von MSML zu erreichen, ist die Schaffung von Konvertierungsmöglichkeiten in weitere Datenformate wichtig. Speziell eine Konvertierung zwischen MSML und dem am Anfang von Kapitel 3 aufgeführten, von der AMAPMOD-Software verwendeten, textbasierten Format zur Codierung von MTGs ist wichtig, um die Fähigkeiten von MSML zur Erfassung multiskalierter Strukturen zu demonstrieren.

- ***Entfernung der Einschränkung auf baumartige Graphstrukturen***
  In diesem Projekt wurde die Einschränkung auf baumartige Graphstrukturen

[1] Resource Description Framework - eine vom W3C vorgeschlagene Sprache zur Definition von Metadaten

aus dem Modell in [GC98] übernommen. Diese Einschränkung basiert auf der Tatsache, dass dieses Modell zur Beschreibung topologischer Strukturen von Pflanzen geschaffen wurde. Diese Einschränkung behindert aber die Erfassung allgemeiner multiskalierter Objekte. So können keine Kanten definiert werden, die zyklische Beziehungen modellieren. Durch Anpassungen in MSML kann dieses Ziel aber erreicht werden, um zum Beispiel auch das in Abschnitt 2.4.2 geschilderte Problem der Nichteindeutigkeit des Ziels eingehender Kanten bei einem multiskalierten Knoten zu beheben.

- ***Optimierung der Implementierung***
  Heutige XSLT-Prozessoren behalten während einer Transformation alle Daten im Speicher. Dies stellt bei Transformationen von großen Datenbeständen hohe Anforderungen an die Speicherausstattung eines Rechners, und die Ausführung kann sehr zeitintensiv sein. Deshalb muss untersucht werden, bis zu welcher Komplexität einer MSML-Datei der Einsatz von XSLT für die Konvertierung in andere Formate geeignet ist. In jedem Fall bietet die jetzige Implementierung noch ausreichend Möglichkeiten zur Optimierung, da längst nicht alle von XSLT 2.0 bereitgestellten Techniken angewendet wurden. So besteht weiterhin noch keine zufriedenstellende Lösung für das Zusammenführen und Vergleichen von Bibliothekseinträgen aus verschiedenen MSML-Dateien.

  Auch die Möglichkeiten des Einsatzes von SAX statt des DOM-Modells müssen bei einer Optimierung der Implementierung der MSML-Reader und MSML-Writer nochmals geprüft werden.

- ***Erweiterung der Benutzeroberfläche von GroIMP***
  GroIMP bietet mit dem derzeitigen Stand der Implementierung grundlegende Fähigkeiten zum Im- und Export multiskalierter Strukturen. Um aber mit Objekten dieser Art arbeiten zu können, müssen der Benutzeroberfläche noch einige Funktionen hinzugefügt werden. Dazu gehört zum Beispiel eine Möglichkeit, zu einem ausgewählten Primitivobjekt den zugehörigen multiskalierten Oberknoten zu finden, falls ein solcher verfügbar ist, um eine andere der für das gewählte Objekt verfügbaren Skalen auszuwählen. Ebenfalls nötig sind Funktionen zur Auswahl und Bearbeitung von gruppierten Objekten.

- ***Entwicklung von Layoutalgorithmen für multiskalierte Graphen***
  Die zurzeit verfügbaren hierarchischen Layoutalgorithmen in der 2D-Graph-Ansicht von GroIMP führen momentan nicht zu einer übersichtlichen Darstellung des GroIMP-internen Graphen, wenn er multiskalierte Objekte enthält. Das liegt vermutlich an der Tatsache, dass die Graphen innerhalb der Skalen Bäume sind, die Verfeinerungskanten zwischen den Knoten der verschiedenen Skalen aber durchaus Graphstrukturen mit Zyklen zufolge haben können.

Es besteht also die Notwendigkeit, einen angepassten Layoutalgorithmus zur übersichtlicheren Darstellung multiskalierter Graphstrukturen zu entwickeln. Von den derzeit in GroIMP zur Verfügung stehenden Layoutalgorithmen erreicht der mit dem Namen „*square*“ versehene Algorithmus mit gesetzter *fTop-Down*-Option noch die übersichtlichste Darstellung der Knoten.

Abschließend bleibt zu ergänzen, dass das Thema der multiskalierten Beschreibung von Objekten offenbar bislang kaum bearbeitet wurde. Die Anzahl der verfügbaren Dokumente, die sich mit dem Thema beschäftigen [GC98][GCS99] [God00], ist verhältnismäßig klein. Außerdem stammt die Mehrzahl der Dokumente aus der selben Quelle, was die Sichtweise auf bestimmte Aspekte eingrenzt. Die Suche nach anderen Quellen und die kritische Auseinandersetzung mit der in den obengenannten Quellen angeführten Definition, dass keine Beziehungen zwischen Elementen nichtvergleichbarer Skalen modelliert werden können, haben einen beträchtlichen Umfang an Zeit in Anspruch genommen. Dies ist einer der Gründe, warum die ursprünglich geplanten Filter zur Konvertierung zwischen MSML und dem MTG-Format nicht mehr realisiert werden konnten. Auch die Beschreibung von nur einer Auswahl der von GroIMP definierten geometrischen Primitivobjekte, durch die GroIMP-spezifischen Eigenschaftselemente und die damit verbundene begrenzte Fähigkeit zum Im- und Export nach X3D/MSML ist darauf zurückzuführen.

Kapitel 8

Zusammenfassung

In dieser Diplomarbeit wurde ein Subsystem für die am Lehrstuhl Praktische Informatik/Grafische Systeme entwickelte GroIMP-Plattform [Kni06] geschaffen, welches den Austausch graphbasierter, multiskalierter Strukturen auf Basis eines XML-Datenformats ermöglicht.

Ein von C. Godin und Y. Caraglio entworfenes Modell [GC98] zur Beschreibung multiskalierter Graphen (=Multiscale Tree Graph (MTG)) bildet die Grundlage der Beschreibung multiskalierter Graphstrukturen. Darin wird über die Beschreibung von Objekten aus einer einzelnen Modell-Perspektive hinausgegangen. Durch die Definition mehrerer Skalen eines Objektes wird eine detailliertere Beschreibung auf der Basis verschiedener Modelle erreicht.

Die Vergleichbarkeit von Skalenelementen stellt das wichtigste Kriterium beim Aufbau von Beziehungen zwischen den Elementen verschiedener Skalen eines multiskalierten Objekts dar. So lassen sich vergleichbare Skalen in hierarchischen Graphstrukturen erfassen, während sich zwischen Elementen nicht vergleichbarer Skalen keine Beziehungen modellieren lassen. Die beim Umgang mit multiskalierten Strukturen auftretenden Probleme wurden erläutert.

Bereits existierende XML-basierte Datenformate zur Erfassung graphbasierter Strukturen wurden auf ihre Eignung als Basis eines multiskalierten Datenformats geprüft.

Verschiedene Möglichkeiten zum Einbinden externer XML-Quellen in ein XML-Dokument wurden untersucht. Ein XSLT-Filter wurde erstellt, um einen über mehrere MSML-Dokumente verteilten Datenbestand in einer Datei zusammenzuführen.

Mit MSML (=Multiscale Modeling Language) wurde ein XML-basiertes Format zur Erfassung multiskalierter Strukturen entworfen. MSML bildet den Kern eines GroIMP-Subsystems für den Datenaustausch mit anderen Formaten. Dafür wurden XSLT-Filter zur Umwandlung zwischen MSML und X3D erzeugt.

Anhand von Beispielen wurde die Möglichkeit demonstriert, grafische Objekte auf mehreren Skalenebenen in GroIMP darzustellen.

Anhang A

MSML-Formatbeschreibung

In den folgenden Abschnitten werden die Elemente des XML-Schemas und damit der Sprachbestandteile von MSML (Multiscale Modeling Language) vorgestellt. Die Attribute der XML-Elemente werden mit einem vorangestellten @ gekennzeichnet. Danach folgen in Klammern Angaben über die Verwendungsart (notwendig oder optional) und den Datentyp[1].
Der XML-Namensraum von MSML lautet: `http://grogra.de/msml`.
Dieser wird im Folgenden mit dem Präfix *msml:* gekennzeichnet.

A.1 msml:msml

Das Wurzelelement einer MSML-Datei.

Als erstes Kindelement ***kann*** es ein Bibliothekselement *msml:library* enthalten. Als nächste Kindelement ***muss*** es ein *msml:msobject*-Element zur Beschreibung einzelner oder ein *msml:group*-Element zur Beschreibung einer Gruppe von multiskalierten Objekten besitzen. Nicht-multiskalierte Objekte werden durch ein multiskaliertes Objekt *msml:msobject* mit nur einer Skale *msml:scale* beschrieben.

- @version (notwendig) Gibt die verwendetet Version von MSML an.

A.2 msml:library

Dieses Element dient zur Erfassung von vordefinierten und wiederverwendbaren Elementen. Es ***kann*** als Kindelemente ein *msml:data* Element, zur Definition von beliebigen Eigenschaftselementen, und eine beliebige Anzahl von multiskalierten Objekten (bzw. multiskalierten Knoten) *msml:msobject* enthalten. Jedes Eigenschaftselement und jedes multiskalierte Objekt, welches innerhalb der Bibliothek definiert

[1] mit dem Namensraum xs: gekennzeichnete Datentypen sind XML-Standarddatentypen [BM04]

wird, ***muss*** ein Attribut namens *DEF* besitzen. Nur innerhalb des *msml:library*-Elements darf das *DEF*-Attribut verwendet werden.

- @DEF (notwendig, xs:ID) Das *DEF*-Attribut enthält den Namen, mit dem der Bibliothekseintrag später referenziert wird. Das *DEF*-Attribut hat den XML-Schema-Datentyp xs:ID. Somit muss es im gesamten XML-Dokument eindeutig sein. Für die Erstellung von Namen gelten die gleichen Regeln, die zur Erstellung vom XML-Schema-Datentyp xs:NCName[2] definiert werden, da sich xs:ID von xs:NCName ableitet.

Eigenschaftselemente und *msml:msobject*-Elemente außerhalb von *msml:library* können unter Verwendung des *USE*-Attributes auf die definierten Elemente zugreifen.

- @USE (optional, xs:IDREF) Bei Nutzung des *USE*-Attributs, wird auf ein in der Bibliothek *msml:library* definiertes Element verwiesen, das an dieser Stelle benutzt wird. Das *USE*-Attribut darf nur außerhalb des *msml:library*-Elementes verwendet werden.

A.3 msml:data

Das *msml:data*-Element dient zum Definieren von Eigenschaften des ihm übergeordneten Elements. Es kann beliebige Eigenschaftselemente enthalten (siehe Abschnitt A.10).

A.4 msml:group

Dieses Element dient zum Gruppieren von multiskalierten Objekten und anderen Gruppen. Es ***muss*** mindestens eines der folgenden Elemente als Kinder enthalten. Genau ein *msml:data* Element, zur Definition von Eigenschaftselementen für diese Gruppe oder/und eine beliebige Anzahl von multiskalierten Objekten *msml:msobject* oder weiteren Gruppenelementen *msml:group*.

- @id (notwendig, xs:ID) Das *id*-Attribut enthält einen im gesamten Dokument eindeutigen Bezeichner, der zur Identifizierung des Elements dient.

- @name (optional, xs:string) Das *name*-Attribut wird zur Vergabe eines Gruppennamen benutzt.

[2] `http://www.w3.org/TR/REC-xml-names/#NT-NCName`, letzter Zugriff am 26.10.2006

A.5 msml:msobject

Das *msml:msobject*-Element wird zur Beschreibung von multiskalierten Objekten benutzt. Nicht-multiskalierte Objekte werden durch ein multiskaliertes Objekt mit nur einer Skale *msml:scale* beschrieben. Als erstes Kindelement ***kann*** es ein *msml:data*-Element besitzen. Als nächstes Kindelement ***muss*** es entweder ein Verweiselement *msml:extSrc* oder eine beliebige Anzahl von Skalen *msml:scale* besitzen.

Kanten *msml:edge* als direkte Kinder von *msml:msobject* dienen zur Beschreibung der Verfeinerungsbeziehungen (Kantentyp: *refinement*) zwischen den Elementen der unterschiedlichen Skalen.

- @id (notwendig, xs:ID) Das *id*-Attribut enthält einen im gesamten Dokument eindeutigen Bezeichner, der zur Identifizierung des Elements dient.
- @name (optional, xs:string) Das *name*-Attribut wird zur Vergabe eines Namen für das multiskalierte Objekt benutzt.
- @USE (optional, xs:IDREF) Bei Verwendung eines in einem *msml:library*-Element vordefinierten *msml:msobject* muss im *USE*-Attribut der Name des *DEF*-Attributes des gewünschten Bibliothekselementes angegeben werden.
- @showScale (optional, xs:NCName) Das *showScale*-Attribut gibt den **Namen** (nicht die ID!) der Skale des multiskalierten Objektes an, die vorausgewählt ist.

A.6 msml:scale

Eine Skale enthält einen Graphen aus Knoten *msml:node* und Kanten *msml:edge*, mit dessen Hilfe das Objekt in dieser Skale dargestellt wird. Kanten unterhalb von *msml:scale* dienen zur Beschreibung der Beziehungen zwischen den Knoten innerhalb der Skale. Es können auch multiskalierte Knoten (diese werden durch ein *msml:msobject*-Element dargestellt) verwendet werden. Dann gelten allerdings ein paar Regeln hinsichtlich der Verwendung von Kanten zwischen solchen multiskalierten Knoten und einfachen Knoten. Siehe dazu Abschnitt 2.4.2, Quelltext B.4 und Abb. 6.4. Soll demzufolge eine Kante von einem multiskalierten Knoten (*msml:object* unterhalb von *msml:scale*) fortführen, so muss für jede seiner Skalen eine solche Kante erstellt werden. Dabei entspricht der Ausgangspunkt dieser erstellten Kanten dem Knoten jeder Skale, der die Verbindung zum Nachfolgeknoten herstellen soll.

Aufgrund der in Abschnitt 2.4.2 vorgenommenen Einschränkungen, muss zur Verbindung eines Knotens und eines multiskalierten Knotens als Ziel der Kante nur die ID des *msml:msobject*-Elements angegeben werden. Damit wird automatisch das

Wurzelelement des Graphen der gewählten Skale des multiskalierten Knoten als Ziel der Kante angesehen.

- @id (notwendig, xs:ID) Das *id*-Attribut enthält einen im gesamten Dokument eindeutigen Bezeichner, der zur Identifizierung des Elements dient.
- @name (notwendig, xs:NCName) Das *name*-Attribut wird zur Vergabe eines Skalennamen benutzt. Dieser ***muss*** innerhalb eines multiskalierten Objektes eindeutig sein.

A.7 msml:node

Das *msml:node*-Element dient zur Beschreibung eines einfachen Knotens in einem Graphen.

- @id (notwendig, xs:ID) Das *id*-Attribut enthält einen im gesamten Dokument eindeutigen Bezeichner, der zur Identifizierung des Elements dient.
- @name (optional, xs:string) Das *name*-Attribut wird zur Vergabe eines Knotennamen benutzt.

A.8 msml:edge

Das *msml:edge*-Element dient zum Beschreiben einer gerichteten Kante. Ungerichtete Kanten werden durch zwei gerichtete Kanten entgegengesetzter Richtung beschrieben.

- @source (notwendig, xs:NCName) Das *source*-Attribut enthält die ID des Elements, das der Ausgangspunkt der Kante ist.
- @target (notwendig, xs:NCName) Das *target*-Attribut enthält die ID des Elements, das der Zielpunkt der Kante ist.
- @type (optional, Aufzählung: *branch, successor, refinement, userdefined*)
  - refinement: Dieser Kantentyp wird zur Modellierung von Verfeinerungsrelationen zwischen den Knoten verschiedener Skalen verwendet. Er darf nur bei Kanten eingesetzt werden, die direkte Kindelemente von *msml:msobject* sind.
  - Der Kantentyp *userdefined* wird genutzt, um eigene Kantentypen kenntlich zu machen und gegebenenfalls beschreibende Eigenschaften in einem untergeordneten *msml:data* Element abzulegen.

– branch/successor: Dieser Kantentyp wird zur Modellierung von Verzweigungs- und Nachfolgerrelationen zwischen den Knoten einer Skale verwendet. Er darf nur bei Kanten eingesetzt werden, die direkte Kindelemente von *msml:scale* sind. Diese beiden Kantentypen entstammen eigentlich den Beispielen zur Darstellung pflanzlicher, topologischer Strukturen aus [GC98]. Die Definition dieser Kantentypen als integrierter Bestandteil von MSML wurde vorgenommen, da sie auch zur Beschreibung allgemeiner topologischer Strukturen genutzt werden können. Gegebenenfalls sollten sie in einer späteren MSML-Version entfernt werden und mithilfe des Kantentyps *userdefined* gekennzeichnet werden.

A.9 msml:extSrc

Das *msml:extSrc*-Element dient dazu, ein Element vom Typ des ihm übergeordneten Elements (*msml:msobject* oder *msml:scale*) an dieser Stelle einzufügen.

- @src (notwendig, xs:anyURI) Das *src*-Attribut gibt mittels URI bzw. URI-Referenz das Element an, auf das verwiesen wird. Das Ziel kann sowohl innerhalb des aktuellen Dokuments, als auch in einem externen MSML-Dokument liegen. Ist im *src*-Attribut eine URI angegeben, wird vorausgesetzt, dass im Zieldokument nur ein Element des übergeordneten Elementtyps vorhanden ist. Dieses wird an dieser Stelle eingefügt. Eine URI-Referenz besteht aus einer URI und, abgetrennt durch #, aus einem Fragmentbezeichner. Dabei muss der Fragmentbezeichner die ID eines Elements vom gleichen Typ wie des übergeordneten Elements des *msml:extSrc*-Elementes sein.

A.10 Eigenschaftselemente

Eigenschaftselemente sind beliebige XML-Elemente, die zur Beschreibung ihrer übergeordneten MSML-Elemente dienen. Über die Art und Anzahl der verwendeten Eigenschaftselemente entscheidet der Benutzer. Dabei muss jedes Element, das eine Eigenschaft beschreibt, durch einen Namensraum gekennzeichnet sein. Wird die Datei von einer Anwendung gelesen, werden nur die Eigenschaftselemente verarbeitet werden, die von dieser Anwendung unterstützt werden.

Es ist Aufgabe desjenigen, der ein Eigenschaftselement definiert, dafür zu sorgen, dass diesem Element ein Namensraum zugeordnet wird, wodurch das Element jederzeit identifizierbar wird. Es ist vorteilhaft, den Namensraum als URL in Verbindung mit dem Namen des Eigenschaftselements zu einer jederzeit verfügbaren Adresse zur Aufbewahrung der Dokumentation zu benutzen. Dadurch ist es möglich, bei Erhalt einer MSML-Datei, die unbekannte Eigenschaftselemente enthält, deren Definition

zu erhalten. Dadurch ist der Benutzer in der Lage, selbst zu entscheiden, ob er bestimmte Eigenschaftselemente verwenden und in seiner Anwendung implementieren möchte, oder ob er diese Elemente in eigene Eigenschaftselemente umwandeln möchte. Dadurch können Anwender aus verschiedenen Fachgebieten mit verschiedenen Anforderungen an Datenmaterial mit einem Datenformat arbeiten.

A.10.1 GroIMP-spezifische Eigenschaftselemente

Der XML-Namensraum der GroIMP-spezifischen Eigenschaftselemente (nachfolgend mit dem Präfix **g:** benannt) lautet: `http://grogra.de/msml/datatypes/groimp`.

Der Umfang der Attribute kann beliebig erweitert werden und stellt momentan nur eine minimale Auswahl der in GroIMP verfügbaren Attribute dar. Zur Beschreibung der Eigenschaften für Form, Aussehen und Lage von Objekten wurden Elemente von X3D gewählt, um eine bessere Kompatibilität zu anderen Anwendungen und ein besseres Verständnis der Benutzer zu erreichen. Um eine vollständige Abbildung aller GroIMP-spezifischen Attribute, insbesondere des Shader-Systems, zu erreichen, müssen zusätzliche Eigenschaftselemente definiert bzw. die bereits in GroIMP vorhandene Serialisierungsfunktion genutzt werden. Die Serialisierungsfunktion wandelt alle GroIMP-spezifischen Attribute eines Objektes in einen String um, der dann innerhalb eines eigenen GroIMP-spezifischen Eigenschaftselements ausgegeben werden kann.

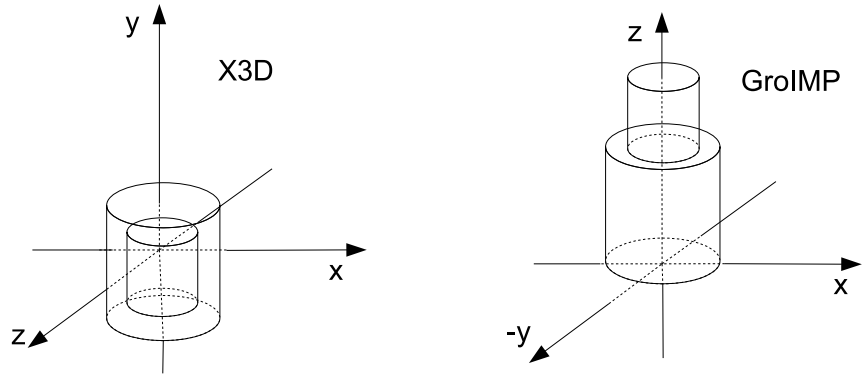
g:Shape

Dieses Element dient zur Beschreibung der geometrischen Form eines Knoten. Knoten ohne *g:shape*-Element sind nicht sichtbar. Das *g:shape*-Element kann alle Elemente vom Typ X3DGeometryNode enthalten. Im Rahmen dieser Arbeit wurden nur die folgenden X3D-Elemente für den Im-/Export mit GroIMP implementiert:

- Box
- Cone
- Cylinder
- Sphere

Die Attribute *startpos* und *endpos* sind notwendig, da es GroIMP im Gegensatz zu X3D ermöglicht, den Koordinatenursprung des lokalen Koordinatensystems eines Objektes entlang seiner senkrechten Achse variabel festzulegen (=*startpos*).

Bei X3D liegt der Koordinatenursprung des lokalen Koordinatensystems bei einem neu erzeugten Primitivobjekt in der Mitte der senkrechten Achse des Objekts. Bei GroIMP gilt das nur für ein Kugel-Objekt. Bei den anderen Primitivobjekten



(a) Standardeinstellung in X3D: startpos=0.5 endpos=0

(b) Standardeinstellung in GroIMP: startpos=0 endpos=1

Abbildung A.1: Beispiel für die unterschiedliche Belegung der startpos- und endpos-Attribute für X3D- und GroIMP-Standardwerte bei einem Zylinder-Objekt. Der kleinere Zylinder ist ein Nachfolger des größeren Zylinders und verdeutlicht die Wirkung des endpos-Attributes.

liegt der Koordinatenursprung des lokalen Koordinatensystems bei einem neu erzeugten Primitivobjekt am Anfang der senkrechten Achse.

Ebenso ermöglicht es GroIMP festzulegen, an welcher Stelle auf der senkrechten Achse eines Objekts der Koordinatenursprung des lokalen Koordinatensystems der Nachfolgerelemente des Objekts festgelegt wird (=*endpos*).

Bei X3D liegt der Koordinatenursprung des lokalen Koordinatensystems eines Nachfolgerelements eines Objektes bei einem neu erzeugten Primitivobjekt im Koordinatenursprung des lokalen Koordinatensystems des Vorgängerelements. Bei GroIMP gilt das nur für ein Kugel-Objekt. Bei den anderen Primitivobjekten liegt der Koordinatenursprung des lokalen Koordinatensystems eines Nachfolgerelements eines Objektes am Ende der senkrechten Achse des Objektes.

Beispiel A.1 *Die Float-Werte der beiden Attribute geben den Wert relativ zu der auf 1 normierten Länge des senkrechten Vektors des Objektes an.*
Float-Wert 0 = Anfang (unteres Ende) des senkrechten Vektors
Float-Wert 0.5 = Mitte des senkrechten Vektors
Float-Wert 1 = Ende (oberes Ende) des senkrechten Vektors

- @startpos (optional, xs:float) Position des Koordinatenursprungs des lokalen Koordinatensystems des Objekts relativ zu seiner senkrechten Achse
- @endpos (optional, xs:float) Position des Koordinatenursprungs des lokalen Koordinatensystems der Nachfolgerelemente des Objekts relativ zu seiner senkrechten Achse

g:Appearance

Dieses Element dient zur Beschreibung des Aussehens eines Knoten. Das *g:appearance*-Element darf genau einen X3DAppearanceNode enthalten. Im Rahmen dieser Arbeit wurden nur die folgenden X3D-Elemente für den Im-/Export mit GroIMP implementiert:

- Appearance
- ImageTexture (nur Import, da beim Import das Bild eingelesen und im Projekt abgelegt wird, ohne die URL zu speichern. Deshalb ist ein Export unter Angabe einer URL nicht möglich. Das Bild als Datei neu zu erstellen ist möglich, aber nicht implementiert.)
- Material
- PixelTexture
- TextureTransform

Dieses Attribut bildet nur einen kleinen Teil der in GroIMP möglichen Aussehensbeschreibungen ab, da der Sprachumfang von X3D zur Beschreibung des Aussehens von Objekten entsprechend begrenzt ist. GroIMP arbeitet bei der Definition des Aussehens eines Objektes auf der Basis von Shadern. Diese können derzeit in X3D nur eingeschränkt beschrieben werden. Es gibt aber Bemühungen [Con04b] X3D-Syntax zur Beschreibung von Shadern zu entwickeln. Diese kann man bei der Definition GroIMP-spezifischer Attribute berücksichtigen.

g:Transform

Dient zur Beschreibung der Lage eines Knotens und seiner Kinder. Im Rahmen dieser Arbeit wurden nur die folgenden X3D-Elemente für den Im-/Export mit GroIMP implementiert.

- Transform

Für eine detaillierte Beschreibung des *x3d:Transform*-Elements verweise ich auf [Con04a][3]. Innerhalb des *g:Transform*-Elements können keine anderen Elemente als *x3d:Transform*-Elemente vorkommen, deshalb kann das *x3d:Transform*-Element beliebig tief verschachtelt werden. Das Produkt dieser Transformationen wird dann auf das Koordinatensystem des aktuellen Knotens angewendet. Alle Kinder des aktuellen Knotens, das heißt über Verzweigungs- oder Nachfolger-Kanten mit ihm verbundene Knoten, definieren ihre Koordinatensysteme relativ zum Koordinatensystem ihres Elternknoten.

[3] `http://www.web3d.org/x3d/specifications/ISO-IEC-19775-X3DAbstractSpecification/Part01/components/group.html#Transform`, letzter Zugriff am 26.10.2006

Anhang B

Quelltexte der MSML-Beispiele

B.1 msml-singleObject.xml

Dies ist der MSML-Quelltext für ein Objekt mit nur einer Skale. Es entspricht dem Objekt in Skale E in Abb. 2.3.

```
<?xml version="1.0" encoding="UTF-8"?>
<msml xmlns="http://grogra.de/msml"
  xmlns:g="http://grogra.de/msml/datatypes/groimp"
  xmlns:x3d="http://www.web3d.org/specifications"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://grogra.de/msml msml-base.xsd" version="1.0">
  <library>
    <data>
      <g:Appearance DEF="wood">
        <x3d:Appearance>
          <x3d:ImageTexture url="WOOD6.jpg" />
        </x3d:Appearance>
      </g:Appearance>
    </data>
  </library>
  <msobject id="mso1" name="single Graph">
    <scale id="mso1-s1" name="singleScale">
      <node id="mso1-s1-n1" name="n1">
        <data>
          <g:Shape>
            <x3d:Cylinder height="2.0" radius="0.6" />
          </g:Shape>
          <g:Appearance USE="wood" />
          <g:Transform>
            <x3d:Transform translation="0 1 0" />
          </g:Transform>
        </data>
      </node>
      <node id="mso1-s1-n2" name="n2">
        <data>
          <g:Shape>
            <x3d:Cylinder height="2.0" radius="0.5" />
          </g:Shape>
          <g:Appearance USE="wood" />
          <g:Transform>
            <x3d:Transform translation="0 2 0" />
          </g:Transform>
        </data>
      </node>
      <node id="mso1-s1-n3" name="n3">
        <data>
          <g:Shape>
            <x3d:Cylinder height="2.0" radius="0.4" />
          </g:Shape>
          <g:Appearance USE="wood" />
          <g:Transform>
            <x3d:Transform translation="0 2 0"
              rotation="0 0 1 1.57" center="0 -1 0"/>
```

```
49          </g:Transform>
          </data>
51      </node>
        <node id="msol-s1-n4" name="n4">
53        <data>
            <g:Shape>
55            <x3d:Cylinder height="2.0" radius="0.5" />
            </g:Shape>
57          <g:Appearance USE="wood" />
            <g:Transform>
59            <x3d:Transform translation="0 2 0"
                rotation="0 0 1 -1.57" center="0 -1 0"/>
61          </g:Transform>
          </data>
63      </node>
        <node id="msol-s1-n5" name="n5">
65        <data>
            <g:Shape>
67            <x3d:Cylinder height="2.0" radius="0.4" />
            </g:Shape>
69          <g:Appearance USE="wood" />
            <g:Transform>
71            <x3d:Transform translation="0 2 0" rotation="0 0 1 1.57" center="0 -1 0"/>
            </g:Transform>
73        </data>
        </node>
75      <node id="msol-s1-n6" name="n6">
          <data>
77          <g:Shape>
              <x3d:Cylinder height="2.0" radius="0.4" />
79          </g:Shape>
            <g:Appearance USE="wood" />
81          <g:Transform>
              <x3d:Transform translation="0 2 0"
83              rotation="0 0 1 0.8" center="0 -1 0"/>
            </g:Transform>
85        </data>
        </node>
87      <node id="msol-s1-n7" name="n7">
          <data>
89          <g:Shape>
              <x3d:Cylinder height="2.0" radius="0.4" />
91          </g:Shape>
            <g:Appearance USE="wood" />
93          <g:Transform>
              <x3d:Transform translation="0 2 0"/>
95          </g:Transform>
          </data>
97      </node>
        <node id="msol-s1-n8" name="n8">
99        <data>
            <g:Shape>
101           <x3d:Cylinder height="2.0" radius="0.3" />
            </g:Shape>
103         <g:Appearance USE="wood" />
            <g:Transform>
105           <x3d:Transform translation="0 2 0"
                rotation="0 0 1 0.8" center="0 -1 0"/>
107         </g:Transform>
          </data>
109     </node>
        <node id="msol-s1-n9" name="n9">
111       <data>
            <g:Shape>
113           <x3d:Cylinder height="2.0" radius="0.4" />
            </g:Shape>
115         <g:Appearance USE="wood" />
            <g:Transform>
117           <x3d:Transform translation="0 2 0"/>
            </g:Transform>
119       </data>
        </node>
121     <node id="msol-s1-n10" name="n10">
          <data>
123         <g:Shape>
              <x3d:Cylinder height="2.0" radius="0.3" />
125         </g:Shape>
            <g:Appearance USE="wood" />
127         <g:Transform>
              <x3d:Transform translation="0 2 0"
129             rotation="0 0 1 -1.57" center="0 -1 0"/>
            </g:Transform>
131       </data>
```

```
      </node>
      <node id="mso1-s1-n11" name="n11">
        <data>
          <g:Shape>
            <x3d:Cylinder height="2.0" radius="0.2" />
          </g:Shape>
          <g:Appearance USE="wood" />
          <g:Transform>
            <x3d:Transform translation="0 2 0"
              rotation="0 0 1 0.8" center="0 -1 0"/>
          </g:Transform>
        </data>
      </node>
      <node id="mso1-s1-n12" name="n12">
        <data>
          <g:Shape>
            <x3d:Cylinder height="2.0" radius="0.3" />
          </g:Shape>
          <g:Appearance USE="wood" />
          <g:Transform>
            <x3d:Transform translation="0 2 0" />
          </g:Transform>
        </data>
      </node>
      <node id="mso1-s1-n13" name="n13">
        <data>
          <g:Shape>
            <x3d:Cylinder height="2.0" radius="0.2" />
          </g:Shape>
          <g:Appearance USE="wood" />
          <g:Transform>
            <x3d:Transform translation="0 2 0"
              rotation="0 0 1 -1.57" center="0 -1 0"/>
          </g:Transform>
        </data>
      </node>
      <node id="mso1-s1-n14" name="n14">
        <data>
          <g:Shape>
            <x3d:Cylinder height="2.0" radius="0.1" />
          </g:Shape>
          <g:Appearance USE="wood" />
          <g:Transform>
            <x3d:Transform translation="0 2 0"
              rotation="0 0 1 0.8" center="0 -1 0"/>
          </g:Transform>
        </data>
      </node>
      <node id="mso1-s1-n15" name="n15">
        <data>
          <g:Shape>
            <x3d:Cylinder height="2.0" radius="0.2" />
          </g:Shape>
          <g:Appearance USE="wood" />
          <g:Transform>
            <x3d:Transform translation="0 2 0"
              rotation="0 0 1 1.57" center="0 -1 0"/>
          </g:Transform>
        </data>
      </node>
      <node id="mso1-s1-n16" name="n16">
        <data>
          <g:Shape>
            <x3d:Cylinder height="2.0" radius="0.1" />
          </g:Shape>
          <g:Appearance USE="wood" />
          <g:Transform>
            <x3d:Transform translation="0 2 0"
              rotation="0 0 1 -0.8" center="0 -1 0"/>
          </g:Transform>
        </data>
      </node>
      <node id="mso1-s1-n17" name="n17">
        <data>
          <g:Shape>
            <x3d:Cylinder height="2.0" radius="0.2" />
          </g:Shape>
          <g:Appearance USE="wood" />
          <g:Transform>
            <x3d:Transform translation="0 2 0" />
          </g:Transform>
        </data>
      </node>
```

```
215       <edge source="mso1-s1-n1" target="mso1-s1-n2" type="successor" />
          <edge source="mso1-s1-n1" target="mso1-s1-n4" type="branch" />
217       <edge source="mso1-s1-n2" target="mso1-s1-n3" type="branch" />
          <edge source="mso1-s1-n2" target="mso1-s1-n9" type="successor" />
219       <edge source="mso1-s1-n4" target="mso1-s1-n5" type="branch" />
          <edge source="mso1-s1-n4" target="mso1-s1-n6" type="branch" />
221       <edge source="mso1-s1-n4" target="mso1-s1-n7" type="successor" />
          <edge source="mso1-s1-n7" target="mso1-s1-n8" type="successor" />
223       <edge source="mso1-s1-n9" target="mso1-s1-n10" type="branch" />
          <edge source="mso1-s1-n9" target="mso1-s1-n12" type="successor" />
225       <edge source="mso1-s1-n10" target="mso1-s1-n11" type="successor" />
          <edge source="mso1-s1-n12" target="mso1-s1-n13" type="branch" />
227       <edge source="mso1-s1-n12" target="mso1-s1-n15" type="branch" />
          <edge source="mso1-s1-n12" target="mso1-s1-n17" type="successor" />
229       <edge source="mso1-s1-n13" target="mso1-s1-n14" type="successor" />
          <edge source="mso1-s1-n15" target="mso1-s1-n16" type="successor" />
231     </scale>
      </msobject>
233 </msml>
```

B.2 msml-multiscaleObject.xml

Dies ist der MSML-Quelltext für ein multiskaliertes Objekt mit fünf Skalen. Es entspricht dem multiskalierten Objekt in Abb. 2.3. Dabei wird die Skale E durch Einbinden der Skale des Objektes aus der Datei *msml-singleObject.xml* erreicht. Die grafische Darstellung der verschiedenen Skalen dieser Datei in GroIMP kann in Abb. 6.2 betrachtet werden.

```
1 <?xml version="1.0" encoding="UTF-8"?>
  <msml xmlns="http://grogra.de/msml"
3   xmlns:g="http://grogra.de/msml/datatypes/groimp"
    xmlns:x3d="http://www.web3d.org/specifications"
5   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://grogra.de/msml msml-base.xsd" version="1.0">
7   <library>
      <data>
9       <g:Appearance DEF="lightgray">
          <x3d:Appearance>
11          <x3d:Material diffuseColor="0.7 0.7 0.7"/>
          </x3d:Appearance>
13      </g:Appearance>
        <g:Appearance DEF="yellow">
15        <x3d:Appearance>
            <x3d:Material diffuseColor="1 1 0"/>
17        </x3d:Appearance>
        </g:Appearance>
19      <g:Appearance DEF="blue">
          <x3d:Appearance>
21          <x3d:Material diffuseColor="0 0 1"/>
          </x3d:Appearance>
23      </g:Appearance>
        <g:Appearance DEF="green">
25        <x3d:Appearance>
            <x3d:Material diffuseColor="0 1 0"/>
27        </x3d:Appearance>
        </g:Appearance>
29    </data>
    </library>
31  <msobject id="mso1" name="MSGexampleTree" showScale="E">
      <!-- Scale A ------------------------------------------------------------------ -->
33    <scale id="mso1-s1" name="A">
        <node id="mso1-s1-n1" name="A-n1">
35        <data>
            <g:Shape>
37            <x3d:Sphere radius="5"/>
            </g:Shape>
39          <g:Appearance USE="yellow"/>
            <g:Transform>
41            <x3d:Transform translation="0 5 0"/>
            </g:Transform>
43        </data>
        </node>
45    </scale>
```

```
    <edge source="mso1-s1-n1" target="mso1-s2-n1" type="refinement"/>
47  <edge source="mso1-s1-n1" target="mso1-s2-n2" type="refinement"/>
    <edge source="mso1-s1-n1" target="mso1-s2-n3" type="refinement"/>
49  <edge source="mso1-s1-n1" target="mso1-s2-n4" type="refinement"/>
    <edge source="mso1-s1-n1" target="mso1-s2-n5" type="refinement"/>
51  <edge source="mso1-s1-n1" target="mso1-s4-n1" type="refinement"/>
    <edge source="mso1-s1-n1" target="mso1-s4-n2" type="refinement"/>
53  <edge source="mso1-s1-n1" target="mso1-s4-n3" type="refinement"/>
    <edge source="mso1-s1-n1" target="mso1-s4-n4" type="refinement"/>
55  <edge source="mso1-s1-n1" target="mso1-s4-n5" type="refinement"/>
    <!-- Scale B ------------------------------------------------------------------- -->
57  <scale id="mso1-s2" name="B">
      <node id="mso1-s2-n1" name="B-n1">
59      <data>
          <g:Shape>
61          <x3d:Cylinder height="4.0" radius="2.5"/>
          </g:Shape>
63        <g:Appearance USE="blue"/>
          <g:Transform>
65          <x3d:Transform translation="0 2 0"/>
          </g:Transform>
67      </data>
      </node>
69    <node id="mso1-s2-n2" name="B-n2">
        <data>
71        <g:Shape>
            <x3d:Cylinder height="2.0" radius="1"/>
73        </g:Shape>
          <g:Appearance USE="blue"/>
75        <g:Transform>
            <x3d:Transform translation="3.5 0.5 0" rotation="0 0 1 -1.57"/>
77        </g:Transform>
        </data>
79    </node>
      <node id="mso1-s2-n3" name="B-n3">
81      <data>
          <g:Shape>
83          <x3d:Cylinder height="2.0" radius="0.5"/>
          </g:Shape>
85        <g:Appearance USE="blue"/>
          <g:Transform>
87          <x3d:Transform translation="-3.5 0.5 0" rotation="0 0 1 1.57"/>
          </g:Transform>
89      </data>
      </node>
91    <node id="mso1-s2-n4" name="B-n4">
        <data>
93        <g:Shape>
            <x3d:Cylinder height="4.0" radius="3.5"/>
95        </g:Shape>
          <g:Appearance USE="blue"/>
97        <g:Transform>
            <x3d:Transform translation="0 4 0"/>
99        </g:Transform>
        </data>
101   </node>
      <node id="mso1-s2-n5" name="B-n5">
103     <data>
          <g:Shape>
105         <x3d:Cylinder height="2.0" radius="1"/>
          </g:Shape>
107       <g:Appearance USE="blue"/>
          <g:Transform>
109         <x3d:Transform translation="0 3 0"/>
          </g:Transform>
111     </data>
      </node>
113   <edge source="mso1-s2-n1" target="mso1-s2-n2" type="successor"/>
      <edge source="mso1-s2-n1" target="mso1-s2-n3" type="branch"/>
115   <edge source="mso1-s2-n1" target="mso1-s2-n4" type="successor"/>
      <edge source="mso1-s2-n4" target="mso1-s2-n5" type="branch"/>
117 </scale>
    <edge source="mso1-s2-n1" target="mso1-s5mso1-s1-n1" type="refinement"/>
119 <edge source="mso1-s2-n1" target="mso1-s5mso1-s1-n2" type="refinement"/>
    <edge source="mso1-s2-n1" target="mso1-s5mso1-s1-n4" type="refinement"/>
121 <edge source="mso1-s2-n1" target="mso1-s5mso1-s1-n5" type="refinement"/>
    <edge source="mso1-s2-n1" target="mso1-s5mso1-s1-n6" type="refinement"/>
123 <edge source="mso1-s2-n2" target="mso1-s5mso1-s1-n7" type="refinement"/>
    <edge source="mso1-s2-n2" target="mso1-s5mso1-s1-n8" type="refinement"/>
125 <edge source="mso1-s2-n3" target="mso1-s5mso1-s1-n3" type="refinement"/>
    <edge source="mso1-s2-n4" target="mso1-s5mso1-s1-n9" type="refinement"/>
127 <edge source="mso1-s2-n4" target="mso1-s5mso1-s1-n10" type="refinement"/>
    <edge source="mso1-s2-n4" target="mso1-s5mso1-s1-n11" type="refinement"/>
```

```
129     <edge source="mso1-s2-n4" target="mso1-s5mso1-s1-n12" type="refinement"/>
        <edge source="mso1-s2-n4" target="mso1-s5mso1-s1-n15" type="refinement"/>
131     <edge source="mso1-s2-n4" target="mso1-s5mso1-s1-n16" type="refinement"/>
        <edge source="mso1-s2-n4" target="mso1-s5mso1-s1-n17" type="refinement"/>
133     <edge source="mso1-s2-n5" target="mso1-s5mso1-s1-n13" type="refinement"/>
        <edge source="mso1-s2-n5" target="mso1-s5mso1-s1-n14" type="refinement"/>
135     <!-- Scale C ------------------------------------------------------------------- -->
        <scale id="mso1-s3" name="C">
137       <node id="mso1-s3-n1" name="C-n1"/>
          <node id="mso1-s3-n2" name="C-n2"/>
139       <node id="mso1-s3-n3" name="C-n3"/>
          <node id="mso1-s3-n4" name="C-n4"/>
141       <node id="mso1-s3-n5" name="C-n5"/>
          <node id="mso1-s3-n6" name="C-n6"/>
143       <node id="mso1-s3-n7" name="C-n7"/>
          <node id="mso1-s3-n8" name="C-n8"/>
145       <node id="mso1-s3-n9" name="C-n9"/>
          <edge source="mso1-s3-n1" target="mso1-s3-n2" type="branch"/>
147       <edge source="mso1-s3-n1" target="mso1-s3-n3" type="branch"/>
          <edge source="mso1-s3-n1" target="mso1-s3-n5" type="successor"/>
149       <edge source="mso1-s3-n3" target="mso1-s3-n4" type="branch"/>
          <edge source="mso1-s3-n5" target="mso1-s3-n6" type="branch"/>
151       <edge source="mso1-s3-n5" target="mso1-s3-n8" type="successor"/>
          <edge source="mso1-s3-n8" target="mso1-s3-n7" type="branch"/>
153       <edge source="mso1-s3-n8" target="mso1-s3-n9" type="successor"/>
        </scale>
155     <edge source="mso1-s3-n1" target="mso1-s5mso1-s1-n1" type="refinement"/>
        <edge source="mso1-s3-n1" target="mso1-s5mso1-s1-n2" type="refinement"/>
157     <edge source="mso1-s3-n2" target="mso1-s5mso1-s1-n3" type="refinement"/>
        <edge source="mso1-s3-n3" target="mso1-s5mso1-s1-n4" type="refinement"/>
159     <edge source="mso1-s3-n3" target="mso1-s5mso1-s1-n5" type="refinement"/>
        <edge source="mso1-s3-n3" target="mso1-s5mso1-s1-n7" type="refinement"/>
161     <edge source="mso1-s3-n3" target="mso1-s5mso1-s1-n8" type="refinement"/>
        <edge source="mso1-s3-n4" target="mso1-s5mso1-s1-n6" type="refinement"/>
163     <edge source="mso1-s3-n5" target="mso1-s5mso1-s1-n9" type="refinement"/>
        <edge source="mso1-s3-n6" target="mso1-s5mso1-s1-n10" type="refinement"/>
165     <edge source="mso1-s3-n6" target="mso1-s5mso1-s1-n11" type="refinement"/>
        <edge source="mso1-s3-n7" target="mso1-s5mso1-s1-n13" type="refinement"/>
167     <edge source="mso1-s3-n7" target="mso1-s5mso1-s1-n14" type="refinement"/>
        <edge source="mso1-s3-n8" target="mso1-s5mso1-s1-n12" type="refinement"/>
169     <edge source="mso1-s3-n8" target="mso1-s5mso1-s1-n15" type="refinement"/>
        <edge source="mso1-s3-n8" target="mso1-s5mso1-s1-n16" type="refinement"/>
171     <edge source="mso1-s3-n9" target="mso1-s5mso1-s1-n17" type="refinement"/>
        <!-- Scale D ------------------------------------------------------------------- -->
173     <scale id="mso1-s4" name="D">
          <node id="mso1-s4-n1" name="D-n1">
175         <data>
              <g:Shape>
177             <x3d:Sphere radius="1.5"/>
              </g:Shape>
179           <g:Appearance USE="green"/>
              <g:Transform>
181             <x3d:Transform translation="0.0 1.5 0"/>
              </g:Transform>
183         </data>
          </node>
185       <node id="mso1-s4-n2" name="D-n2">
            <data>
187           <g:Shape>
                <x3d:Sphere radius="2.5"/>
189           </g:Shape>
              <g:Appearance USE="green"/>
191           <g:Transform>
                <x3d:Transform translation="3.0 2.0 0"/>
193           </g:Transform>
            </data>
195       </node>
          <node id="mso1-s4-n3" name="D-n3">
197         <data>
              <g:Shape>
199             <x3d:Sphere radius="1.5"/>
              </g:Shape>
201           <g:Appearance USE="green"/>
              <g:Transform>
203             <x3d:Transform translation="-0.5 2.5 0"/>
              </g:Transform>
205         </data>
          </node>
207       <node id="mso1-s4-n4" name="D-n4">
            <data>
209           <g:Shape>
                <x3d:Sphere radius="2.5"/>
211           </g:Shape>
```

```
            <g:Appearance USE="green"/>
213         <g:Transform>
              <x3d:Transform translation="1 3 0"/>
215         </g:Transform>
          </data>
217     </node>
        <node id="mso1-s4-n5" name="D-n5">
219       <data>
            <g:Shape>
221           <x3d:Sphere radius="0.5"/>
            </g:Shape>
223         <g:Appearance USE="green"/>
            <g:Transform>
225           <x3d:Transform translation="-0.5 2.75 0"/>
            </g:Transform>
227       </data>
        </node>
229     <edge source="mso1-s4-n1" target="mso1-s4-n2" type="branch"/>
        <edge source="mso1-s4-n1" target="mso1-s4-n3" type="successor"/>
231     <edge source="mso1-s4-n3" target="mso1-s4-n4" type="successor"/>
        <edge source="mso1-s4-n4" target="mso1-s4-n5" type="successor"/>
233   </scale>
      <edge source="mso1-s4-n1" target="mso1-s3-n1" type="refinement"/>
235   <edge source="mso1-s4-n1" target="mso1-s3-n2" type="refinement"/>
      <edge source="mso1-s4-n2" target="mso1-s3-n3" type="refinement"/>
237   <edge source="mso1-s4-n2" target="mso1-s3-n4" type="refinement"/>
      <edge source="mso1-s4-n3" target="mso1-s3-n5" type="refinement"/>
239   <edge source="mso1-s4-n3" target="mso1-s3-n6" type="refinement"/>
      <edge source="mso1-s4-n4" target="mso1-s3-n7" type="refinement"/>
241   <edge source="mso1-s4-n4" target="mso1-s3-n8" type="refinement"/>
      <edge source="mso1-s4-n5" target="mso1-s3-n9" type="refinement"/>
243   <!-- Scale E ------------------------------------------------------------------------ -->
      <scale id="mso1-s5" name="E">
245     <extSrc src="msml-singleObject.xml#mso1-s1"/>
      </scale>
247 </msobject>
  </msml>
```

B.3 msml-scene.xml

Dies ist der MSML-Quelltext für eine Gruppe von Objekten. Diese besteht aus einem Bodenelement mit Grastextur, zwei multiskalierten Objekten, wobei zweimal die Datei *msml-multiscaleObject.xml* eingebunden wird und einem einfachen Objekt, das durch Einbinden der *msml-singleObject.xml* erzeugt wird. Die entsprechende Darstellung dieser Szene ist in Abb. 6.1 ersichtlich.

```
<?xml version="1.0" encoding="UTF-8"?>
<msml xmlns="http://grogra.de/msml"
  xmlns:g="http://grogra.de/msml/datatypes/groimp"
  xmlns:x3d="http://www.web3d.org/specifications"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://grogra.de/msml msml-base.xsd" version="1.0">
  <library>
    <data>
      <g:Appearance DEF="meadow">
        <x3d:Appearance>
          <x3d:ImageTexture url="GRASS2.jpg"/>
        </x3d:Appearance>
      </g:Appearance>
    </data>
  </library>
  <group id="g1" name="scene1">
    <msobject id="g1-mso2" name="Surface">
      <scale id="g1-mso2-s1" name="Ground">
        <node id="g1-mso2-s1-n1">
          <data>
            <g:Shape>
              <x3d:Box size="20 1 20"/>
            </g:Shape>
            <g:Appearance USE="meadow"/>
          </data>
        </node>
      </scale>
    </msobject>
    <msobject id="g1-mso3" name="Tree1" showScale="A">
      <data>
        <g:Transform>
          <x3d:Transform translation="9 1 -5"/>
        </g:Transform>
      </data>
      <extSrc src="msml-multiscaleObject.xml"/>
    </msobject>
    <group id="g1-g2">
      <data>
        <g:Transform>
          <x3d:Transform translation="-9 1 3" rotation="0 1 0 0.78"/>
        </g:Transform>
      </data>
      <msobject id="g1-g2-mso4" name="Tree2" showScale="D">
        <extSrc src="msml-multiscaleObject.xml"/>
      </msobject>
    </group>
    <msobject id="g1-mso5" name="Tree3">
      <data>
        <g:Transform>
          <x3d:Transform translation="0 1 0" rotation="0 1 0 1.6"/>
        </g:Transform>
      </data>
      <extSrc src="msml-singleObject.xml"/>
    </msobject>
  </group>
</msml>
```

B.4 msml-ex-msnode.xml

Das ist der MSML-Quelltext für das Beispiel eines multiskalierten Knoten aus Abb. 6.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<msml xmlns="http://grogra.de/msml"
  xmlns:g="http://grogra.de/msml/datatypes/groimp"
  xmlns:x3d="http://www.web3d.org/specifications"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://grogra.de/msml msml-base.xsd" version="1.0">
  <library>
    <data>
      <g:Appearance DEF="blue">
        <x3d:Appearance>
          <x3d:Material diffuseColor="0 0 1"/>
        </x3d:Appearance>
      </g:Appearance>
      <g:Appearance DEF="red">
        <x3d:Appearance>
          <x3d:Material diffuseColor="1 0 0"/>
        </x3d:Appearance>
      </g:Appearance>
      <g:Appearance DEF="green">
        <x3d:Appearance>
          <x3d:Material diffuseColor="0 1 0"/>
        </x3d:Appearance>
      </g:Appearance>
    </data>
  </library>
  <msobject id="mso1" name="multiskal.Objekt" showScale="einfach">
    <scale id="mso1-s1" name="einfach">
      <node id="mso1-s1-n1" name="einf.Knoten1">
        <data>
          <g:Shape startpos="0" endpos="1">
            <x3d:Cylinder height="2.0" radius="1"/>
          </g:Shape>
          <g:Appearance USE="blue"/>
        </data>
      </node>
      <msobject id="mso1-s1-mso1" name="multiskal.Knoten" showScale="einfach">
        <scale id="mso1-s1-mso1-s1" name="einfach">
          <node id="mso1-s1-mso1-s1-n1" name="Knoten1">
            <data>
              <g:Shape startpos="0" endpos="1">
                <x3d:Cylinder height="2.0" radius="1.0"/>
              </g:Shape>
              <g:Appearance USE="green"/>
            </data>
          </node>
        </scale>
        <scale id="mso1-s1-mso1-s2" name="komplex">
          <node id="mso1-s1-mso1-s2-n1" name="Knoten1">
            <data>
              <g:Shape startpos="0" endpos="1">
                <x3d:Cylinder height="1.0" radius="0.5"/>
              </g:Shape>
              <g:Appearance USE="red"/>
            </data>
          </node>
          <node id="mso1-s1-mso1-s2-n2" name="Knoten2">
            <data>
              <g:Shape startpos="0" endpos="1">
                <x3d:Cylinder height="1.0" radius="0.4"/>
              </g:Shape>
              <g:Appearance USE="red"/>
            </data>
          </node>
          <edge source="mso1-s1-mso1-s2-n1" target="mso1-s1-mso1-s2-n2" type="successor"/>
        </scale>
      </msobject>
      <node id="mso1-s1-n2" name="einf.Knoten2">
        <data>
          <g:Shape startpos="0" endpos="1">
            <x3d:Cylinder height="2.0" radius="1"/>
          </g:Shape>
          <g:Appearance USE="blue"/>
        </data>
      </node>
      <edge source="mso1-s1-n1" target="mso1-s1-mso1" type="successor"/>
      <edge source="mso1-s1-mso1-s1-n1" target="mso1-s1-n2" type="successor"/>
      <edge source="mso1-s1-mso1-s2-n2" target="mso1-s1-n2" type="successor"/>
    </scale>
  </msobject>
</msml>
```

Literaturverzeichnis

[ABD+01] M. Altheim, F. Boumphrey, S. Dooley, S. McCarron, S. Schnitzenbaumer, and T. Wugofski. Modularization of XHTML, 10 April 2001. [http://www.w3.org/TR/2001/REC-xhtml-modularization-20010410, last accessed 24. October 2006].

[BCF+06] S. Boag, D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language, 8 June 2006. [http://www.w3.org/TR/xquery, last accessed 24. October 2006].

[BEH+02] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall. GraphML Progress Report: Structural Layer Proposal. In *Proc. 9th Intl. Symp. Graph Drawing (GD '01), LNCS 2265*, pages 501–512. Springer-Verlag, 2002. [http://www.inf.uni-konstanz.de/algo/publications/behhm-gprsl-01.ps.gz, last accessed 24. October 2006].

[BEL04] U. Brandes, M. Eiglsperger, and J. Lerner. GraphML Primer. Technical report, Uni Konstanz, 1 June 2004. [http://graphml.graphdrawing.org/primer/graphml-primer.html, last accessed 24. October 2006].

[BFM05] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 3986: Uniform Resource Identifier (URI): Generic Syntax, January 2005. [http://www.ietf.org/rfc/rfc3986.txt, last accessed 24. October 2006].

[BHLT06] T. Bray, D. Hollander, A. Layman, and R. Tobin. Namespaces in XML 1.0 (Second Edition), 16 August 2006. [http://www.w3.org/TR/xml-names, last accessed 24. October 2006].

[BKK01] T. Bayer, T. Kieninger, and Ö. Kipik. Entwurfsmuster für XML und XSLT. *Java Magazin*, pages 1–3, December 2001. [http://javamagazin.de/itr/online_artikel/psecom,id,152,nodeid,11.html, last accessed 24. October 2006].

[BLP05] U. Brandes, J. Lerner, and C. Pich. GXL to GraphML and Vice Versa with XSLT. In *Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004)*, volume 127 of *ENTCS*, pages 113–125. AKA Akademische Verlagsgesellschaft Berlin, March 2005. [`http://tfs.cs.tu-berlin.de/grabats/Final04/brandes.pdf`, last accessed 24. October 2006].

[BM04] P.V. Biron and A. Malhotra. XML Schema Part 2: Datatypes Second Edition, 28 October 2004. [`http://www.w3.org/TR/xmlschema-2`, last accessed 24. October 2006].

[Bon04] F. Bongers. *XSLT 2.0 - Das umfassende Handbuch zu XSLT 2.0, XPath 2.0 und Saxon 7.* Galileo Press, Bonn, Germany, 1. edition, 2004.

[BPSM+06] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0, 16 August 2006. [`http://www.w3.org/TR/xml`; last accessed 24. October 2006].

[Bru06] D. Brutzman. X3D-Edit for Extensible 3D (X3D) Graphics, 8 July 2006. [`http://www.web3d.org/x3d/content/README.X3D-Edit.html`, last accessed 24. October 2006].

[Bus00] F. Buschmann. *Pattern-orientierte Softwarearchitektur – Ein Pattern System.* Addison-Wesley, München, Germany, 1. edition, 2000.

[CD99] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0, 16 November 1999. [`http://www.w3.org/TR/xpath`, last accessed 24. October 2006].

[CIM+03] D. Carlisle, P. Ion, R. Miner, N. Poppelier, and et. al. Mathematical Markup Language (MathML) Version 2.0 (Second Edition), 21 October 2003. [`http://www.w3.org/TR/MathML`, last accessed 24. October 2006].

[Cla99] J. Clark. XSL Transformations (XSLT) Version 1.0, 16 November 1999. [`http://www.w3.org/TR/xslt`, last accessed 24. October 2006].

[Con04a] Web3D Consortium. ISO/IEC 19775:2004, Extensible 3D (X3D), 2004. [`http://www.web3d.org/x3d/specifications/ISO-IEC-19775-X3DAbstractSpecification`, last accessed 24. October 2006].

[Con04b] Web3D Consortium. Programmable X3D Shader, 2004. [`http://www.web3d.org/x3d/workgroups/x3d-shaders`, last accessed 24. October 2006].

[Con06] Web3D Consortium. X3D International Specification Standards, 2006. [http://www.web3d.org/x3d/specifications/x3d, last accessed 24. October 2006].

[DMO01] S. DeRose, E. Maler, and D. Orchard. XML Linking Language (XLink) Version 1.0, 27 June 2001. [http://www.w3.org/TR/xlink, last accessed 24. October 2006].

[DRE02] S. DeRose, R.D.jr., and E.Maler. XPointer xpointer() Scheme, 19 December 2002. [http://www.w3.org/TR/xptr-xpointer, last accessed 24. October 2006].

[DREJ03] S. DeRose, R.D.jr., E.Maler, and J.Marsh. XPointer xmlns() Scheme, 25 March 2003. [http://www.w3.org/TR/xptr-xmlns, last accessed 24. October 2006].

[DuC03] B. DuCharme. Writing Your Own Functions in XSLT 2.0, 3 September 2003. [http://www.xml.com/pub/a/2003/09/03/trxml.html, last accessed 24. October 2006].

[FJ03] J. Ferraiolo and D. Jackson. Scalable Vector Graphics (SVG) 1.1 Specification, 14 January 2003. [http://www.w3.org/TR/SVG, last accessed 24. October 2006].

[GC98] C. Godin and Y. Caraglio. Multiscale Model of Plant Topological Structures. *Journal of Theoretical Biology*, 191(1):1–46, 7 March 1998.

[GCS99] C. Godin, E. Costes, and H. Sinoquet. A method for describing plant architecture which integrates topology and geometry. *Annals of Botany*, 84:343–357, 1999.

[GEJN03a] P. Grosso, E.Maler, J.Marsh, and N.Walsh. XPointer element() Scheme, 25 March 2003. [http://www.w3.org/TR/xptr-element, last accessed 24. October 2006].

[GEJN03b] P. Grosso, E.Maler, J.Marsh, and N.Walsh. XPointer Framework, 25 March 2003. [http://www.w3.org/TR/xptr-framework, last accessed 24. October 2006].

[GG97] C. Godin and Y. Guédon. AMAPmod v1.8. Introduction and reference manual. Technical report, CIRAD, Montpellier, France, 1997. [http://amap.cirad.fr/amapmod/refermanual18/partI.html, last accessed 24. October 2006].

[GHJ95] E. Gamma, R. Helm, and R. E. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, München, Germany, 1995.

[God00] C. Godin. Representing and encoding plant architecture. A review. *Annals of Forest Science*, 57:413–438, 2000.

[Gra04] The GraphML File Format, 2004. [`http://graphml.graphdrawing.org`, last accessed 24. October 2006].

[Him97] M. Himsolt. GML-Homepage, 20 July 1997. [`http://infosun.fmi.uni-passau.de/Graphlet/GML`, last accessed 24. October 2006].

[HSB06] I. Hermann, R. Swick, and D. Brickley. RDF-Homepage, 19 July 2006. [`http://www.w3.org/RDF`, last accessed 24. October 2006].

[HSSW02] R. Holt, A. Schürr, S.E. Sim, and A. Winter. GXL-Homepage, 17 July 2002. [`http://www.gupro.de/GXL`, last accessed 24. October 2006].

[HSSW05] R.C. Holt, A. Schürr, S.E. Sim, and A. Winter. GXL: A graph-based standard exchange format for reengineering. *Science of Computer Programming*, pages 1–22, 2005.

[IH00] M.S. Marshall I. Herman. GraphXML - An XML Based Graph Interchange Format. Technical Report INS-R0009, Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands, 30 April 2000. [`http://db.cwi.nl/rapporten/abstract.php?abstractnr=613`, last accessed 24. October 2006].

[Kay] M. Kay. Saxonica.com - XSLT and XQUERY Processing. [`http://saxonica.com`, last accessed 24. October 2006].

[Kay04] M. Kay. *XSLT 2.0 Programmer's Reference (Programmer to Programmer).* Wiley Publishing, Inc, Indianapolis, IN, USA, 3. edition, 2004.

[Kay06] M. Kay. XSL Transformations (XSLT) Version 2.0, 8 June 2006. [`http://www.w3.org/TR/xslt20`, last accessed 24. October 2006].

[Kni04] O. Kniemeyer. Rule-based modelling with the XL/GroIMP software. In Harald Schaub, Frank Detje, and Ulrike Brüggemann, editors, *The Logic of Artificial Life. Proceedings of 6th GWAL*, pages 56–65, Bamberg, Germany, 14-16 April 2004. AKA Akademische Verlagsgesellschaft Berlin. [`http://grogra.de/publications/gwal6.pdf`, last accessed 24. October 2006].

[Kni06] O. Kniemeyer. GroIMP, 2006. [`http://www.grogra.de/software/groimp`, last accessed 24. October 2006].

[Mar] S. Marshall. GVF - The Graph Visualization Framework. [`http://gvf.sourceforge.net`, last accessed 24. October 2006].

[MO04] J. March and D. Orchard. XML Inclusions (XInclude) Version 1.0, 20 December 2004. [`http://www.w3.org/TR/xinclude`, last accessed 24. October 2006].

[Ne] P. Nielsen and et. al. CellML. [`http://www.cellml.org`, last accessed 24. October 2006].

[Pun01] J. Punin. XGMML (eXtensible Graph Markup and Modeling Language), 24 August 2001. [`http://www.cs.rpi.edu/~puninj/XGMML`, last accessed 24. October 2006].

[TBMM04] H.S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures Second Edition, 28 October 2004. [`http://www.w3.org/TR/xmlschema-1`, last accessed 24. October 2006].

[TT04] H.S. Thompson and R. Tobin. Validator for XML Schema, 22 April 2004. [`http://www.w3.org/2001/03/webdata/xsv`, last accessed 24. October 2006].

[URI01] URI Planning Interest Group, W3C/IETF. URIs, URLs, and URNs: Clarifications and Recommendations 1.0, 21 September 2001. [`http://www.w3.org/TR/uri-clarification`, last accessed 24. October 2006].

[W3C06] W3C. HyperText Markup Language (HTML) Home Page, 7 September 2006. [`http://www.w3.org/MarkUp`, last accessed 24. October 2006].

[Wat01] A. Watt. *3D-Computergrafik*. Pearson Studium, München, Germany, 2001.

[WFMe06] N. Walsh, M. Fernandez, A. Malhotra, and et. al. XQuery 1.0 and XPath 2.0 Data Model (XDM), 11 July 2006. [`http://www.w3.org/TR/xpath-datamodel`, last accessed 24. October 2006].

[yEd06] yEd-Homepage, 2006. [`http://www.yworks.com/en/products_yed_about.htm`, last accessed 24. October 2006].

[YL01] Yao-ming Yeh and Kuan-Sheng Lee. A XML-Based Plant Modeling Language. In *Proceeding of 2001 14th IPPR Conference on Computer Vision , Graphics and Image Processing*, August 2001.