5 CONTIGRA: Das deklarative Dokumentenmodell

Im Kapitel 4 wurden Anforderungen an eine 3D-Komponentenarchitektur aufgestellt, existierende und verwandte Arbeiten analysiert und ein durchgängig deklaratives Komponentenmodell für 3D-Grafik skizziert. Nachdem die CONTIGRA-Architektur als dokumentzentrierter Lösungsansatz im Überblick vorgestellt wurde, ist dieses Kapitel nun der detaillierteren Beschreibung des zugrundeliegenden Dokumentenmodells gewidmet. Aus den Ebenen bei der Entwicklung komponentenorientierter 3D-Grafik und ihren assoziierten Aufgaben ist deutlich geworden, welche Beschreibungssprachen für den deklarativen Ansatz notwendig sind. Als Lösungsbasis für die Auszeichnungssprachen wurde XML-Schema gewählt, dessen Verwendung zu Beginn dieses Kapitels motiviert wird. Es folgt in drei Unterkapiteln die Beschreibung der entwickelten XML-Grammatiken *CoApplication, CoComponent* und *CoComponentImplementation*. Auf der obersten Ebene ist *CoApplication* als Beschreibungssprache für interaktive 3D-Applikationen angesiedelt. Sie dient der Integration einer komplexen Komponente in einen Anwendungsrahmen und erlaubt die Festlegung typischer Szenenparameter, wie Beleuchtung, Kamera und vordefinierter Blickpunkte.

Jede CONTIGRA-Komponente besteht aus einem Schnittstellendokument als Instanz der Grammatik *CoComponent* und einem Implementierungsdokument als Instanz von *CoComponentImplementation*. Zunächst wird der Aufbau von Schnittstellendokumenten erläutert, wobei vielfältige Metainformationen und weitere Bestandteile für die Unterstützung des Autorenprozesses der Verwendung einer Komponente in mehreren Lebensphasen und durch verschiedene Autoren gerecht werden. Es schließt sich die Erläuterung des Parameterkonzepts an. Komponenten offerieren ihre Funktionalität vollständig über Parameter, die sich zur Anpassung einer Komponente konfigurieren lassen. Schließlich wird dargestellt, wie Implementierungsdokumente einer Komponente aus den wesentlichen Bestandteilen Komponenten- und Szenengraph aufgebaut sind. Dabei erfolgt eine Aufteilung des Szenengraphs in die Teile *Audio, Behavior* und *Geometry*. Während für den Geometrieteil X3D zum Einsatz kommt, wurden für die ersten beiden die eigenen Auszeichnungssprachen *Audio3D* und *Behavior3D* entwickelt, die im Überblick dargestellt werden. Abgerundet wird dieses Kapitel durch die Vorstellung des Verknüpfungskonzeptes für die einzelnen Implementierungsbestandteile und ihre Zuordnung zu den Parametern der Komponentenschnittstelle.

5.1 Einsatz der Extensible Markup Language (XML)

Die unter 4.4 im Überblick vorgestellte CONTIGRA-Architektur basiert auf strukturierten Dokumenten, mit denen sich von der Schnittstelle über die Komponentenimplementierung bis hin zur Konfiguration, Komposition und Verknüpfung von 3D-Komponenten zu komplexeren Szenen alle Details deklarativ beschreiben lassen. Für einen solchen dokumentzentrierten Lösungsansatz mußten Auszeichnungssprachen für die einzelnen Komponentenebenen entwickelt werden, mit denen eine konsistente und durchgängig deklarative Beschreibung interaktiver 3D-Applikationen möglich wird. Die *Extensible Markup Language (XML)* [XML@], eine Metagrammatik zur Definition kontextfreier Grammatiken, wurde dafür als Basis gewählt. Während die Verwendung dieses einfachen und textbasierten Formates zu Beginn dieser Arbeit noch deutlich seltener vorkam, ist XML jetzt weitverbreitet und soll deswegen hier nicht näher beschrieben werden. Für detaillierte Informationen wird z.B. auf die Ressourcen-Seiten des World Wide Web Consortiums [XML@] verwiesen. Die Vorteile von XML als *Meta-Markup-Language* sind inzwischen in zahlreichen Anwendungsdomänen und Projektkontexten bestätigt worden. Einige sollen hier wiedergegeben werden, um den Einsatz von XML im CONTIGRA-Projekt zu motivieren:

- Plattformunabhängigkeit des Formates;
- Standardisierung und Interoperabilität mit zahlreichen Medien- und Internetstandards;
- Verfügbarkeit von zahlreichen XML-Werkzeugen, -Editoren, -Datenbanken und -Hilfsprogrammen; einheitlicher Zugriff über die Programmierschnittstelle *Document Object Model (DOM)* [DOM@]; Stylesheet-gesteuerte Transformationen in vielfältige Zielformate über *Extensible Stylesheet Language Transformations (XSLT)* [XSLT@] möglich;
- Menschenlesbares Format und gleichzeitig gute Eignung für automatisierte Verarbeitung;
- Strukturierte Beschreibung von Datensätzen, Metadaten, textuellen Dokumentationen, Programmlogik etc. in homogener Weise;
- Eignung für hierarchisch strukturierte Dokumente, somit auch für Szenengraphen bzw. Hierarchien von Komponenten;
- Aktive Weiterentwicklung und Entwicklung zahlreicher verwandter und kompatibler Technologien, darunter XSLT, XPath, XPointer oder XML Query.

Es existieren zwei grundlegende Modelle zur formalen Strukturbeschreibung einer Klasse von XML-Dokumenten. Zunächst die *Document Type Definitions (DTD)* als praktikable Möglichkeit zur einfachen Definition eigener Auszeichnungssprachen. Die Alternative zur Deklaration von Struktur, Inhalt und Semantik von XML-Dokumenten bietet *XML Schema* [XMLSchema@] als Weiterentwicklung von DTDs. XML Schema besitzt einige Vorteile gegenüber DTDs, weshalb es auch für die CONTIGRA-Beschreibungssprachen gewählt wurde. Dazu zählen:

- Unterstützung von Namensräumen *(name spaces)* zur einfachen und homogenen Verwendung der CONTIGRA-Dokumente auf allen Ebenen;
- Erweiterte Datentypen gegenüber stringbasierten bei DTDs, somit Typsicherheit möglich;
- Annäherung an objektorientierte Konzepte durch Unterstützung von Typableitungen, *substitution groups* und abstrakten Datentypen;
- Unterstützung regulärer Ausdrücke;
- Selbstdokumentation, vor allem auch für Werkzeugunterstützung von Bedeutung.

XML Schema ist – im Gegensatz zu XML DTD – selbst mit der XML 1.0 Syntax definiert. Die Abbildung 17 stellt die verschiedenen Ableitungsebenen der XML-Auszeichnungssprachen dar. Dokumente, die einem XML Schema genügen, werden auch als Instanzdokumente bezeichnet. So sind die CONTIGRA-XML-Schemata selbst Instanzdokumente der Definitionssprache XML-Schema. Wenn aber im folgenden Text von Instanzdokumenten die Rede ist, sind damit immer Dokumente gemeint, die mit den CONTIGRA-XML-Auszeichnungssprachen definiert wurden.

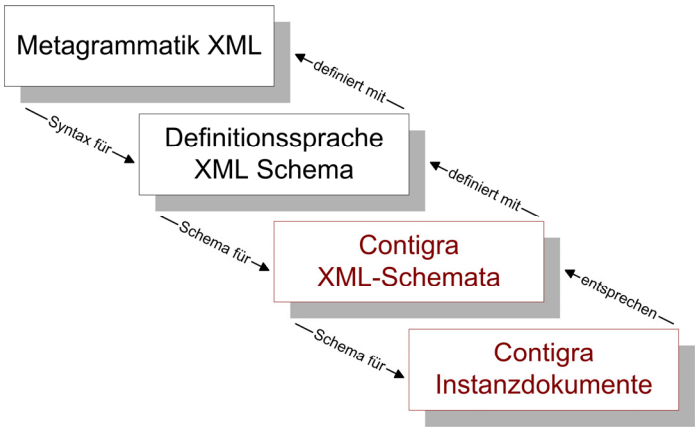

Abbildung 17: Ebenen der CONTIGRA-XML-Dokumente

Ursprünglich wurden Auszeichnungssprachen entwickelt, um Texte strukturieren und annotieren zu können und nicht, um ein Format für numerische Daten bereitzustellen. Da es sich um menschenlesbare Formate handelt, stand und steht Performance oft nicht im Vordergrund. Dies sind Gründe dafür, daß strukturierte Auszeichnungssprachen lange Zeit nicht für die Definition von Austauschformaten für 3D-Grafik verwendet wurden, da die Syntax häufig auch nicht kompakt genug ist. Neben reinen Polygoninformationen gewinnt jedoch die Beschreibung von Metainformationen, High-Level-Szenenstrukturen und Semantik zunehmend an Bedeutung, wodurch XML immer häufiger auch im Bereich 3D-Grafik eingesetzt wird.

So wird XML im V³D²-Teilprojekt *ModNav3D* [ModNav3D@] als Auszeichnungssprache für die Modellierung von 3D-Dokumenten verwendet, wobei semantische Modellinformationen und geometrische Objektrelationen bzw. Konstruktionshierarchien in strukturierten Dokumenten abgelegt werden. Ein weiteres Beispiel ist *Extensible 3D (X3D)* [X3D@]. Neben der klassischen VRML-Syntax bietet X3D als wichtige Neuerung eine XML-Syntax zur Beschreibung von 3D-Szenengraphen an. Dies ist neben der Standardisierung auch der Hauptgrund dafür, daß dieses Format für die interne Szenengraphrepräsentation von CONTIGRA gewählt wurde. Interoperabilität sowohl zu X3D als auch zu SMIL [SMIL@] gewährleistet das *eXtensible MPEG-4 Textual Format (XMT)* [Kim00]. Mit den beiden Profilen XMT-A und XMT-Ω lassen sich dreidimensionale MPEG-4-Szenen mit Hilfe von XML textuell beschreiben, was den Datenaustausch zwischen verschiedenen Autoren, Werkzeugen und Playern erleichtert. Bei den genannten Beispielen handelt es sich nur um wichtige Vertreter. XML wird zudem für zahlreiche domänenspezifische 3D-Anwendungen eingesetzt, darunter zur Definition von chemischen Strukturen, Avataren, verteilten militärischen Simulationen und zur GUI-Beschreibung.

5.2 CoApplication – Beschreibung von 3D-Applikationen

Das XML-Schema *CoApplication* ist eine Beschreibungssprache für interaktive 3D-Applikationen und dient der Integration einer komplexen Komponente in einen Anwendungsrahmen. Instanzdokumente repräsentieren eine deklarative Beschreibung einer 3D-Szene mit ihren typischen Parametern, wie Beleuchtung, Kamera, Klangeigenschaften, Szenenhintergrund und vordefinierten Blickpunkten. Dieser Grammatik genügende Dokumente verweisen zudem auf die eigentlichen interaktiven Szeneninhalte und können in ein konkretes 3D-Zielformat übersetzt werden. Die Abbildung 18 zeigt die Struktur des XML-Schemas *CoApplication* im Überblick. Im Anhang ist eine Legende für diese Strukturabbildung (Tabelle 22) zu finden, der gesamte XML-Quelltext unter [CONTIGRA@].

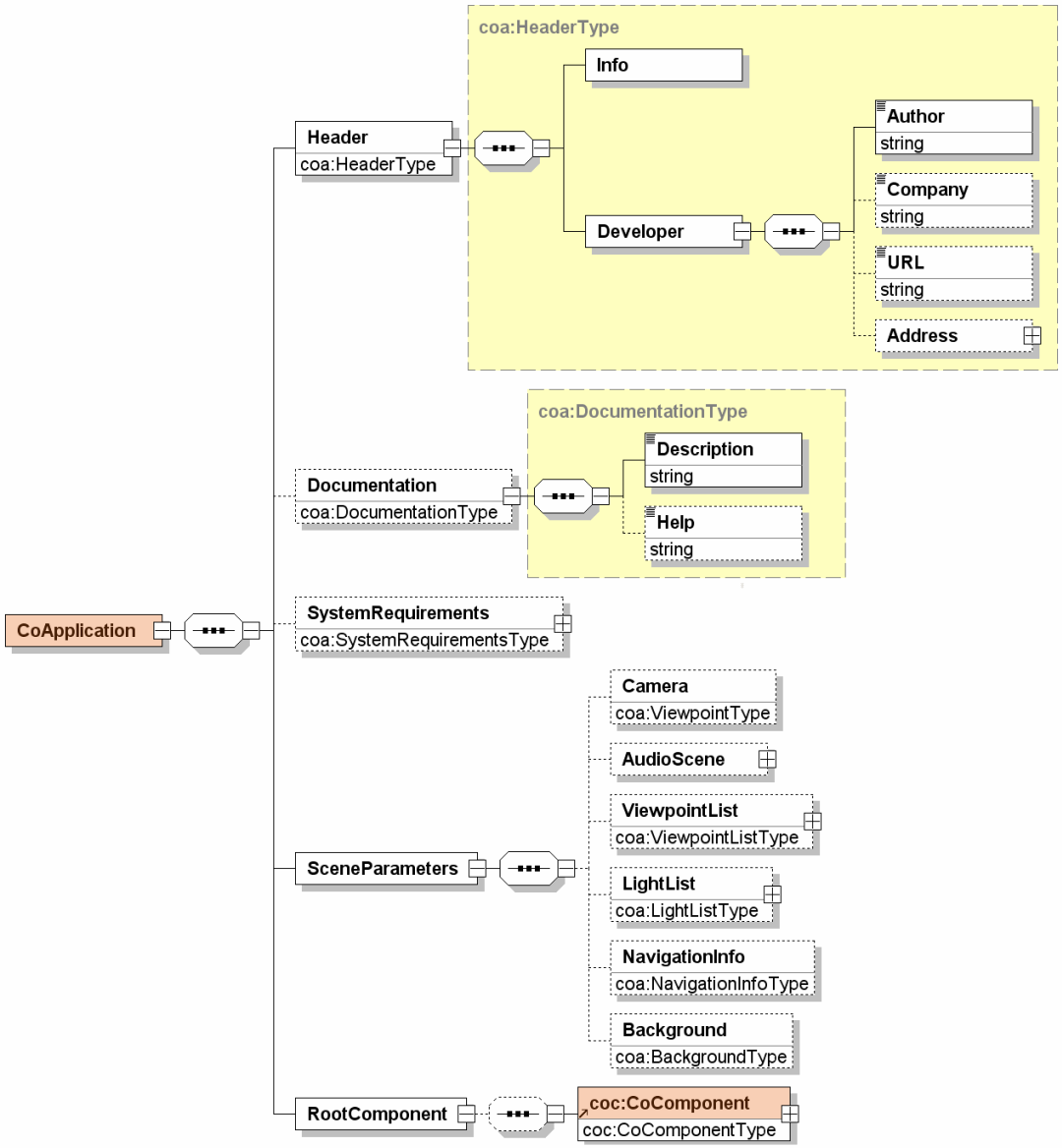

Abbildung 18: Struktur des CONTIGRA-XML-Schemas *CoApplication*

Das verbindliche Element *Header* enthält Anwendungsinformationen, wie Name, ID, Version und letzte Aktualisierung sowie Metainformationen über den Entwickler der 3D-Applikation bzw. Firmenangaben. Die Anwendungsfunktionalität wird innerhalb des optionalen Elementes *Documentation* in Form einer Kurzzusammenfassung und von Hilfe-Informationen bzw. eines Verweises darauf dokumentiert. Ebenfalls optional sind die in *SystemRequirements* de-

finierten Anforderungen der 3D-Applikation an Prozessor, Betriebssystem, benötigten Hauptspeicher, spezielle Treiber, notwendige bzw. unterstützte Ein- oder Ausgabegeräte und minimale Netzwerkverbindungen. Mit Hilfe dieser Unterelemente sollen perspektivisch Adaptionen einer Anwendung an heterogene Endgeräte und Systemkonfigurationen deklariert werden.

Innerhalb der *SceneParameters* werden typische globale Szeneneinstellungen festgelegt, die für die gesamte Anwendung gelten. So können die aktuelle Szenenkamera, eine Liste von vordefinierten Blickpunkten, eine Liste von Lichtquellen, Hintergrundbilder bzw. -farben und Browser-Hinweise beschrieben werden. Dabei sind diese Unterelemente an die entsprechenden X3D-Knoten angelehnt. Hinzu kommt im Element *AudioScene* die Festlegung von globalen Audioeigenschaften, wie z.B. der Position des Hörers oder der Schalldämpfung (s. 5.4.3 zu *Audio3D*-Details).

RootComponent ist schließlich als wichtigstes Element für die tatsächliche 3D-Szene verantwortlich. Die eigentliche Beschreibung der in der Szene enthaltenen Subkomponenten und Szenengraphbestandteile sowie ihrer Verknüpfungen erfolgt in Form einer typischerweise nur als Verweis auf eine *CoComponent*-Instanzdatei referenzierten Wurzelkomponente. In umgekehrter Sichtweise könnte man somit sagen, daß eine komplette 3D-Applikation eigentlich als komplexe Komponente mit Hilfe der XML-Schemas *CoComponent* und *CoComponentImplementation* beschrieben wird und durch eine Instanzdatei von *CoApplication* lediglich um Szeneneigenschaften, Systemanforderungen und Metainformationen zu einer konkreten 3D-Anwendung erweitert wird. Die Beschreibung dieser globalen Eigenschaften in Form einer zusätzlichen Grammatik bzw. eines zusätzlichen Dokuments hat sich als sinnvoll erwiesen, da einzelne Komponenten in verschiedenen Szenen- und Anwendungskontexten wiederverwendet werden können und damit z.B. Informationen zur Kamera innerhalb von Komponentenbeschreibungen wenig sinnvoll wären.

5.3 CoComponent – Beschreibung der Schnittstelle

Jede CONTIGRA-Komponente besteht aus einem Schnittstellendokument als Instanz der Grammatik *CoComponent* und einem Implementierungsdokument als Instanz von *CoComponentImplementation*. Durch diese Aufteilung konnte eine klare Trennung von Schnittstelle und Implementierung einer 3D-Komponente im Sinne des Geheimnisprinzips erreicht werden. Mit dem XML-Schema *CoComponent* wurde eine Grammatik entwickelt, die einerseits zur Definition von Komponentenschnittstellen und andererseits auch zur Deklaration einer spezifischen Komponentenkonfiguration bzw. -ausprägung dient. Damit unterscheidet sich diese Auszeichnungssprache von typischen Schnittstellenbeschreibungssprachen, die nur Methoden und Attribute für Klassen und Komponenten im Sinne einer Java-Interface-Datei oder eines C++-Headers deklarativ beschreiben und dann erst an anderer Stelle instanziiert werden.

Eine aus den beiden genannten Dokumenten bestehende CONTIGRA-Komponente stellt an sich schon eine konkrete Komponentenausprägung dar, die im Sinne der Blackbox-Wiederverwendung allein oder innerhalb einer Komposition in ein konkretes 3D-Zielformat überführt werden kann. Es ist für CONTIGRA-Komponenten allerdings auch möglich, daß für das gleiche *CoComponentImplementation*-Instanzdokument mehrere Schnittstellendokumente gemäß *CoComponent*-Grammatik existieren, die unterschiedliche Ausprägungen bzw. Parametrisierungen einer Komponente darstellen. Ein Schnittstellendokument kann also als Prototyp aufgefaßt werden, der sich kopieren und dann ändern läßt. Üblich ist jedoch die Instanziierung einer Komponente innerhalb einer komplexen Komponente oder Szene, indem ein Konstrukt innerhalb des Komponentengraphs von *CoComponentImplementation* verwendet wird, mit dem nur die im Vergleich zum referenzierten Prototyp (= Schnittstellendokument der Komponente) geänderten Parameter beschrieben werden (s. Beschreibung unter 5.4.1).

Ein *CoComponent*-Instanzdokument bildet durch die Trennung von der Implementierung – innerhalb derer auch auf Szenengraphen verwiesen wird – eine Abstraktionsebene oberhalb von Szenengraphen und eignet sich neben der programmtechnischen Verarbeitung auch für Komponentenkataloge sowie Distribution und Suche von 3D-Komponenten. Die Abbildung 19 zeigt die Struktur des XML-Schemas *CoComponent* im Überblick. Im Anhang ist eine Legende für diese Strukturabbildung (Tabelle 22) zu finden, der gesamte XML-Quelltext unter [CONTIGRA@]. Die Bestandteile der Struktur werden in den folgenden Unterkapiteln näher erläutert.

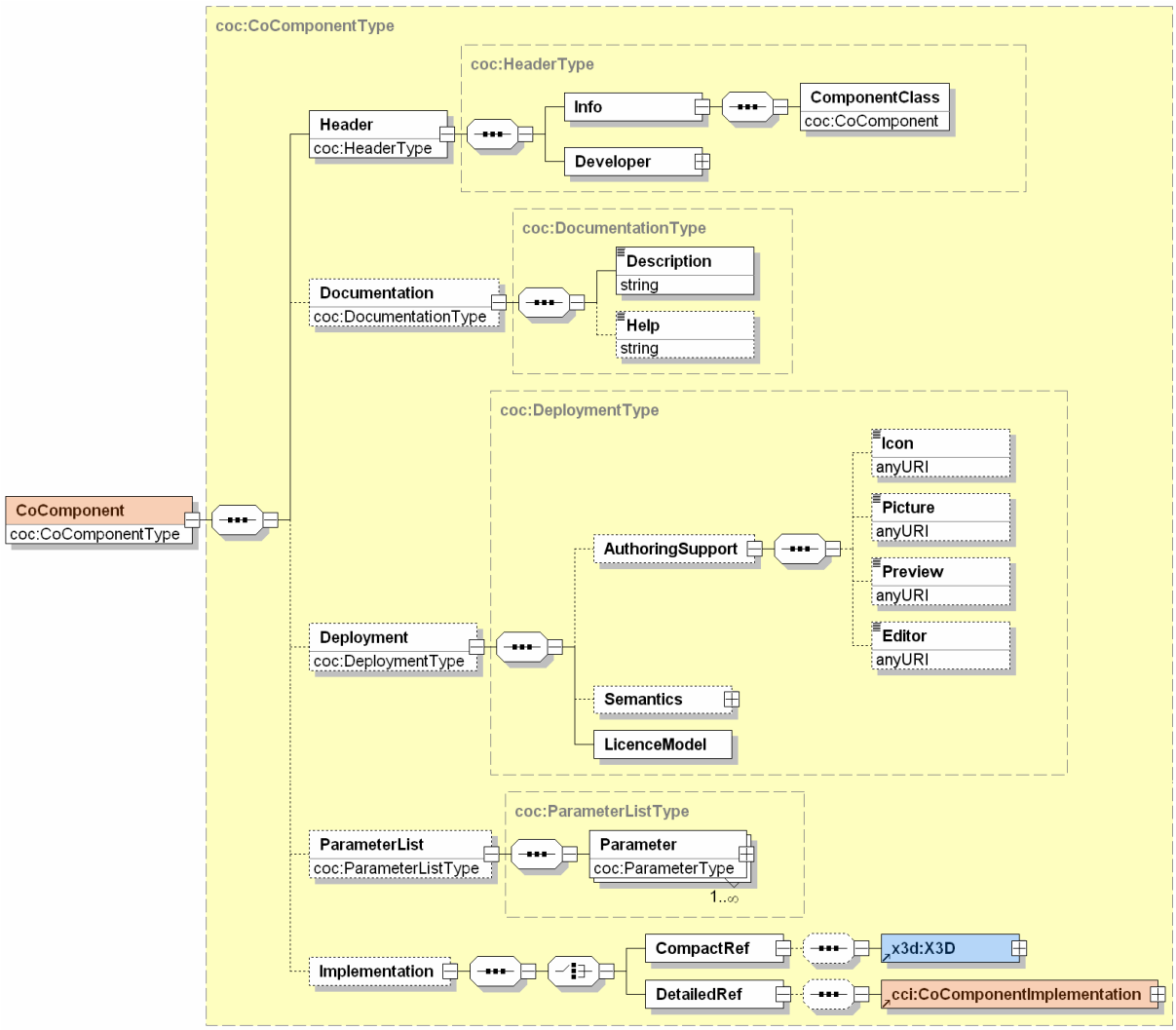

Abbildung 19: Struktur des CONTIGRA-XML-Schemas *CoComponent*

5.3.1 Grundlegende Metainformationen und Dokumentation

Im Element *Header* werden typische Metainformationen zur beschriebenen Komponente abgelegt. Analog zu den Angaben bei *CoApplication* sind das Informationen über den Namen, Entwickler, die Firma, Version und letzte Revision einer Komponente. Dazu kommt noch die Spezifikation einer Komponentenklasse, wobei es sich nicht um eine Klasse im softwaretechnischen Sinne handelt, sondern um die Zuordnung der konkreten Komponente zu ihrem ursprünglichen Komponentenprototypen bzw. der unter 3.4 vorgenommenen Klassifikation von 3D-Widgets. Ein Beispiel dafür ist die Komponentenklasse *CoButton* innerhalb der Widget-Hierarchie, von der es für eine bestimmte Firma eine eigene, abgeleitete Komponente *xyzButton* geben kann, bei der im Unterschied zum Prototypen bestimmte Parameter – z.B. die Erscheinung des Buttons – modifiziert sind. Wie bei *CoApplication* enthält das optionale Element *Documentation* eine Beschreibung und evtl. auch Benutzungshilfe für die Komponente.

5.3.2 Unterstützung des Autorenprozesses und Implementierung

Der optionale *Deployment*-Teil enthält wichtige Informationen für den Autorenprozeß, seine Werkzeuge und den generellen Einsatz einer Komponente. So sind im Element *AuthoringSupport* in Form von Referenzen verschiedene Bilder, Icons und ein 3D-Preview (üblicherweise als VRML97- oder X3D-Datei) enthalten. Dazu kommt ein möglicher Verweis auf ein Editor-Plug-In für ein Autorenwerkzeug wie den CONTIGRABUILDER (s. Kap. 6), mit dem über das an Datentypen orientierte Editieren von Komponentenparametern hinaus ein komfor-

tableres oder intuitiveres Anpassen einer Komponente möglich ist. Im *Semantics*-Element kann enthalten sein, inwieweit sich eine Komponente für ein bestimmtes Einsatzgebiet eignet, wie universell sie z.B. verwendet werden kann, mit welchen Komponenten sie sich gut kombinieren läßt oder zu welchen sie eine Alternativrealisierung darstellt. Die konkrete Interpretation dieses prototypisch entworfenen Bestandteils, dessen Inhalt sich nur schwer formalisieren läßt, wurde jedoch noch nicht näher betrachtet. Gleiches gilt für das Element *LicenceModel*, mit dem die gewünschte oder mögliche Form der Lizenzierung festgelegt werden kann.

Die sich in der Dokumentstruktur anschließende optionale Parameterliste wird im folgenden Unterkapitel beschrieben. Mit dem Element *Implementation* – das optional ist, weil eine Komponente auch nur als Schnittstellenspezifikation existieren kann – wird auf die konkrete Implementierung einer Komponente in Form von Szenengraphen, Subkomponenten und ihren Verknüpfungen verwiesen. Diese Struktur wird für die Phase der Komponentenentwicklung in einem Instanzdokument von *CoComponentImplementation* (s. 5.4) beschrieben. Eine solche Datei wird im Element *DetailedRef* typischerweise referenziert. Parallel dazu gibt es für die Entwicklungsphase im Element *CompactRef* einen Verweis auf eine X3D-Szenengraphdatei, welche die komplette Implementierung einer Komponente in kompakter Form beinhaltet. Damit können die interne Realisierung einer Komponente und ihre Bestandteile verborgen werden. Das Dokument ist das Ergebnis eines Übersetzungsprozesses, der aus der Beschreibung der Implementierung einer Komponente eine (potentiell komplexe) X3D-Datei generiert. In Analogie zur Programmierung könnte man sagen, daß Programmcode kompiliert wird, nur daß es sich hier um eine weitere Metaebene handelt, weil X3D selbst in einem Player ausgeführt wird oder in ein weiteres 3D-Format transformiert werden kann.

5.3.3 Das Parameterkonzept

Typische Softwarekomponenten der unter 4.2 beschriebenen Technologien favorisieren aufgrund ihres inhärent imperativen Ansatzes die Änderung von Eigenschaften einer Komponente mit Hilfe von Methoden. Während bei der ersten Vorstellung des CONTIGRA-Ansatzes in [Dachs02] noch eine Mischform aus deklarativ beschriebenen Methoden und Parametern gewählt wurde, werden CONTIGRA-Komponenten inzwischen ausschließlich über ihre Parameter konfiguriert. Deren Änderungen können Ereignisse auslösen, die bestimmte Komponentenfunktionalität realisieren. Darin ähnelt der CONTIGRA-Ansatz dem ereignisorientierten VRML97-Konzept. Das Element *ParameterList* kann die Beschreibung mindestens eines oder beliebig vieler Parameter enthalten. Der folgende Ausschnitt aus einem *CoComponent*-Instanzdokument für ein Ringmenü zeigt beispielhaft die Definition eines Parameters.

```
<!-- Deklaration des Parameters RingRadius der Contigra-Komponente RingMenu -->
<Parameter name="RingRadius" description="Radius of the items' ring"
           dataType="CoFloat" semantics="appearance"
           forDesigner="true" forProgrammer="false" forSoundEngineer="false"
           configurable="true" receivesEvents="false" generatesEvents="false">
    <cpt:CoFloat>10.0</cpt:CoFloat>
</Parameter>
```

Die Abbildung eines Parameters auf konkrete Bestandteile der Komponentenimplementierung erfolgt erst in einem Instanzdokument von *CoComponentImplementation* im Element *InterfaceParameterLinks* und wird unter 5.4.6 näher beschrieben. Jede Parameterdeklaration enthält als Unterelement einen CONTIGRA-Parameterdatentyp mit konkretem Wert sowie verschiedene Attribute, die im folgenden näher erläutert werden sollen.

5.3.3.1 Grundlegende Parameterinformationen

Der Name des Parameters in der Komponentenschnittstelle wird mit dem Attribut *name* festgelegt und spielt eine wichtige Rolle für die spätere Definition von Verknüpfungen. Mit *des-*

cription wird eine kurze Beschreibung des Parameters und seiner Verwendung gegeben. Damit wird verbal ein Teil der Funktionalität der Komponente ausgedrückt, die z.B. als Tooltip-Text in Anwendungen erscheinen kann. Wichtig für die Verknüpfung von Parametern ist der in *dataType* festgelegte Parameterdatentyp, der einem der CONTIGRA-Datentypen genügen muß. Diese werden im separaten XML-Schema *CoParameterTypes* definiert. In vielen Fällen handelt es sich um Abbildungen auf X3D-Datentypen für Knoten und Felder, die Kapselung macht den CONTIGRA-Ansatz jedoch unabhängig und erlaubt flexible Erweiterungen. So ist das XML-Schema für X3D zum Zeitpunkt des Schreibens dieser Arbeit nach wie vor nicht in einem validierbaren Zustand und kommt damit für eine Typdefinition nicht unmittelbar in Frage.

5.3.3.2 Parameterkategorie

Mit dem Attribut *semantics* läßt sich festlegen, zu welcher Kategorie ein Parameter gehört. Dies ist insbesondere für den Autorenprozeß und die Darstellung in Komponentenkatalogen sehr sinnvoll, da sich differenziert erkennen läßt, inwieweit sich eine Komponente konfigurieren läßt und nicht sämtliche Parameter in nur einer langen Liste angeboten, sondern nach ihrer Semantik geordnet werden können. Die folgenden Werte sind möglich:

- *geometry:* Ein direkt austauschbarer Geometrieteil, üblicherweise ein Verweis auf eine VRML97- oder X3D-Datei, in jedem Fall auf einen geometrischen Szenengraph.
- *appearance:* Ein Parameter, der das Aussehen der Komponente beschreibt, z.B. Farbe, Material oder Textur, aber auch Radius, Abstände etc.
- *audio:* Ein direkt austauschbarer Audiobestandteil (Verweis auf einen Knoten eines Audio-Szenengraphen oder eine Sounddatei) oder ein Parameter für andere Audioeigenschaften.
- *behavior:* Ein Parameter, der das funktionale Verhalten der Komponente im weitesten Sinne beschreibt, z.B. ihren Zustand, eine Drehgeschwindigkeit oder aktuell gewählte Werte.
- *general:* Alle Parameter, die sich nicht sinnvoll in eine der zuvor genannten Kategorien einordnen lassen.

5.3.3.3 Sichtbarkeit für Autorenrollen

Gerade für den Entwurf interdisziplinärer Autorenwerkzeuge (s. Anforderungen unter 4.1) ist es wichtig, nicht nur die Semantik der Parameter zu berücksichtigen, sondern auch ihre Eignung bzw. Sichtbarkeit für bestimmte Autorenrollen. Mit einem Attribut der Attributgruppe *authorRole* kann deshalb festgelegt werden, für welche Autorengruppe ein Parameter primär editierbar sein soll. Die einzelnen Boolean-Attribute *forProgrammer, forDesigner* und *forSoundEngineer* können unabhängig voneinander gesetzt werden und legen die Sichtbarkeit für Programmierer, Designer und Toningenieure respektive fest. Damit können Editoren bestimmte Parameter einer Komponente auf Wunsch ausblenden oder auch spezielle Editorwerkzeuge aktivieren.

5.3.3.4 Zeitpunkt und Art der Änderung von Parametern

Parameterdefinitionen enthalten drei Attribute, die Auskunft über den möglichen Zeitpunkt der Änderung eines Parameters und der damit verbundenen Ereignisverarbeitung geben. So läßt sich mit den voneinander unabhängigen Boolean-Attributen *configurable, receivesEvents* und *generatesEvents* festlegen, ob der Wert eines Parameters bereits während der Instanziierung festgelegt werden kann oder erst zur Laufzeit durch das Empfangen eines Ereignisses. Schließlich kann auch deklariert werden, ob bei der Änderung des Parameters zur Laufzeit ein Ereignis generiert wird und sich so der Parameter mit anderen ereignisempfangenden Parame-

tern verknüpfen läßt. Ist das Attribut *configurable* auf *true* gesetzt, muß als Unterelement eines Parameters auch ein Default-Wert angegeben werden. Erfolgt eine Wertezuweisung erst zur Laufzeit, genügen die Attributdefinitionen, ein bereits angegebener Parameterwert wird dann ignoriert. Als Kombination der drei genannten Attribute ergeben sich acht verschiedene Möglichkeiten, die in Tabelle 19 dargestellt sind. Dabei sind zum Vergleich die Feldtypen angegeben, die man bei VRML97 bzw. X3D Feldern zuweisen kann. An dieser Stelle wird deutlich, daß die Festlegung des Änderungszeitpunktes für CONTIGRA-Komponentenparameter eine reichere Ausdrucksmöglichkeit gegenüber VRML97/X3D-Feldtypen gestattet, wodurch Schwierigkeiten eliminiert werden konnten. Für eine nähere Beschreibung des auch für Verhaltensknoten verwendeten Konzeptes und einen Vergleich zum 3D-Standardformat wird auf [Dachs03a] verwiesen.

	Wertekombination			entspricht VRML97- / **X3D**-Feldtyp	Bemerkungen
	configurable	*receives Events*	*generates Events*		
1	false	false	false	–	ohne praktische Bedeutung
2	false	false	true	eventOut / **outputOnly**	Kein direktes Setzen von Parametern, aber Ereignisgenerierung. Interessant für inneres Komponentenverhalten.
3	false	true	false	eventIn / **inputOnly**	Parameter kann nur durch empfangene Ereignisse zur Laufzeit gesetzt werden.
4	false	true	true	–	Parameter kann nur zur Laufzeit gesetzt / geändert werden, nicht zur Autorenzeit. Geeignet z.B. für Zuweisung einer dynamischen IP-Adresse.
5	true	false	false	field / **initializeOnly**	Parameteränderung nur zur Autorenzeit, z.B. Schriftart.
6	true	false	true	–	Typischerweise Zustandswert, der zur Autorenzeit festgelegt und durch Komponentenverhalten geändert wird.
7	true	true	false	–	Parameter kann zur Konfigurationszeit und zur Laufzeit durch empfangene Events gesetzt werden, z.B. Farbe.
8	true	true	true	exposedField / **inputOutput**	Parameter kann zur Konfigurationszeit und zur Laufzeit gesetzt werden und generiert bei Änderung selbst Events.

Tabelle 19: Mögliche Wertekombinationen der Parameterattribute *configurable, receivesEvents* und *generatesEvents* innerhalb einer CONTIGRA-Komponentenschnittstelle

Da Komponenten auch wieder Subkomponenten mit Parametern enthalten können, stellt sich die Frage, ob überhaupt und welche davon an der Schnittstelle der Containerkomponente sichtbar sein sollen. Dazu werden im entsprechenden *CoComponent*-Instanzdokument genau die gewünschten Parameter (erneut) deklariert, die von der Containerkomponente angeboten werden können. So kann bei der Festlegung der Komponentenschnittstelle im Autorenwerkzeug die Liste sämtlicher Parameter der enthaltenen Subkomponenten angeboten werden, aus der dann die gewünschten gewählt werden können. In jedem Fall erfolgt die Abbildung dann in Form eines Verweises auf den gewünschten Parameter einer Subkomponente.

5.4 CoComponentImplementation – Beschreibung der Implementierung

Im Unterkapitel 5.3 wurde bereits die klare Trennung von Schnittstelle und Implementierung einer 3D-Komponente erläutert. Instanzdokumente des XML-Schemas *CoComponent-Implementation* beschreiben den Implementierungsteil einer Komponente. In der Abbildung 20 ist die Struktur des XML-Schemas *CoComponentImplementation* im Überblick zu sehen, eine Abbildungslegende (Tabelle 22) findet sich im Anhang, der XML-Quelltext unter [CONTIGRA@].

Eine Komponentenimplementierung setzt sich primär aus einem Komponentengraph *(ComponentGraph)* und einem Szenengraph *(SceneGraph)* zusammen. Dabei handelt es sich jeweils um eine Transformationshierarchie von aggregierten Komponenten *(SubComponents)* respektive verwendeten Subszenengraphen *(SubSceneGraphs)* sowie den unten (s. 5.4.5) erläuterten Verknüpfungskonstrukten für die jeweilige Hierarchie (Elemente *ComponentLinks* und *SceneGraphLinks*). Elementare Komponenten, die vollständig neu erstellt werden, haben typischerweise nur Szenengraphbestandteile und keine Subkomponenten. Komponenten, die ausschließlich aus der Kombination von Subkomponenten ohne jegliche Szenengraphbestandteile auskommen, werden eher selten anzutreffen sein. Mischformen, bei denen Subkomponenten genauso wie zusätzliche Szenengraphbestandteile eingesetzt werden, stellen den häufigsten Fall dar. In den folgenden Unterkapiteln werden der Komponentengraph, die verschiedenen Szenengraphen und dafür neu entwickelten Sprachen sowie das Verknüpfungskonstrukt näher erläutert. Schließlich wird unter 5.4.6 dargestellt, wie im dritten Hauptelement der Komponentenimplementierung – *InterfaceParameterLinks* – die Verbindung zwischen Schnittstellenparametern und Implementierungsbestandteilen etabliert werden kann.

5.4.1 Der Komponentengraph

Zentraler Gedanke der CONTIGRA-Architektur ist die Assemblierung komplexer interaktiver 3D-Anwendungen aus vorgefertigten Komponenten. Somit kommt dem Element *SubComponents* als Teil des *ComponentGraph* eine wichtige Bedeutung zu, da hier die Transformationshierarchie von konfigurierten Komponenten beginnt. Mit dem an X3D-Transformationsknoten angelehnten *Transform-Element* lassen sich beliebige Hierarchien aufbauen, innerhalb derer Subkomponenten gruppiert und im Raum positioniert werden können. Dabei gibt es zwei grundsätzliche Möglichkeiten der Integration von Komponenten (hier gleichbedeutend zu ihrer konfigurierten Schnittstellenbeschreibung, wie in 5.3 beschrieben). Der seltenere Fall wäre die komplette Auflistung einer Komponentenschnittstelle inklusive konfigurierter Parameter in Form eines der Grammatik *CoComponent* genügenden XML-Fragments. Eine solche Variante würde man zur Reduzierung der Dateianzahl wählen, wenn am Ende möglichst ein einzelnes CONTIGRA-XML-Dokument existieren soll.

Üblich ist jedoch die Referenzierung einer Komponente bzw. ihre Instanziierung innerhalb des *ComponentInstance*-Elements. Dabei wird mit dem Attribut *fileRef* eine Referenz zu einem Instanzdokument von *CoComponent* etabliert. In diesem Fall wird die in der externen Datei beschriebene Komponente in dieser Form als Subkomponente in die Transformationshierarchie eingebunden. Es besteht jedoch weiterhin die Möglichkeit, durch die Angabe von *Parameter*-Deklarationen im Unterelement *ParameterList* genau die Parameter zu konfigurieren, die sich bei der Komponenteninstanz im Vergleich zur externen Datei unterscheiden sollen. Damit ist eine elegante und platzsparende Methode geschaffen worden, Komponenteninstanzen anzupassen und trotzdem externe Referenzen beizubehalten.

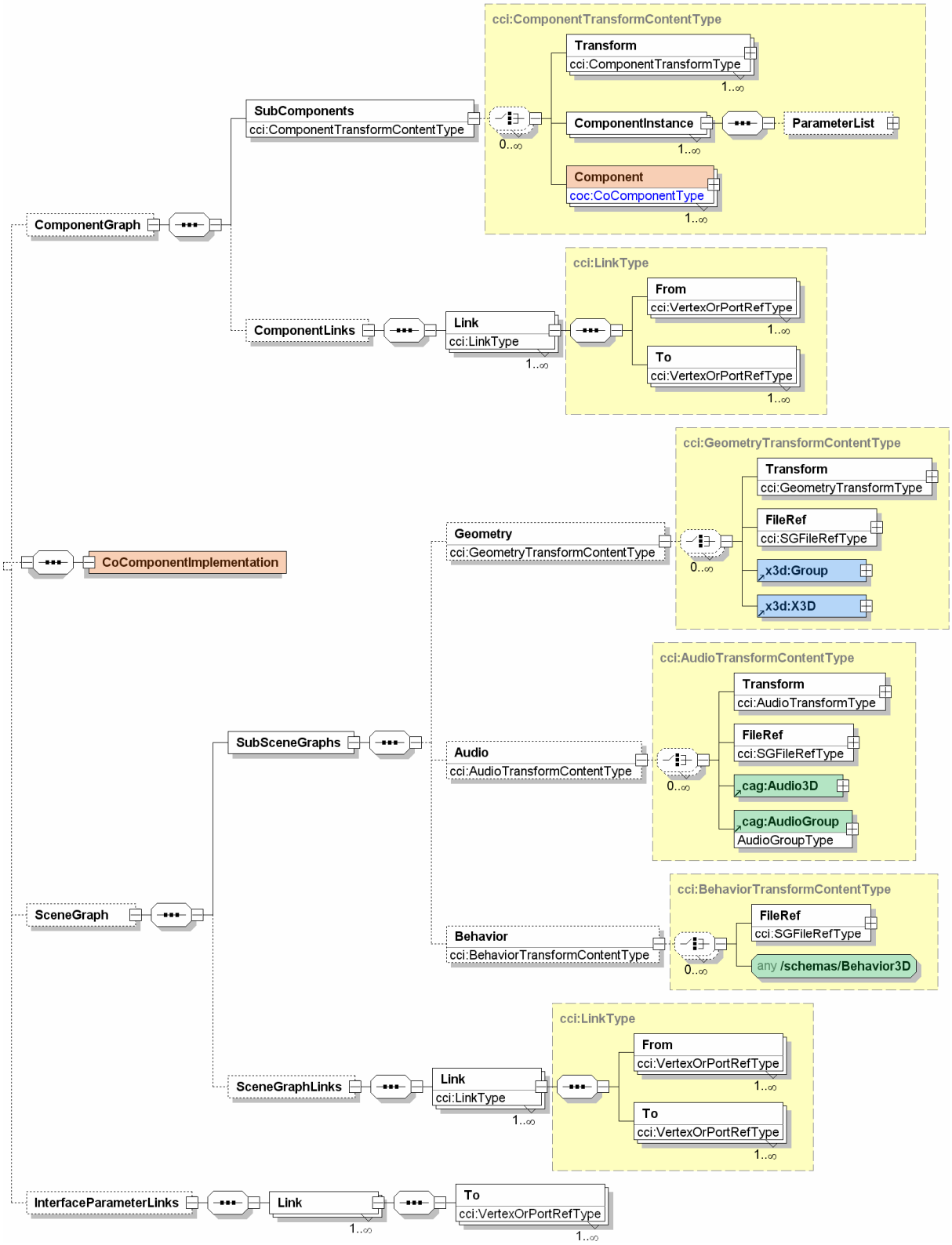

Abbildung 20: Struktur des CONTIGRA-XML-Schemas *CoComponentImplementation*

Das folgende Codefragment eines aus drei Schiebereglern bestehenden Farbwählers verdeutlicht, wie eine Komponente mehrfach, jedoch mit angepaßten Parametern, innerhalb des Komponentengraphs verwendet werden kann.

```
<!-- Ausschnitt der Komponentenimplementierung der Contigra-Komponente ColorSlider -->
<ComponentGraph>
    <Subcomponents>
        <Transform translation="0 4 0" DEF="SliderRedTransform">
            <ComponentInstance fileRef="Slider/Slider.coc" DEF="SliderRed">
                <ParameterList>
                    <Parameter name="SelectorGeometry">
                        <cpt:CoGeometryGroup>
                            <cpt:CoAnyURI>Ressources/SliderRed_SelectorGeometry.x3d</cpt:CoAnyURI>
                        </cpt:CoGeometryGroup>
                    </Parameter>
                </ParameterList>
            </ComponentInstance>
        </Transform>
        <Transform translation="0 1 0" DEF="SliderGreenTransform">
            <ComponentInstance fileRef="Slider/Slider.coc" DEF="SliderGreen">
                <ParameterList>
                    <Parameter name="SelectorGeometry">
                        <cpt:CoGeometryGroup>
                            <cpt:CoAnyURI>Ressources/SliderGreen_SelectorGeometry.x3d</cpt:CoAnyURI>
                        </cpt:CoGeometryGroup>
                    </Parameter>
                </ParameterList>
            </ComponentInstance>
        </Transform>
        …
    </Subcomponents>
    …
</ComponentGraph>
```

5.4.2 Getrennte Szenengraphen: Geometrie, Audio, Verhalten

Bei den Szenengraphbestandteilen einer Komponentenimplementierung (Element *SceneGraph*) wurde innerhalb des Unterelements *SubSceneGraphs* eine klare Trennung zwischen Szenengraphbestandteilen für Geometrie *(Geometry)*, auditive Aspekte *(Audio)* und Verhaltensrealisierungen *(Behavior)* vorgenommen. Die Vorteile dieser Aufteilung liegen in besserer Wiederverwendbarkeit durch Separierung, in besserer Übersicht und Wartbarkeit und in einfacher Austauschbarkeit. Zudem können separate Graphen von spezialisierten Playern oder APIs auch besser und schneller traversiert werden. Die Idee der Trennung von Geometrie- und Verhaltensgraph wird bereits bei den interaktiven, animierten 3D-Widgets von Döllner und Hinrichs [Dölln98] vorgestellt und bei den *Three-dimensional Beans* von Dörner und Grimm [Dörne00] mit den *Meta Beans* [Dörne01] als Komponenten zur Kapselung von Interaktionsmetaphern aufgegriffen. Während man auditive Informationen auch dem Erscheinungsbild einer Szene zuordnen könnte, spielen sie doch beim Rendern in einem 3D-Viewer im Gegensatz zu Texturen, Farben oder Videos zunächst keine Rolle. Deshalb wurden diese Informationen ebenfalls in einem separaten Graph gekapselt, der für das auditive Rendern Vorteile bietet.

Das ursprüngliche Ziel, ein eigenes neutrales Szenengraphformat für die deklarative CONTIGRA-Implementierung zu entwickeln, wurde verworfen, da es eine aufwendige und unnötige Re-Implementierung bedeutet und sich nicht an existierenden Standards orientiert hätte. Deshalb wurde für die interne Szenengraphbasis X3D gewählt. Für die Beschreibung von Geometrie und Erscheinungsbild bietet das Szenengraphformat zahlreiche Knoten, weist jedoch erhebliche Defizite bei der Integration von Raumklang und eine nur geringe Ausdrucksmächtigkeit für die deklarative Verhaltensbeschreibung auf. Eine direkte Erweiterung des Web3D-Standards mit den Mitteln *X3D-Component* oder durch *X3D-Profiles* wäre zwar

wünschenswert gewesen, erwies sich jedoch zum Zeitpunkt der Entwicklung der CONTIGRA-Architektur als undurchführbar. Das liegt am aktuellen Entwicklungsstand von X3D, am nach wie vor nicht validierbaren X3D-XML-Schema und vielen ungeklärten Fragen im Zusammenhang mit X3D-Erweiterungsmechanismen. So wurden für den Audio- und Verhaltensszenengraph eigene Sprachen entwickelt, die sich jedoch auf X3D abbilden lassen und perspektivisch mit Techniken zur Erweiterung von X3D umsetzbar sind. *Audio3D* und *Behavior3D* werden in den folgenden Unterkapiteln 5.4.3 und 5.4.4 vorgestellt.

Innerhalb der Elemente *Geometry* und *Audio* können mit Hilfe des Elementes *Transform* beliebige Transformationshierarchien aufgebaut werden. Für den *Behavior*-Teil ist das nicht nötig, weil die Anordnung von rein funktionalen Verhaltensknoten im Raum keinen Sinn ergeben würde. Bei allen drei Graphen ist die Angabe einer Dateireferenz mit *FileRef* möglich, wobei als Inhalt dann die jeweils in Abbildung 20 farbig gekennzeichneten Typen erwartet werden. Blau steht für X3D-Knoten, grün für die CONTIGRA-eigenen Szenengraphgrammatiken. Alle Typen können auch direkt als XML-Fragment in der jeweiligen Hierarchie integriert sein. Im Falle von *Geometry* ist dafür ein X3D-Gruppenknoten erlaubt bzw. der Wurzelknoten von X3D, das Element *X3D*. Für den Audiograph unter *Audio* kann eine Audiogruppe *(AudioGroup)* oder das Wurzelelement der Audio3D-Grammatik *(Audio3D)* integriert werden. Beim Verhaltensgraph kann unter dem Element *Behavior* neben der Dateireferenz auch ein beliebiger Verhaltensknoten aus dem im XML-Schema *Behavior3D* beschriebenen Repertoire von Knoten aufgeführt werden.

Das folgende XML-Codefragment zeigt Ausschnitte des Geometrie- und Verhaltensgraphs für eine interaktive Laptop-Komponente. Dabei ist zu sehen, wie Dateireferenzen integriert, geometrische Transformationshierarchien aufgebaut und Verhaltensknoten verwendet werden.

```
<! -- Ausschnitt des Szenengraphteils der Implementierung einer Contigra-Komponente -->
<SubSceneGraphs>
    <Geometry>
        <Transform DEF="LCD_Transform" center="0 0.0151 -0.2051">
            <FileRef xlink:href="Ressources/lcd_without_texture.x3d"/>
            <Transform center="3.84 -0.328 -0.2" rotation="1 0 0  1.5708" scale="0.002 0.002 0.002">
                <x3d:Group>
                    <Shape>
                        <Appearance>
                            <Material specularColor="0.3 0.3 0.3"/>
                            <ImageTexture DEF="lcd_texture" repeatS="true" repeatT="true"/>
                        </Appearance>
                        …
                    </Shape>
                </x3d:Group>
            </Transform>
        </Transform>
        …
    </Geometry>
    <Behavior>
        <b3d:TouchSensor DEF="LCD_Sensor"/>
        <b3d:Sequential DEF="OpenKeyboard">
            <b3d:AnimateTranslation DEF="OpenKeyboard_Translation" key="0 1" to="…" cycleIntervall="1"/>
            <b3d:AnimateRotation DEF="OpenKeyboard_Rotation" key="0 1" to="…" cycleIntervall="1"/>
        </b3d:Sequential>
        …
    </Behavior>
</SubSceneGraphs>
```

In diesem XML-Codebeispiel und auch im unter 5.4.1 aufgeführten Komponentengraphbeispiel wird deutlich, daß mit dem Attribut *DEF* verschiedenste Bestandteile einer Implementierung benannt werden können. Diese Bezeichnung erweist sich u.a. als notwendig für die Referenzierung von Subszenengraphen und -komponenten innerhalb der Verknüpfungskonstrukte, die im Unterkapitel 5.4.5 beschrieben werden.

Wann immer Dateireferenzen in einem der Szenengraphteile mit dem Element *FileRef* eingesetzt werden, läßt sich über das Attribut *xlink:href* nicht nur der Speicherort der Datei, sondern mit Hilfe von XPath-Ausdrücken auch ein bestimmter Teil davon ansprechen. Weiterhin besteht mit Hilfe von sogenannten *DEFassociations* als Unterelementen von *FileRef* die Möglichkeit, Teile des in der externen Datei abgelegten Szenengraphen mit DEF-Attributen zu versehen, ohne die Benennung direkt in die Datei einfügen zu müssen. Dafür kommt ein XLink *Locatortype* zum Einsatz, auch XPath-Ausdrücke können zur genauen Bestimmung des zu benennenden Szenengraphteils verwendet werden.

5.4.3 Audio3D – Integration von Raumklang

Gerade in Desktop-VR-Umgebungen spielt Sound eine wesentliche Rolle zur Vermittlung eines Immersionsgefühls als Substitution für spezialisierte Ein- und Ausgabegeräte. So schlägt [Hand97] zum Beispiel die Nutzung von Sound als Ersatz für haptische Empfindungen vor. Ein Mangel verschiedener 3D-Formate – darunter VRML97/X3D – ist ihre oft unzureichende Beschreibungsmächtigkeit für Raumklang. Deshalb wurde in [Hoffm01] ein eigenes XML-Format *Audio3D* zur Beschreibung von Raumklang in VR-Umgebungen entwickelt. Dieses deklarative Format bildet die Grundlage für den Audiograph von CONTIGRA-Komponenten, ist jedoch auch eigenständig verwendbar und läßt sich auf 3D-Sound-APIs abbilden. Aus Platzgründen kann es hier nur im Überblick vorgestellt werden, weitere Details sind in [Hoffm01] und [Hoffm03] zu finden.

5.4.3.1 Eigenschaften und Beschreibungsebenen

Das entwickelte deklarative Format *Audio3D* dient der Beschreibung von akustischen Umgebungsparametern und Soundquellen für auditive Szenendarstellungen und besitzt folgende Eigenschaften:

- Unabhängigkeit von speziellen Plattformen und Sound-APIs;
- Realisierung mit XML-Schema, lizenzfreies Format;
- Konzept eines hierarchischen Szenengraphs;
- Kombinierbarkeit mit anderen XML-basierten 3D-Grafikformaten;
- Beschreibung von akustischen Umgebungen mit mehreren Räumen in beliebigen Detailstufen durch reflektierende und absorbierende Oberflächen sowie Hallparameter;
- Orientierung an der Funktionalität von 3D-Sound-APIs, die Richtlinien der *Interactive Audio Special Interest Group (IASIG)* [IASIG@] berücksichtigen.

Der Audio-Szenengraph des mit XML-Schema entwickelten Formates *Audio3D* beschreibt animierte Soundquellen, Hörerpositionen, Schallwellen reflektierende und absorbierende Hindernisse, abstrakte geometrische Räume mit Hallparametern und Klangeigenschaften von Materialien. Diese Beschreibungsmöglichkeiten wurden in drei verschiedene Grammatiken mit unterschiedlicher Ausdrucksmächtigkeit aufgeteilt, um unterschiedlichen Anforderungen und Einsatzfällen gerecht werden zu können. Die Tabelle 20 zeigt die drei Ebenen im Überblick und beschreibt ihre jeweiligen Eigenschaften.

Ebene	*Core*	*Common*	*Full*
Charakteristik	Generelle Struktur, essentielle Eigenschaften des Audio-Szenengraphs *(IASIG Level 1 konform)*	Erweiterung um Beschreibungsmöglichkeiten für Raumakustik *(IASIG Level 2 konform)*	Erweiterung um Beschreibungsmöglichkeiten für sehr komplexe Audioszenen
Beschreibung von:	Szenengraph mit hierarchischen Transformationen, Entfernungsdämpfung, Dopplereffekt, Hörerposition, Punktquellen, Animationen	Materialeigenschaften, Verdeckungseffekte, Raumgeometrien mit Öffnungen, Halleigenschaften, Level-of-Detail-Stufen, Prioritäten	Einzelreflektionen, Mediumeigenschaften, Größe von Soundquellen, spezielle Soundquellen
Anwendungsfälle	Interaktionsfeedback, Nutzerinformationen, Musikanwendungen ohne Raumsimulation	Telekommunikation in virtuellen Räumen, Entertainment, virtuelle Gebäuderundgänge	Virtuelle Realität, wirklichkeitsnahe Soundsimulation, z.B. für Gebäudeakustik
Implementierung (Player)	viele Features von aktuellen Sound-APIs bereitgestellt, geringer Programmieraufwand	nur teilweise von aktuellen Sound-APIs unterstützt, höherer Programmieraufwand	kaum von aktuellen Sound-APIs unterstützt, sehr hoher Programmieraufwand

Tabelle 20: Die drei XML-Schemata von *Audio3D* und ihre Ausdrucksmächtigkeit

5.4.3.2 Die Realisierung mit XML-Schema

Die gezeigten drei Ebenen wurden in den XML-Schemata *Audio3d_core, Audio3d_common* und *Audio3d_full* realisiert. Dabei bildet *Audio3d_core* die Basis, auf der die anderen Schemata hierarchisch in der angegebenen Reihenfolge aufbauen. Jede XML-Grammatik fügt der zugrundeliegenden eine Menge neuer Elemente und Attribute hinzu. Somit ähnelt dieses Konzept dem der Profile in X3D und erlaubt z.B. unterschiedliche Implementierungsvarianten von Playern, die zu einer der drei Stufen konform sind. In der Abbildung 21 ist die Struktur des XML-Schemas *Audio3D_core* im Überblick zu sehen, eine detailliertere Darstellung findet sich in [Hoffm01] bzw. online unter [CONTIGRA@].

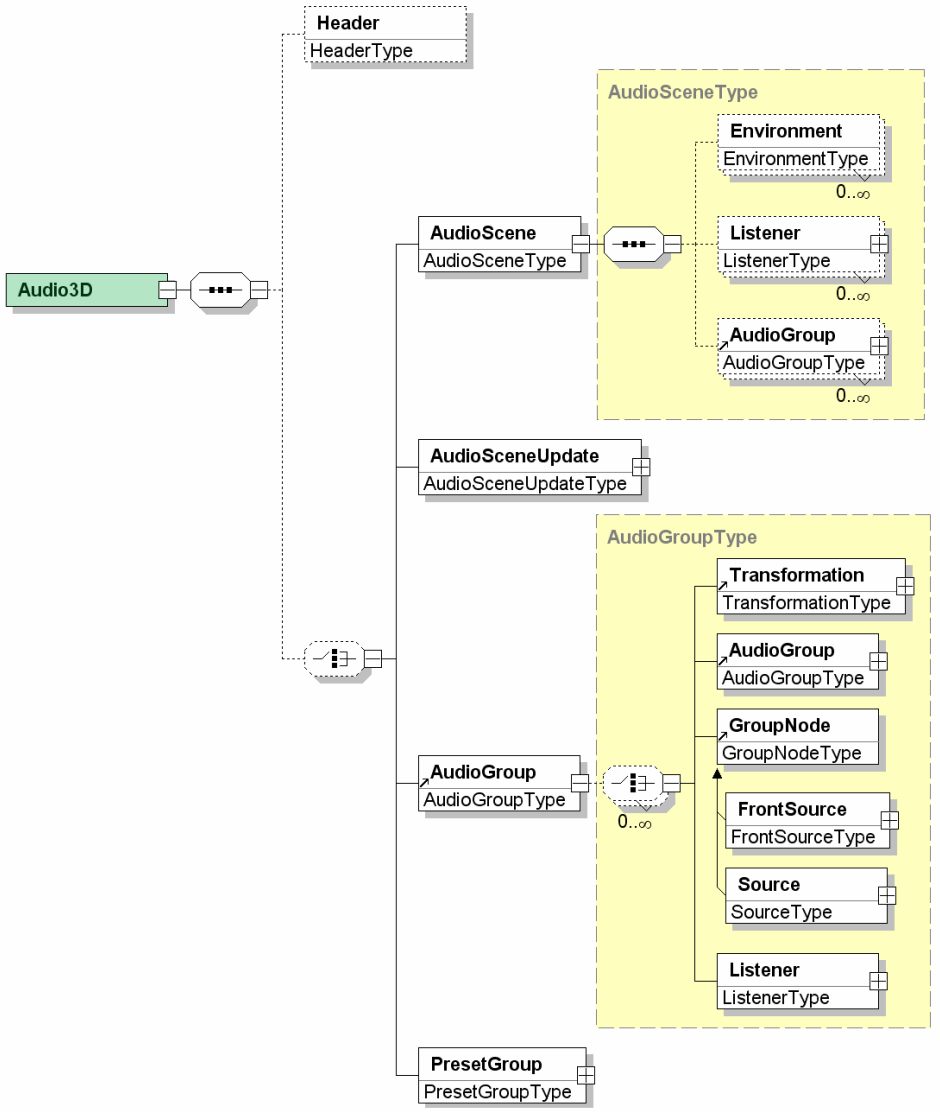

Abbildung 21: Grundlegende Struktur der *Audio3D* XML-Schemata

Nach den grundlegenden Metainformationen im *Header*-Element besteht die Auswahl zwischen verschiedenen Elementen. Typischerweise wird eine gesamte Szene mit ihren auditiven Eigenschaften samt Hörerposition im Element *AudioScene* beschrieben, wobei mit dem wichtigen Element *AudioGroup* ein auditiver Gruppenknoten zum hierarchischen Aufbau von Szenen zur Verfügung steht. Dieser Gruppenknoten kann auch separat und ohne die generellen Szeneninformationen eingesetzt werden, wenn die Audio3D-Datei später als Teil einer komplexen Szene verwendet werden soll. In jedem Fall ist es möglich, innerhalb einer *AudioGroup* mit dem Element *Transformation* – wie bei geometrischen Szenengraphen auch – eine Positionierung weiterer Subknoten vorzunehmen. Das können weitere Audiogruppen, Soundquellen oder Hörerpositionen sein.

AudioSceneUpdate erlaubt eine Form von Streaming-Funktionalität, da sich innerhalb des Elementes festlegen läßt, wie Teile der Szene zu späteren Zeitpunkten automatisch ausgetauscht, gelöscht oder auch neu hinzugefügt werden können. Neben dem ebenfalls realisierten DEF/USE-Konzept besteht mit den *PresetGroup*-Knoten eine weitere Möglichkeit der Wiederverwendung. Sie erlauben die Auslagerung beliebiger Audio3D-Subszenengraphen in externe Dokumente, die referenziert und so in unterschiedlichen Zusammenhängen verwendet werden können.

Das folgende XML-Codefragment zeigt ein *Audio3D*-Instanzdokument im Ausschnitt.

```
<Audio3D …>
    <Header title="…" author="…" />
    <AudioScene coordinateSystem="RFT">
        <Environment rollOffFactor="0.5" dopplerFactor="0.9“ distanceFactor="1.0“/>
        <Listener position="0 0 1.8" gain="0.7“ …/>
        <AudioGroup>
            <Transformation translation="5.0 0.0 10.5" rotation="0 0 1 1.57"/>
            <AudioGroup>
                <Transformation scale="0.8 0.8 0.8" …/>
                <Room priority="1">
                    <Reverb earlyLevel="-1000" earlyDelay="0.007" lateLevel="100" lateDelay="0.014“ …/>
                    <Walls transHF="800" occlusionReverbRatio="0.5“ …/>
                    <Transformation translation="10 0 0"/>
                    <Box size="10 20 3" DEF="OurBox"/>
                </Room>
                <Source position="14 2 10" startTime="0">
                    <Sound format="WAV" location="…"/>
                </Source>
            </AudioGroup>
            <Source …> … </Source>
        </AudioGroup>
    </AudioScene>
</Audio3D>
```

Mit dem Format wird eine klare Trennung der geometrischen Daten für die visuelle Darstellung von denen für das akustische Rendern vorgenommen. So lassen sich mit Audio3D vereinfachte Geometrien für Räume und Hindernisse beschreiben, wobei auf X3D-Geometriebeschreibungen zurückgegriffen wird. Diese reduzierte geometrische Komplexität macht ein akustisches Rendern teilweise überhaupt erst möglich. Einfache Geometrien, wie z.B. Quader oder polygonale Wände für Gebiete mit Halleigenschaften, genügen zumeist den Anforderungen an Soundberechnungen. Auf die speziellen Typen von Soundquellen und die Beschreibungsmöglichkeiten von Räumen mit ihren akustischen Eigenschaften kann hier jedoch nicht näher eingegangen werden.

5.4.4 Behavior3D – Beschreibung von Interaktionen und Objektverhalten

Die einfache und intuitive Beschreibung von Objektverhalten, Animationen, Interaktionsmöglichkeiten und Anwendungsfunktionalität spielt eine wesentliche Rolle für erfolgreiche interaktive 3D-Anwendungen. Insbesondere immer wiederkehrende Aufgaben sollten nicht jedesmal neu implementiert werden müssen. Zwar bieten Web3D-Formate – z.B. zur Objektanimation – bereits einige grundlegende Verhaltensbausteine, welche sich jedoch häufig nur schwer erweitern lassen und auch nur einen Teil der Anwendungsfälle abdecken. Die einzige Lösung sind Skripte bzw. andere Formen imperativer Verhaltensbeschreibung. Das Ziel des CONTIGRA-Ansatzes ist jedoch eine weitestgehend deklarative Realisierung auf allen Ebenen. Als Lösung wurde in [Rukzi02] die Verhaltensbeschreibungssprache *Behavior3D* entwickelt und in [Dachs03a] vorgestellt. Sie bildet die Basis für den CONTIGRA-Verhaltensgraph und kann aus Platzgründen hier nur kurz vorgestellt werden.

5.4.4.1 Charakterisierung und Abgrenzung zu X3D

Behavior3D wurde auf der Basis von VRML97/X3D entwickelt, erweitert und vereinheitlicht jedoch die dort existierenden Möglichkeiten zur Definition von funktionalem Verhalten. Dem X3D-Entwickler stehen drei unterschiedliche Möglichkeiten zur Verfügung, Verhalten zu realisieren: Knoten, Skripte und Prototypen. Die im X3D-Sprachstandard enthaltenen Knoten

für einfache Objektanimationen und Interaktionen, darunter *Interpolators, Sensors, Triggers* und *Sequencers* reichen für komplexere Animationen oder zustandsbasiertes Verhalten nicht aus. Skriptknoten bieten eine weitere – und oft mit standardisierten Knoten kombinierte – Möglichkeit zur Verhaltensimplementierung, wobei beliebige Funktionalität programmierbar ist. Probleme dabei sind jedoch unter anderem fehlende Typsicherheit und mangelnde Wiederverwendbarkeit. Schließlich können in VRML97/X3D komplexere Szenengraphen in Form von Prototypen gekapselt werden, die dann quasi als Komponente wiederverwendet werden können. Neben der – gegenüber Standardknoten unterschiedlichen – Syntax bei der Verwendung in Szenengraphen gibt es auch bei Prototypen keine Möglichkeit der Vererbung.

Um diese Nachteile auszugleichen und ein einheitliches, flexibles Konzept zur Definition von vielfältigen Verhaltensknoten, deren Implementierung und einfachen Integration in verschiedene Szenengraphen zu erzielen, wurde *Behavior3D* mit folgenden Kerneigenschaften entwickelt.

- Erweiterbarkeit und Flexibilität des Ansatzes, Unabhängigkeit von konkreten 3D-Formaten;
- Integration in bzw. Realisierung mit X3D;
- Objektorientierte Möglichkeiten bei der Entwicklung neuer Knoten: Vererbung, strenge Typisierung, Polymorphismus;
- Realisierung der einfachen Knoten-Definitionssprache *Behavior3DNode* mit XML-Schema;
- Vereinfachte Implementierung von Verhaltensknoten durch automatische Codegenerierung;
- Dynamische Generierung eines XML-Schemas auf der Grundlage aller existierenden Knotendefinitionen, wodurch Verhaltensknoten in anderen Szenengraphformaten gleichberechtigt verwendet werden können;
- Verfügbarkeit einer großen Anzahl vordefinierter, klassifizierter Verhaltensknoten zur deklarativen Verhaltensbeschreibung, gruppiert in Kollektionen.

5.4.4.2 Definition von Verhaltensknoten – Behavior3DNode

Die Definition neuer Verhaltensknoten erfolgt mit der in XML-Schema realisierten Grammatik *Behavior3DNode.* In der Abbildung 22 ist die Struktur dieser Auszeichnungssprache dargestellt.

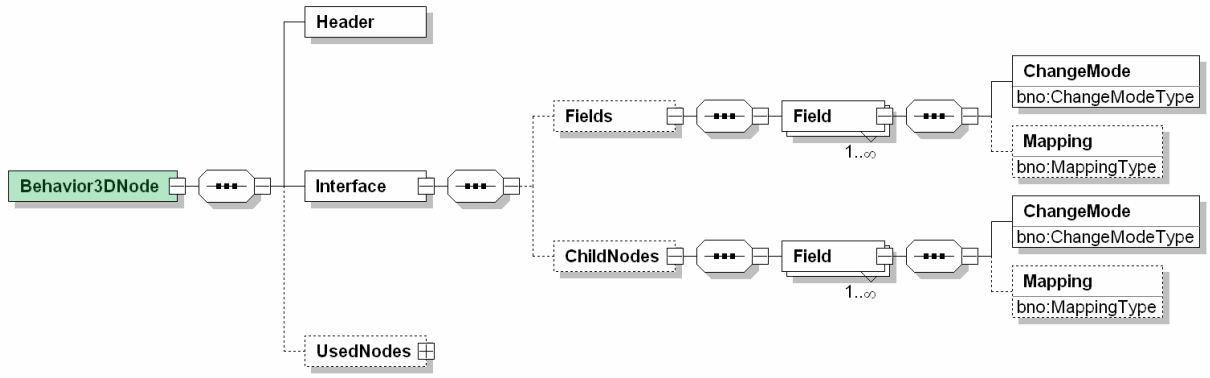

Abbildung 22: Grundlegende Struktur des XML-Schemas *Behavior3DNode*

Knoten können voneinander erben, was im *Header* beschrieben wird. Ebenso ist die Komposition aus anderen Knoten über das Element *UsedNodes* möglich. Verhaltensknoten besitzen ein oder mehrere Felder zur Konfiguration bzw. Repräsentation ihrer Funktionalität. Diese werden im *Interface* deklariert, wobei zwischen einfachen Feldtypen innerhalb von *Fields* und Knotenfeldtypen innerhalb von *ChildNodes* unterschieden wird. Das Element *ChangeMode* regelt den Zeitpunkt und die Art der Änderung von Feldern und wurde mit seinen drei Attributen konzeptionell bereits beim Parameterkonzept unter 5.3.3.4 beschrieben. Das folgende XML-Codebeispiel zeigt die Definition eines vom Knoten *Animation* erbenden Verhaltensknoten *AnimateRotation.*

```
<Behavior3DNode>
    <Header name="AnimateRotation"/>
    <Interface nodeType="public" extends="Animation">
        <Fields>
            <Field name="key" dataType="Floats" default="">
                <ChangeMode configurable="true" receivesEvents="true" generatesEvents="true"/>
            </Field>
            …
        </Fields>
    </Interface>
</Behavior3DNode>
```

Verhaltensknoten können – ähnlich wie funktionell zusammengehörige X3D-Szenengraphknoten in *X3D-Components* – in sogenannten *Collections* zusammengefaßt werden. Durch diese Klassifizierung wird dem Nutzer von Verhaltensknoten die Auswahl erleichtert. Alle existierenden und verhaltensrelevanten X3D-Knoten sind in die Systematik aufgenommen worden. Bisher vollständig definiert und implementiert wurden die Knoten-Kollektionen *StateMachine* und *Animation*.

5.4.4.3 Automatische Generierung der Verhaltensknoten-Sprache Behavior3D

Abbildung 23 zeigt die Entwicklungs- und Anwendungsebene für Behavior3D-Knoten sowie die zugehörigen XML-Schemata und Instanzdokumente. Die im vorangegangenen Unterkapitel beschriebene Knotendefinition und Einordnung von Knoten in Kollektionen ist im rechten unteren Teil der Abbildung zu sehen, links davon das bereits beschriebene XML-Schema *Behavior3DNode.*

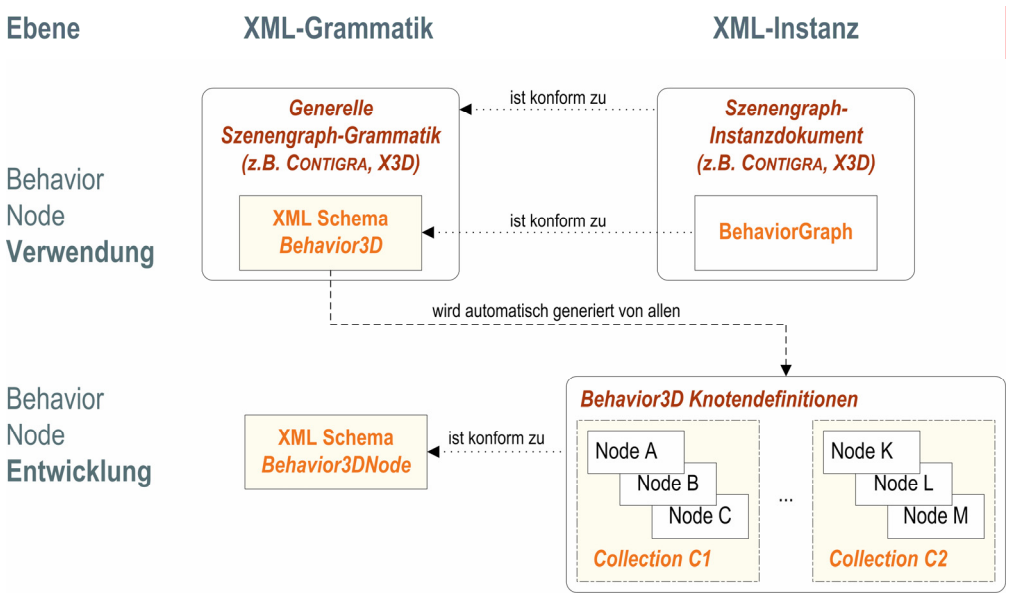

Abbildung 23: Überblicksdarstellung der Ebenen, Grammatiken und Instanzdokumente von *Behavior3D*

Ein wichtiges Ziel ist jedoch neben der Beschreibung von Verhaltensknoten ihre einfache Verwendung innerhalb von Szenengraphen, wie z.B. im CONTIGRA-Verhaltensgraph. Dazu wird in einem XSLT-Transformationsprozeß aus allen vorhandenen Verhaltensknotendefinitionen das XML-Schema *Behavior3D* generiert, das in andere Szenengraph-Grammatiken importiert werden kann. So wird aus der unter 5.4.4.2 gezeigten Definition des *AnimateRotation*-Knotens der folgende XML-Code in Behavior3D automatisch erzeugt:

```
<element name="AnimateRotation" type="AnimateRotationType“ substitutionGroup="Animation"/>
<complexType name="AnimateRotationType">
    <complexContent>
        <extension base="AnimationType">
            <attribute name="key" type="x3d:Floats"/>
            <attribute name="to" type="x3d:Rotations"/>
            <attribute name="by" type="x3d:Rotations"/>
        </extension>
    </complexContent>
</complexType>
```

Damit kann dieser Knoten in einem Instanzdokument einer Szenengraphgrammatik als Knoten erster Klasse und – anders als bei VRML/X3D-Prototypen – ohne weitere Hilfskonstrukte eingesetzt werden. Das folgende Codefragment zeigt die zweifache Verwendung des *AnimateRotation*-Kotens innerhalb eines Sequenzknotens:

```
<Sequential begin="5.0">
    <AnimateRotation key="0 1" to="1 0 0 0, 1 0 0 -1.5"/>
    <AnimateRotation key="0 1" to="1 0 0 -1.5 , 1 0 0 0"/>
</Sequential>
```

An diesem Beispiel läßt sich die einfache deklarative Verwendung funktionaler Knoten erkennen. Natürlich müssen diese zum Entwicklungszeitpunkt auch implementiert sowie der Programmcode in das Zielformat integriert bzw. zur Laufzeit aufgerufen werden. Die Implementation der Knoten bzw. die Transformation in andere Formate soll jedoch hier nicht näher erläutert werden, sie wird ausführlich in [Rukzi02] und [Dachs03a] beschrieben.

5.4.5 Das Verknüpfungskonzept für Komponenten- und Szenengraphen

Zum Verständnis der Verknüpfungstypen sollen zunächst zwei Begriffe eingeführt werden. In der CONTIGRA-Architektur umfaßt der Begriff *Vertex* entweder eine Komponente oder einen Szenengraphknoten. *Port* steht respektive für einen Parameter (einer Komponente) oder ein Feld (eines Knotens). Mit dieser Generalisierung wird vom Unterschied zwischen Komponenten- und Szenengraph abstrahiert.

Durch die Separierung der einzelnen Graphen (Komponenten, Geometrie, Audio, Verhalten) war es notwendig geworden, Möglichkeiten zur Verknüpfung der einzelnen Szenengraphbestandteile und Komponenten zu schaffen. Im Gegensatz zu den in VRML97/X3D verwendeten ROUTE-Statements, die ausschließlich einzelne Felder von Szenengraphknoten miteinander verbinden können, wurde jedoch ein erweitertes Verknüpfungskonzept entwickelt, mit dem auch Komponenten verknüpft werden können, n:m – Relationen möglich sind und Links eine eigene Semantik besitzen, die über eine reine Ereignisbehandlung hinausgeht. Auch Attributierungen von Verknüpfungen wurden erwogen, mit denen es z.B. möglich wäre, automatische Typkonvertierungen zwischen Ports vorzunehmen oder zeitliche Verzögerungen einzubauen. Diese zusätzlichen Attribute hätten die Linktypen zwar noch mächtiger gemacht, besitzen aber den Nachteil eines zusätzlichen Konzepts und einer Verkomplizierung des Formates. Sie sind außerdem nicht nötig, weil komplexere Funktionalität leicht durch zusätzliche Behavior3D-Knoten zwischen zwei Verknüpfungen eingefügt werden kann (s. [Rukzi02] für weitere Details).

Wie bereits weiter oben erwähnt wurde und in der Abbildung 20 ersichtlich ist, enthalten sowohl die Elemente *ComponentGraph* und *SceneGraph* einen Verknüpfungsteil *ComponentLinks* respektive *SceneGraphLinks*. Beide können eine Sequenz von mindestens einem bis beliebig vielen *Link*-Elementen enthalten. Fünf verschiedene Linktypen wurden spezifiziert, die eine Vereinigung der ROUTE- und DEF/USE-Konzepte von VRML97/X3D und eine Erweiterung um neue Konzepte realisieren. In der Abbildung 24 werden die in den folgenden Abschnitten näher beschriebenen Linktypen visualisiert.

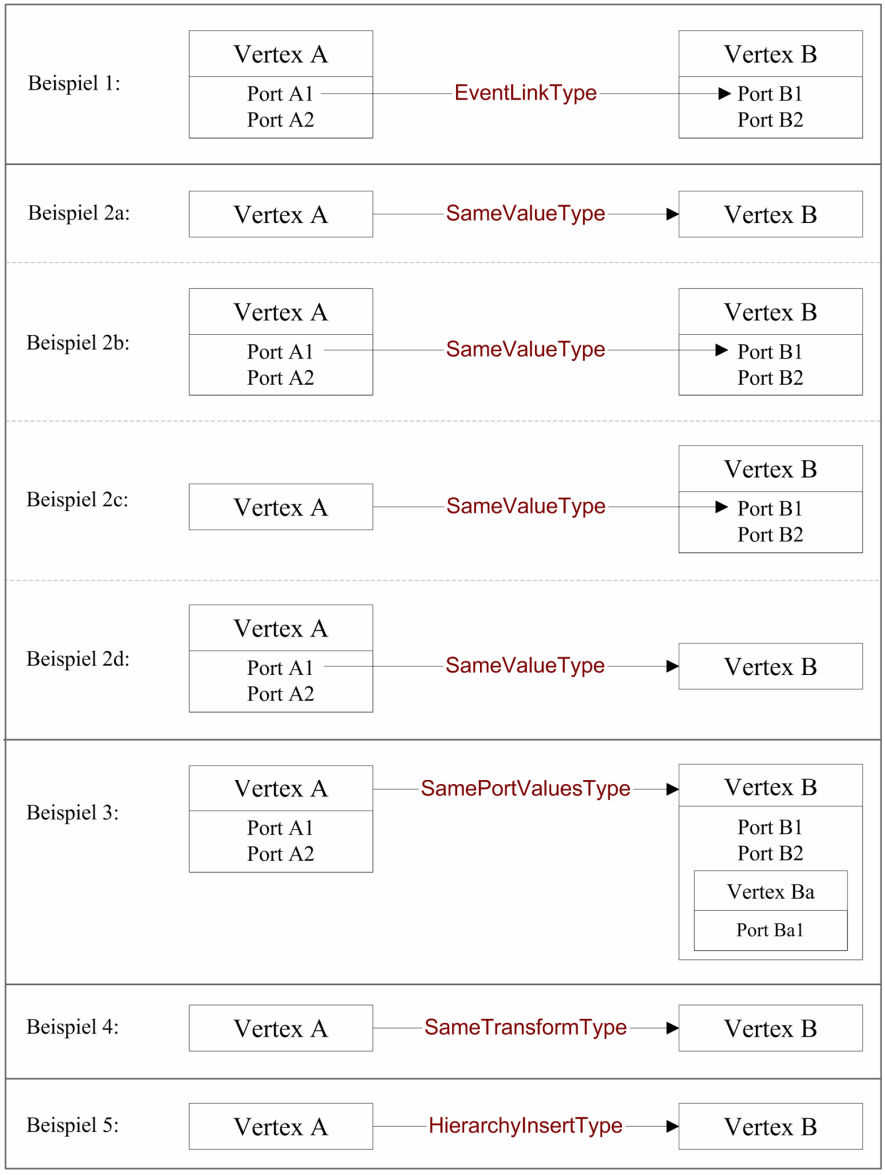

Abbildung 24: Beispiele für die verschiedenen CONTIGRA-Linktypen

5.4.5.1 EventLinkType

Der *EventLinkType* basiert auf dem ROUTE-Konzept von VRML97/X3D. Er ist der einzige ereignisbasierte CONTIGRA-Linktyp. Mit seiner Hilfe ist eine ereignisgesteuerte, unidirektionale Werteübertragung zwischen Quell- und Zielport möglich, die man auch als Ereignispfad bezeichnen kann. Quellports müssen dabei ein mit *true* gesetztes Attribut *generatesEvents* besitzen, Zielports dann *receivesEvents* respektive. Die Datentypen der Ports müssen zudem übereinstimmen. Das folgende XML-Codebeispiel aus dem Komponentengraph zeigt die Anwendung eines *EventLinkType* zur Verknüpfung eines Komponentenparameters mit einem Feld eines Behavior3D-Knotens.

```
<!—Ausschnitt der Komponentenverknüpfung der Contigra-Komponente ColorSlider -->
<ComponentLinks>
    <Link xsi:type="EventLinkType" configurationTime="false" runTime="true">
        <From vertex="SliderRed" port="CurrentValue"/>
        <To vertex="ColorSliderBehavior" port="Red"/>
    </Link>
    …
</ComponentLinks>
```

Auch alle anderen Linktypen haben eine ähnliche XML-Struktur. Für jeden Link kann außerdem durch die optionalen Boolean-Attribute *configurationTime* und *runTime* definiert werden, ob die Semantik der Verknüpfung während der Konfigurationszeit im Autorenwerkzeug bzw. während der Laufzeit etabliert wird.

5.4.5.2 SameValueType

Von seiner Funktionalität ist der Linktyp *SameValueType* dem DEF/USE-Konzept von VRML97/X3D entlehnt und findet im Beispiel 2a eine direkte Entsprechung. Dabei ist die Werteübertragung eines Quellknotens (links) zu einem Zielknoten (rechts) abgebildet, wobei dies in VRML97 dem Konstrukt *USE A* entsprechen würde, gleiche Knotentypen vorausgesetzt. Es wird der gesamte Quellvertex – im Falle eines Knotens also alle seine Felder und Kinderknoten – auf den Zielvertex abgebildet. Da beide Vertices den gleichen Typ besitzen müssen, sind Verknüpfungen zwischen Komponenten und Knoten nicht erlaubt. In Erweiterung zum bekannten USE-Konzept können neben Vertices auch Ports miteinander verknüpft werden, was im Beispiel 2b zu sehen ist. Durch die Verbindung zweier Ports wird sichergestellt, daß Zielport und Quellport den gleichen Wert besitzen. Für die Verknüpfung eines Komponentenparameters und eines Knotenfeldes müssen sich die entsprechenden Datentypen aufeinander abbilden lassen. Auf Knotenebene ist es auch möglich, daß ein Knoten Feldwert eines anderen Knotens sein kann. Die im Beispiel 2c und 2d gezeigten Fälle stellen solche Knoten-Feld-Verknüpfungen dar. Im Beispiel 2c wird das Feld B1, welches einen Knoten als Datentyp besitzt, mit dem Knoten A belegt, im Beispiel 2d jedoch der Knoten B mit dem Inhalt des Feldes A1, das ebenfalls einen Knotendatentyp aufweist.

5.4.5.3 SamePortValuesType

Dieser Linktyp ähnelt dem *SameValueType* im Beispiel 2a, wo eine Verknüpfung zwischen zwei Vertices hergestellt wird. Im Unterschied dazu werden aber keine Kinderknoten abgebildet, sondern ausschließlich die Port-Werte des Vertex A an die Port-Werte des Vertex B übertragen, wie im Beispiel 3 zu sehen ist. Das setzt voraus, daß beide Vertices dieselbe Zahl und denselben Typ von Ports aufweisen. Ihre Kinder-Vertices – so vorhanden – können jedoch unterschiedlich sein und werden nicht berücksichtigt. Der *SamePortValuesType* stellt eine syntaktische Vereinfachung dar, da man ihn alternativ auch mit n *SameValueType*-Verknüpfungen für die n Ports eines Vertex nachbilden könnte.

5.4.5.4 SameTransformType

Folgender Hintergrund war für die Entwicklung dieses Linktyps ausschlaggebend. Die Trennung von Geometrie- und Audiograph hat zur Folge, daß der Audiograph zwar Bestandteile mit konkreter Raumposition, z.B. eine Soundquelle, besitzt, jedoch eine andere Transformationshierarchie aufweisen kann als das geometrische Pendant, z.B. ein Radio auf einem Tisch. Als Lösung wurde der *SameTransformType* eingeführt. Dabei wird die aktuelle globale Transformation des Quellvertex auf den Zielvertex übertragen und somit die Positionen im Raum synchronisiert. Dies kann zur Autorenzeit z.B. die Positionierung einer Soundquelle und zur Laufzeit ihre automatische Nachführung bei Änderung der Position einer korrespondierenden Geometrie erleichtern.

5.4.5.5 HierarchyInsertType

Da in CONTIGRA die Szenengraphen Geometrie, Verhalten und Audio unterschieden werden, wurde ein Konstrukt zur Zusammenführung von Szenengraphbestandteilen entwickelt. Der im Beispiel 5 gezeigte *HierarchyInsertType* erlaubt die Einordnung eines Quellvertex mit der möglicherweise existierenden gesamten Subhierarchie unterhalb des angegebenen Zielvertex. Die Einordnung einer Komponente unter eine andere Komponente ist bisher nicht möglich.

5.4.5.6 Verknüpfungskardinalitäten

Ein Kriterium bei der Entwicklung des CONTIGRA-Verknüpfungskonzeptes war neben semantisch aussagekräftigen Links auch die Reduzierung nötiger Verbindungen in einer Implementierung. Im Falle von VRML97-ROUTE-Statements handelt es sich um unidirektionale, 1:1 – Verknüpfungen zwischen Feldern. Bei komplexeren interaktiven 3D-Szenen wächst die Zahl nötiger Verknüpfungsanweisungen sehr schnell und führt häufig zu Unübersichtlichkeit im Dokument. Die erwähnten CONTIGRA-Linktypen erlauben hingegen die Definition von m:n – Verknüpfungen, wobei $m,n \geq 1$ ist. Im folgenden XML-Fragment wird deutlich, wie z.B. eine 1:n – Abbildung in kompakter und leicht wartbarer Form notiert werden kann.

```
<!--Deklaration einer 1:n - Verknüpfung -->
<Link xsi:type="SameValueType">
    <From vertex="Globales_Material" port="diffuseColor"/>
    <To vertex="Material_Kugel" port="diffuseColor"/>
    <To vertex="Material_Quader" port="diffuseColor"/>
    <To vertex="Material_Kegel" port="diffuseColor"/>
</Link>
```

5.4.6 Zuordnung von Schnittstellenparametern zu Implementierungsteilen

Wie in der Abbildung 20 ersichtlich ist, wird nach dem Komponenten- und Szenengraph der dritte Teil eines *CoComponentImplementation*-Instanzdokuments durch das Element *InterfaceParameterLinks* gebildet. Dieses dient der Abbildung der in der Komponenten-Schnittstelle beschriebenen High-Level-Parameter auf konkrete Bereiche der Implementierung, also Szenengraph-Bestandteile oder Parameter anderer Subkomponenten. Für jeden in einem *CoComponent*-Instanzdokument deklarierten Parameter wird deshalb am Ende eines Implementierungsdokuments ein Element *Link* angegeben, mit dessen Attribut *fromParameter* die Verbindung zum Parameter der Schnittstelle hergestellt wird. Das folgende XML-Codefragment zeigt zwei der Link-Definitionen für die Komponente Ringmenü.

```
<!-- Verknüpfung von Schnittstellenparametern des Ringmenüs mit seinen Implementierungsteilen -->
<InterfaceParameterLinks>
    <Link fromParameter="RingRadius">
        <To vertex="RingMenuBehavior" port="RingRadius"/>
    </Link>
    <Link fromParameter="FixedGeometry">
        <To vertex="FixedGeometryTransform" port="children"/>
    </Link> …
</InterfaceParameterLinks>
```

Im Unterelement *To* wird festgelegt, zu welchem Vertex und Port der Implementierung die Abbildung erfolgt. Das Attribut *vertex* enthält demzufolge eine Zeichenfolge, die eine benannte Subkomponente oder einen benannten Knoten innerhalb des CONTIGRA-Implementierungsdokumentes referenziert. Mit *port* wird ein Parameter einer benannten Komponente oder ein Feld eines benannten Knotens referenziert. Dabei bezieht sich das Attribut immer auf die Komponenten bzw. Knoten, die im Attribut *vertexRef* referenziert werden. Das Element *To* kann auch mehrfach auftauchen, wenn beispielsweise ein High-Level-Parameter Farbe auf die *diffuseColor*-Felder verschiedener Materialknoten abgebildet wird.