4 CONTIGRA: Konzeption einer 3D-Komponentenarchitektur

Im letzten Kapitel ist deutlich geworden, wie wichtig eine exakte Spezifikation, flexible Anpassung und einfache Kombination bzw. Verknüpfung von 3D-Widgets und anderen 3D-Bausteinen für eine erfolgreiche Entwicklung von 3D-Applikationen ist. Wiederverwendbarkeit und einfache 3D-Anwendungserstellung sind wesentliche Kriterien für die weite Verbreitung von interaktiven dreidimensionalen Applikationen und künftige Standardisierungen auf diesem Gebiet. Deshalb werden in diesem Kapitel zunächst Anforderungen an eine dafür geeignete Architektur, ihre modularen Bausteine und den Autorenprozeß gestellt.

Aus der Anforderungsanalyse ergibt sich die Schlußfolgerung, daß etablierte Komponententechnologien aus der Softwaretechnik einen möglichen Lösungsansatz darstellen. Verbreitete Software-Komponentenmodelle wurden daher auf ihre Eignung für die Entwicklung interaktiver 3D-Anwendungen untersucht. Als Fazit der Analyse ergeben sich mehrere Defizite dieser Technologien in Bezug auf die gestellten Anforderungen und Ziele. Ein wesentlicher Nachteil ist die fehlende Unterstützung von 3D-Grafik, weshalb mehrere Forschungsansätze versuchen, den Komponentenansatz mit verschiedenen Grafik-Technologien in Form spezieller 3D-Komponentenarchitekturen zu verknüpfen. Diese verwandten Arbeiten werden im Detail vorgestellt und nach verschiedenen Kriterien klassifiziert.

Als Resultat der Analyse entstand die Idee, ein eigenes, konsequent deklaratives Komponentenmodell für 3D-Grafik zu entwickeln, welches die wichtigsten ermittelten Defizite beseitigt. CONTIGRA – *Component-OrieNted Three-dimensional Interactive GRaphical Applications* – ist ein dokumentzentrierter Lösungsansatz, der am Ende dieses Kapitel im Überblick vorgestellt werden soll. Dazu wird zunächst der eigene Komponentenbegriff eingeführt und der deklarative Ansatz begründet. Es folgt die Darstellung der einzelnen Entwicklungsebenen für Komponenten innerhalb der CONTIGRA-Komponentenarchitektur. Schließlich werden in einer Übersicht die entwickelten deklarativen Beschreibungssprachen auf XML-Basis vorgestellt, auf denen der gesamte Ansatz und das Autorenwerkzeug basieren.

4.1 Motivation und Anforderungsanalyse

Im Kapitel 3 wurden interaktive 3D-Applikationen mit ihren Bestandteilen und insbesondere ihrer 3D-Benutzungsoberfläche charakterisiert. Dazu gehören 3D-Widgets als wesentliche Interaktionselemente. Ihre exakte Spezifikation, die flexible Anpaßbarkeit für verschiedene Anwendungen und ihre einfache Kombination und Verknüpfung sind Kriterien für die erfolgreiche Entwicklung von 3D-Applikationen. Diese sollten sich künftig an Standards orientieren. Der Vergleich von 2D- und 3D-Benutzungsoberflächen im Unterkapitel 3.1.5 hat verdeutlicht, wie wichtig weitere Studien und Arbeiten für eine bessere Bewertung der Eignung und Benutzbarkeit von 3D-GUIs sind.

Deutlich wurde im Kapitel 3 auch – z.B. bei den zahlreichen Literaturverweisen in 3.1.6 – die Notwendigkeit, das immer noch geringe und weit gestreute Repertoire von 3D-Widgets und -Interaktionstechniken zu systematisieren und zu erweitern. Damit stellt sich einerseits die Frage, wie sich Widgets einheitlich spezifizieren und implementieren lassen und welche technologische Basis dafür gewählt werden sollte. Andererseits spielt auch die möglichst einfache Konstruktion von komplexeren 3D-Anwendungen aus solchen Interaktionsbausteinen eine zentrale Rolle, wobei in [Hand97] beschrieben wird, wie wenig Unterstützung dem VE-Anwendungsentwickler bisher in Form von Werkzeugen und Toolkits zur Verfügung steht. Welche Autorenwerkzeuge können also zur effektiven und interdisziplinären Erstellung von 3D-Applikationen eingesetzt werden, und gibt es diese bereits?

In [Zelez00] wird zudem bemerkt, daß bisher kaum Softwarekomponenten für Interaktionstechniken und Objektverhalten in VE existieren oder über Systemgrenzen hinweg verwendet werden. Wiederum wird in [Spalt02] darauf hingewiesen, daß gerade die 3D-Anwendungsdomäne sehr von Komponentenbausteinen profitieren würde. Im folgenden werden aus dieser Motivation zur Entwicklung eines modularen Ansatzes für die einfache Konstruktion dreidimensionaler Benutzerschnittstellen heraus Anforderungen aufgestellt, die eine High-Level-Architektur idealerweise erfüllen sollte. Die wiederverwendbaren Bestandteile dreidimensionaler Applikationen werden in dieser Anforderungsanalyse allgemein als Komponenten bezeichnet. Ein Teil der folgenden Anforderungen wurde bereits in [Dachs01a] beschrieben.

Generelle Anforderungen

- *Interne Szenengraph-Basis:* Nutzung des bewährten Szenengraphkonzepts für interaktive 3D-Grafik; Einsatz eines weit verbreiteten oder standardisierten 3D-Formats bzw. -Toolkits als Realisierungsbasis.
- *High-Level-Sicht:* Schaffung einer Abstraktionsebene oberhalb des Szenengraph-Niveaus, um eine Erstellung von 3D-Applikationen aus komplexeren semantischen Einheiten zu ermöglichen, statt computergrafischen, technischen Details Rechnung tragen zu müssen.
- *Anwendungsvielfalt:* Erstellung von 3D-Anwendungen für möglichst verschiedene Anwendungsfälle; demzufolge nicht nur Fokus auf 3D-Widgets und Benutzeroberflächen, sondern ebenso Unterstützung von 3D-Animationen, 3D-Objekten ohne Interaktion etc.
- *Erweiterbarkeit und Flexibilität:* durch modulare Basis und offene Schnittstellen.
- *Medienintegration:* Erweiterung des zugrundeliegenden Szenengraphformates bzw. Toolkits bezüglich der Integration verschiedener Medien, z.B. für Raumklang.
- *Unabhängigkeit:* Plattformunabhängigkeit, Vermeidung von Abhängigkeiten zu proprietären Technologien, konkreten Programmiersprachen, Betriebssystemen, Zielplattformen, Webbrowsern etc.
- *Standards:* Weitgehende Integration / Interoperabilität zu Medien- und Internetstandards.

- *Portabilität und Internetfähigkeit:* der erzeugten Anwendungen, die sowohl *stand-alone* als auch Web-basiert funktionieren sollten.
- *Integrierbarkeit:* Flexible Integration dreidimensionaler Bestandteile in beliebige Anwendungen, z.B. Webseiten, und geeignete Kommunikation zwischen ihnen.
- *Adaption:* Einfache Transformation in verschiedene 3D-Formate, ggf. Anpassung an Netzwerkverbindung, Zielplattform (Skalierung je nach Grafikfähigkeiten) und Nutzerpräferenzen (z.B. Darstellungsqualität, Sprache, Vorlieben).

Anforderungen an wiederverwendbare Bausteine

- *High-Level-Parameter:* zur Anpassung der semantischen Einheiten statt Änderung von Szenengraphdetails (Knoten, Felder).
- *Verbergen der Implementierung:* Trennung v. Komponentenschnittstelle u. Realisierung.
- *Konfigurierbarkeit und Anpassung:* Parametrisierung jeder Komponente bezüglich ihrer Funktion und optischen Erscheinung; Anpaßbarkeit in vorgegebenen Grenzen.
- *Deskriptive Komponentenschnittstelle:* Strukturierte, textuelle Repräsentation (z.B. durch Schnittstellenbeschreibungssprache) der angebotenen Funktionalität, der konfigurierbaren Parameter, Abhängigkeiten, Komponentenbeschreibung etc., dazu Metainformationen zur Distribution, Suche und Verwendung, z.B. Autor, Firma, Version, Lizenzinformationen.
- *Trennung von Erstellung und Verwendung:* Unterstützung der verschiedenen Teile des Lebenszyklus von Komponenten.
- *Komposition und Gruppierung:* Komponenten (auch anderer Hersteller) müssen sich leicht miteinander kombinieren und verknüpfen lassen. Komponentenaggregate sollen sich wieder als Komponenten kapseln lassen, wobei Sichtbarkeitsfragen an der neuen Komponentenschnittstelle geklärt werden müssen.
- *Editierbarkeit:* Leichte Erstellung und Anpassung von Komponenten auch ohne Programmierkenntnisse oder spezialisierte Autorenwerkzeuge.

Anforderungen an den Autorenprozeß und seine Werkzeuge

- *Werkzeugunterstützung:* für alle Phasen, also Erstellung, Distribution, Auswahl, Anpassung und auch Verwendung von Komponenten bei der 3D-Anwendungsentwicklung.
- *Visuelle Werkzeuge:* Autorenwerkzeuge sollten einfach und mit möglichst wenig Programmierkenntnissen zu benutzen, aber dabei nicht restriktiv sein. Das heißt, neben einer einfachen, visuellen High-Level-Anwendungsentwicklung sollte jederzeit die genaue Kontrolle von Details möglich sein.
- *Interdisziplinäre Anwendungsentwicklung:* Unterstützung unterschiedlicher Aufgaben und Sichtweisen von Nutzern aus verschiedenen Fachgebieten, z.B. durch andere mentale Modelle, angepaßte Editoren, Expertenmodi etc.
- *Austauschbarkeit und alternative Werkzeuge:* Verschiedene Werkzeuge sollten über ein nicht-proprietäres, menschenlesbares Austauschformat alle wichtigen Daten austauschen können. Alternative Editoren sollten jederzeit einsetzbar sein, z.B. neben ausgereiften visuellen Werkzeugen auch Texteditoren.
- *Werkzeugintegration:* Bereits existierende Standardwerkzeuge, z.B. zur Medienbearbeitung oder zum 3D-Modeling, sollten in den Autorenprozeß integriert werden können.
- *Rapid Prototyping:* Unterstützung von evolutionärem Rapid-Prototyping für einen iterativen Autorenprozeß, Vermeidung typischer *edit-compile-run* Zyklen.
- *Erweiterbarkeit:* Werkzeuge sollten eine offene Schnittstelle und erweiterbare Architektur besitzen, um sie auch für andere Entwickler zugänglich zu machen.

4.2 Software-Komponentenmodelle und ihre Eignung für 3D-Grafik

Aus den zuvor dargestellten Anforderungen wird deutlich, daß die Idee der komponentenbasierten Softwareentwicklung eine mögliche Lösung auch für die Implementierung von 3D-Grafikanwendungen darstellt. Besonders die unter 4.1 formulierten Anforderungen an wiederverwendbare 3D-Bausteine ähneln den typischen Eigenschaften einer Softwarekomponente, wie sie z.B. in [Szype98] formuliert sind. Deshalb wurden zunächst bekannte Komponententechnologien auf ihre Eignung für die Erstellung von 3D-Applikationen untersucht. Alle etablierten Komponententechnologien folgen den bekannten Definitionen für Softwarekomponenten von Szyperski ([Szype98] S. 3 u. 34):

> *Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system.*
>
> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

In [Braig00a] wurden aktuelle Komponententechnologien anhand verschiedener Bewertungskriterien verglichen. Dazu zählen Plattformunabhängigkeit, Sprachunabhängigkeit, Kommunikationsmechanismus, verteilte Programmierung, Objekt-Dienste und Schnittstellenbeschreibung. Drei hauptsächliche Entwicklungslinien von Komponentenarchitekturen lassen sich gegenwärtig ausmachen, die wesentlich an Hersteller gebunden sind. Das ist erstens die *Common Object Request Broker Architecture (CORBA)* [CORBA@] der *Object Management Group (OMG)*. Zweitens ist Microsofts *Component Object Model (COM)* [COM@] zu nennen, worauf u.a. das verteilte Komponentenmodell *Distributed Component Object Model (DCOM)* und die darüber kommunizierenden *ActiveX-Controls* aufbauen. Zur dritten Gruppe zählt die SUN-Entwicklungslinie auf Java-Basis, zu der *JavaBeans* [JavaBeans@] für kleinere Anwendungen mit Benutzungsschnittstellen und *Enterprise JavaBeans* [EJB@] für verteilte Geschäftsanwendungen zählen. Diese Technologien sollen hier nur in sehr kurzer Form mit einigen wichtigen Eigenschaften dargestellt werden. Für einen genaueren Überblick wird auf die Originalquellen bzw. die zahlreiche Sekundärliteratur, wie z.B. [Szype98], verwiesen.

4.2.1 Common Object Request Broker Architecture (CORBA)

Die *Object Management Group* veröffentlichte 1991 CORBA (inzwischen Version 3.0.2) [CORBA@] mit dem Ziel der Interoperabilität verteilter Applikationen durch eine Referenzarchitektur für Middleware-Technologie, die verschiedene Sprachen, Implementierungen und Plattformen in einer heterogenen Umgebung verbindet. Kern der CORBA-Architektur ist der *Object Request Broker (ORB)*. Er garantiert Portabilität und Interoperabilität von Objekten über ein Netzwerk heterogener Systeme durch einen entfernten Methodenaufrufdienst. Basis für diese Funktionalität sind eine gemeinsame Schnittstellen-Beschreibungssprache und Sprachbindungen zu gebräuchlichen Programmiersprachen. Zur Schnittstellendefinition wurde die *OMG Interface Definition Language (OMG IDL)* entwickelt. Für die Kommunikation zwischen verschiedenen Systemen ist als Interoperabilitätsprotokoll das seit 1995 standardisierte *Internet Inter-ORB Protocol (IIOP)* geschaffen worden. In CORBA sind außerdem zahlreiche Dienste spezifiziert, die eine komponentenorientierte Entwicklung speziell von Geschäftsanwendungen unterstützen. Als Verbindungsstandard ist CORBA sehr erfolgreich. Mehrere Implementierungen, verschiedene Erweiterungen und Schnittstellen zu anderen Objekttechnologien und ein eigenes Komponentenkonzept *(CORBA Components)* existieren.

CORBA-Komponenten sind eine Erweiterung und Spezialisierung des CORBA *Object* Metatyps und stellen für die Interaktion mit anderen CORBA-Elementen die vier Schnittstellen

bzw. Ports *Facets, Receptacles, Event Sinks & Sources* und *Attributes* bereit. Zur Schnittstellendefinition wird ebenfalls IDL verwendet, woraus u.a. *Server Skeletons* und *Client Stubs* generiert werden können. Die *Component Implementation Definition Language (CIDL)* wiederum ist eine deklarative Sprache, die für die automatische Erzeugung von *Component Descriptors* und *Component Implementation Skeletons* verwendet wird. Die gesamte Komponentenentwicklung ist über ein *Component Implementation Framework* standardisiert. CORBA und auch CORBA-Komponenten sind aufgrund des großen Kommunikationsoverheads und der Orientierung auf komplexe, verteilte Client-Server-Anwendungen für interaktive 3D-Applikationen kaum geeignet.

4.2.2 Component Object Model (COM)

Das Microsoft *Component Object Model (COM)* [COM@] ist die Basis aller Komponentenansätze der Windows-Plattform und legt die Kommunikation binärer Objekte unabhängig von deren Sprache oder Implementierung über definierte Schnittstellen fest. Der Aufbau einer COM-Klasse ist vollständig gekapselt, der Zugriff auf ein Objekt erfolgt nur über die Schnittstellenmethoden. Mit *COM+* wird ein flexibles, standardisiertes Objektmodell und eine Infrastruktur für Drittanbieter-Komponenten zur Verfügung gestellt. Damit ist eine sichere und effizientere Intraprozeßkommunikation zwischen Komponenten möglich, jedoch nicht über Maschinengrenzen hinweg. Das wiederum leistet das *Distributed Component Object Model (DCOM)* als transparente Erweiterung der COM-Konzepte um ein Protokoll zur verläßlichen, sicheren und effizienten Komponentenkommunikation durch entfernten Methodenaufruf über ein Netzwerk. Dazu werden heterogene Daten in die plattformunabhängige *Network Data Representation (NDR)* umgewandelt. DCOM arbeitet über COM sowohl mit Java Applets als auch mit ActiveX-Komponenten zusammen. Zur Beschreibung von Schnittstellen dient die *COM Interface Definition Language (COM IDL)*.

Auf Basis von COM entstand z.B. die low-level 3D-Grafikbibliothek *Direct3D*, die von dem effizienten Methodenaufruf zwischen COM-Objekten profitiert, jedoch plattformabhängig ist. DCOM ist – ähnlich wie CORBA – aufgrund der Auslegung in Richtung verteilter Client-Server-Anwendungen für 3D-Grafikanwendungen wenig geeignet. Eher in Frage kämen dafür *ActiveX-Controls* als dritte Generation von *OLE (Object Linking and Embedding) – Controls.* Sie sind ebenfalls zur Distribution von Komponenten über Netzwerke konzipiert und erlauben die Integration in Webbrowser. Allerdings beschreiben ActiveX-Controls sich selbst über ein Binärfile, welches die Properties, Methoden und Events auflistet. Das widerspricht dem Kriterium eines menschenlesbaren Austauschformates. Alle COM-basierten Technologien werden als plattformabhängige Technologien der Forderung nach Portabilität und Plattformunabhängigkeit nicht gerecht.

4.2.3 JavaBeans und Enterprise JavaBeans

JavaBeans [JavaBeans@] wurden 1996 von SUN als Komponentenmodell für die Sprache Java veröffentlicht. Die in der Version 1.01 vorliegende API ist vorrangig für kleine bis mittelgroße Client-Applikationen mit Bedien- und Steuerelementen gedacht. Ein Bean ist eine Menge von Klassen und Ressourcen. Der Standard definiert weniger technische Details, sondern primär Regeln für die Namen von Schnittstellen, einen Kommunikationsmechanismus (ereignisbasiert) und ein Archivdateiformat zur Distribution von Komponenten (Persistenz). Die Anpassung *(Customization)* von Beans erfolgt über *Properties* als benannte Attribute von Komponenten, wobei der Zugriff durch Methoden gekapselt wird. Die Funktionalität einer Klasse läßt sich durch das Konzept der *Introspektion* ermitteln. Die Bean-Funktionalität kann unterschieden werden in Code für die Design- wie auch die Laufzeit. Durch die enge Anlehnung an die Programmiersprache Java und durch existierende visuelle Werkzeuge lassen sich schnell und elegant komponentenbasierte Anwendungen mit Nutzerschnittstellen entwickeln.

Die Sprachabhängigkeit ist jedoch auch zugleich der größte Nachteil dieser Technologie. Dazu kommt durch die interpretierte Ausführung noch der prinzipielle Performance-Nachteil gegenüber anderen Technologien. Trotzdem werden JavaBeans den Anforderungen interaktiver 3D-Grafik gerecht [Braig00a], was auch in der Kombination mit Java3D bei den *Three-dimensional Beans* [Dörne00] deutlich wird (s. 4.3.3). Damit wird jedoch die Forderung nach Unabhängigkeit von einem konkreten 3D-Format verletzt.

Enterprise JavaBeans (EJB) [EJB@] sind serverseitige, komplett in Java programmierte Komponenten für die Java 2 Plattform, Enterprise Edition (J2EE). Das EJB-Komponentenmodell erlaubt die schnelle und konsistente Entwicklung verteilter, transaktionsorientierter, sicherer und portabler Java-Applikationen mit Datenbankanbindung. Es handelt sich bei EJB um eine plattformunabhängige Komponentenarchitektur für verteilte Geschäftsanwendungen und zugleich um ein Framework für die Entwicklung von Middleware-Applikationen. Auch die Unterstützung von *Web Services* ist seit der EJB Spezifikation 2.1 möglich. Zur API und Gesamtarchitektur gehören EJB Servers, EJB Containers, EJB Clients, unterstützende Dienste, wie das *Java Name and Directory Interface* oder der *Java Transaction Service* und zahlreiche andere Module. Die Kommunikation zwischen einem JavaBeans-Client bzw. einer HTML-Seite und der EJB Komponente erfolgt z.B. über das *IIOP* oder Java-Servlets. *Deployment-Descriptors* erlauben die Anpassung und Konfiguration von EJB-Komponenten. Da es sich bei EJB ausschließlich um *nicht-visuelle* Komponenten handelt, die auf einem EJB Server innerhalb eines EJB Containers existieren, sind sie kaum für interaktive Benutzungsoberflächen auf der Clientseite geeignet. Neben der Sprachabhängigkeit als Nachteil erweist sich die EJB-Spezifikation – wie bei DCOM und CORBA – mit ihrer komplexen Funktionalität zur Entwicklung verteilter Geschäftsanwendungen als wenig geeignet für interaktive 3D-Grafikanwendungen.

4.2.4 Fazit der Analyse

An dieser Stelle soll noch einmal zusammengefaßt werden, warum die existierenden Technologien für Software-Komponenten nur wenig oder gar nicht für die modulare Entwicklung von 3D-Grafikapplikationen – wie in den Anforderungen unter 4.1 beschrieben – geeignet sind. Auch in anderen Arbeiten, z.B. [Capps00], wird festgestellt, daß Komponententechnologien bisher nur selten in VR-Systemen eingesetzt werden. Dörner und Grimm bemerken dazu in [Dörne00], daß Komponentenarchitekturen sich nicht ohne weiteres für die Erstellung von 3D-Komponenten erweitern lassen.

- Die Forderungen nach *Editierbarkeit, Interdisziplinärer Anwendungsentwicklung, Austauschbarkeit und alternativen Werkzeugen* legen nahe, daß kein imperatives, sondern ein deklaratives Entwicklungsmodell gewählt werden sollte. Alle erwähnten Komponententechnologien sind jedoch programmzentriert, sieht man von den deklarativen Schnittstellenbeschreibungen für Komponenten ab.
- Die Fokussierung auf eine bestimmte Komponententechnologie würde die Forderung nach *Unabhängigkeit* von konkreten Betriebssystemen, Programmiersprachen und Technologien nicht erfüllen. Lediglich CORBA bietet sowohl Sprach- als auch Plattformunabhängigkeit. Auch der Einsatz von offengelegten *Standards* ist nicht bei jeder Komponententechnologie gewährleistet.
- Forderung nach einer *internen Szenengraph-Basis* bedingt einfache Integration einer 3D-Technologie in die Komponentenarchitektur. Dies ist ohne größeren Aufwand lediglich bei Java Beans und Java3D als Grafikformat gegeben. Ähnliches gilt für die Anforderung der *Medienintegration,* die insbesondere von den verteilten Komponentenarchitekturen nur ungenügend unterstützt wird, weil diese nicht auf multimediale Anwendungsentwicklung zugeschnitten sind.

- Ein Großteil der Funktionalität von Komponententechnologien für transaktionsorientierte Anwendungen (CORBA, DCOM, EJB) ist für die Entwicklung von einfachen dreidimensionalen *stand-alone* oder Web-Anwendungen überflüssig, weil es sich bei solchen Anwendungen in den seltensten Fällen um verteilte, aufwendige Client-Server-Systeme mit Datenbankzugriff handelt. Zu den nicht benötigten Funktionen zählen z.B. Kommunikationsoverhead, Mechanismen zur Transaktionskontrolle und Sperrung, zur Persistenz, Selbstinstallation, Selbsttest, Sicherheit, späten Bindung über Skriptsprachen u.a.

4.3 Verwandte Arbeiten – eine Klassifikation

Es ist im vorigen Unterkapitel deutlich geworden, daß traditionelle Software-Komponententechnologien nur bedingt oder gar nicht als Realisierungsmodell für 3D-Grafikapplikationen geeignet sind. Hier sollen nun beispielhaft reine 3D-Architekturen vorgestellt werden.

Zunächst kamen dafür all jene 3D-Toolkits und Architekturen in Frage, die speziell für die Entwicklung von 3D-Widgets konzipiert wurden. Dazu zählen eine um 3D-Widgets erweiterte OSF-Motif-Bibliothek der Uni Karlsruhe [Bröck92], das *UGA Widget Toolkit* der Brown University von Stevens et al. [Steve94], der *Virtuality Builder II* von Balaguer und Gobbetti [Balag93], die Architektur für interaktive, animierte 3D-Widgets von Döllner und Hinrichs [Dölln98], das C-Lab-Werkzeug *WidgetEdit* von Geiger et al. [Geige98], die *SVIFT Interactors* von Kessler [Kessl99] und die *Interactive Toolkit Library for 3D Applications* von Osawa et al. [Osawa02]. Alle diese Architekturen stellen – teils visuelle – Möglichkeiten zur Verfügung, 3D-Widgets schnell zu entwerfen und zusammenzufügen, wobei teilweise *Constraints* zum Einsatz kommen. Häufig werden Widgets aus geometrischen Primitiven erstellt (z.B. beim *UGA Widget Toolkit*), was der Forderung nach einem High-Level-Ansatz widerspricht. Die meisten Tools bieten jedoch Kompositionsprinzipien für hierarchisch komplexere Objekte an. Alle Systeme sind von konkreten Programmiersprachen und viele von 3D-Formaten abhängig (z.B. *WidgetEdit* von Open Inventor) bzw. besitzen als reine Programmbibliotheken gar keine Austauschformate oder visuelle Editoren, womit sie als Realisierungsbasis ausscheiden. Auch handelt es sich bei den meisten dieser Forschungsarbeiten um abgeschlossene Projekte, deren Ergebnisse nur begrenzt zur Verfügung stehen. Für einen detaillierten Vergleich dieser Technologien wird auf die Arbeit von Braig verwiesen ([Braig00a], Kap. 3.3).

Neben diesen als Entwicklungsbasis ungeeigneten Toolkits wurden in den letzten Jahren jedoch auch mehrere Forschungsarbeiten durchgeführt, die komponentenorientierte Ansätze zum High-Level-Authoring dreidimensionaler Anwendungen verfolgen und hier näher betrachtet werden sollen. In [Geige02] wird auf die Thematik des strukturierten Entwurfs von VE und 3D-Komponenten eingegangen. In [Dachs01a] wurde bereits eine Einteilung von 3D-Komponentenansätzen vorgenommen, die als Basis der hier dargestellten Klassifikation verwandter Arbeiten dient. Der Einteilung in fünf Gruppen liegt als Ordnungsprinzip die Entwicklung von rein Code-zentrierten bis hin zu Dokument-zentrierten Architekturen zugrunde.

4.3.1 Frühe Komponentenansätze

Bereits 1992 wurden mit der Entwicklung des Open Inventor 3D Toolkits [Strau92] Mechanismen vorgestellt, mit denen sich neue Knotentypen und Abstraktionen oberhalb des Szenengraphniveaus realisieren ließen. Die Inventor *Node Kits* waren mit ihrer Parametrisierung und Zusammenfassung von Subszenengraphen zu semantischen Einheiten und der programmtechnischen Umsetzung als DLL (Windows) / DSO (IRIX) bereits eine Art 3D-Komponente, die auch deklarativ in Inventor-Dokumenten instanziiert werden konnte. Auch VRML97 bietet ein ähnliches Konzept zur deklarativen High-Level-Wiederverwendung an, die *Prototypen*. Damit läßt sich eine Abstraktion vom Szenengraph-Knotenniveau jedoch nur schlecht und mit Skriptprogrammierung realisieren, auch erweist sich die Vermischung von Schnittstelle und Szenengraphrealisierung als ungünstig. Zudem ist Vererbung oder ein ähnliches Ableitungsprinzip bei Prototypen nicht möglich. Auch wenn in beiden Ansätzen Wiederverwendbarkeit unterstützt wird, sind sie auf das jeweilige Format zugeschnitten und im Bezug auf Konfiguration und Distribution limitiert.

4.3.2 Code-zentrierte Lösungen

Im Bereich der Programmierung dynamisch erweiterbarer, skalierbarer DVE ist NPSNET-V [Capps00] als wichtiger Vertreter zu nennen. Dafür wurde mit dem Komponentensystem *Bamboo* [Watse98] ein portables Framework und eine dynamisch durch Plug-Ins erweiterbare Laufzeitumgebung entwickelt, in der Code-Bausteine über Programmiersprachen- und Betriebssystemgrenzen hinweg interoperabel sind. Sogenannte *Module* können Verzeichnisstrukturen mit Plug-Ins, Quellcode, Geometrie-, Textur- und anderen Daten enthalten. Da weitere Komponentenmechanismen, wie z.B. CORBA, JavaBeans oder DCOM integriert werden können, ließe sich Bamboo auch unter der Rubrik 4.3.3 einordnen. XML wird für die Beschreibung von Netzwerkbotschaften verwendet, der NPSNET / Bamboo – Ansatz ist jedoch klar Code-zentriert und bietet verschiedenste Formen der Erweiterbarkeit.

Mit dem *Scene-Graph-As-Bus* [Zelez00] wurde ein Konzept zur Verbindung unabhängiger, verteilter, Szenengraph-basierter 3D-Applikationen vorgestellt. Dabei dient der *Neutral Scene Graph (NSG)* der Kopplung von Komponenten, die keinem speziellen Komponentenschnittstellenmodell genügen müssen. Die einzige Bedingung zur Kommunikation verteilter 3D-Applikationen ist ein beliebiges Szenengraphformat, das dann auf eine NSG-Schicht abgebildet wird. Nachteil dieses sehr auf verteilte, formatübergreifende 3D-Applikationen ausgerichteten Ansatzes ist die fehlende Abstraktion oberhalb des Szenengraphniveaus.

Ein Ansatz zur Visualisierung und Animation dreidimensionaler Produktkomponenten auf Basis von Java und Java3D wird in [Blanc01] vorgestellt. Java-Klassen für primitive und zusammengesetzte Objekte bzw. Komponenten mit getypten sogenannten *Contact Points* enthalten dabei spezifische, festgelegte Java3D-Szenengraphstrukturen. Mit einer XML-Beschreibungssprache werden Eigenschaften und Metadaten für Komponenten beschrieben. Das Programm *GenAu-3D (Generic Authoring Environment)* enthält verschiedene Werkzeuge zum Erstellen und Konfigurieren von Komponenten, wobei für Komponenten zwischen Autorenzeit und Laufzeit unterschieden wird. Bei diesem Ansatz wird eine Szenengraphabstraktion vorgenommen, wobei jedoch eine enge Verzahnung mit Java3D und Java-Programmierung nötig ist.

4.3.3 Systeme unter Nutzung existierender Komponententechnologien

Bei dieser Gruppe handelt es sich um Systeme, die Standard-Komponententechnologien modifizieren oder erweitern, um 3D-Grafik- bzw. Szenengraphfunktionalität zu integrieren. Der unter 4.3.2 vorgestellte *Bamboo*-Ansatz verbindet beispielsweise sehr flexibel OpenGL++ als grafische Realisierungsbasis mit Brücken zu verschiedenen Komponententechnologien. Allerdings erweist sich trotz dieser Flexibilität das relativ selten zum Einsatz kommende und nicht standardisierte OpenGL++ als ungünstiges 3D-Basisformat.

Eine konkretere Technologieverknüpfung wird mit den *Three-dimensional Beans* [Dörne00] umgesetzt, wobei das JavaBeans-Konzept mit Java3D-Szenengraphsemantik erweitert wird. 3D Beans sind von JavaBeans abgeleitet und fügen 3D-Bestandteile zur Schnittstelle hinzu. Ein visuelles Werkzeug, die *3D Beanbox*, erlaubt die Konfiguration und Assemblierung von Komponenten. In einer Erweiterung des Systems wird mit den *Meta Beans* [Dörne01] als Komponenten zur Kapselung von Interaktionsmetaphern eine Trennung von Geometrie und Verhalten vorgenommen. Meta Beans sind Teil des Java3D-Szenengraphs und immer mit einem 3D Bean assoziiert. Eine andere Erweiterung stellt die deklarative Auszeichnungssprache *Bean3D-ML* [Abawi01] dar, welche an die *Bean Markup Language (BML)* [BML@] angelehnt ist und mit der sich komponentenorientierte 3D-Szenen beschreiben lassen. Viele der unter 4.1 genannten Anforderungen werden mit diesem Ansatz erfüllt. Die Lösung ist jedoch trotzdem Code-zentriert, besitzt kein Austauschformat und ist von den erwähnten Technologien und Java als Programmiersprache abhängig.

4.3.4 Spezielle 3D-Komponentenansätze

Technologien dieser Gruppe haben konkrete 3D-Grafikformate oder 3D-APIs als Basis, die erweitert oder in eine eigene Architektur integriert werden, um Komponentenfunktionalität zu erreichen. Bei dem *i4D*-System des C-Labs Paderborn [Geige00] handelt es sich um eine flexible Schichtenarchitektur für die strukturierte Entwicklung von VR- und AR-Inhalten mit dem Schwerpunkt des schnellen Entwurfs komplexer Animationen. Dabei wird mit der High-Level-Metapher der sogenannten *Actors*, mit High-Level-Attributen und schließlich Aktionen zu deren kontinuierlicher Modifikation eine Abstraktion oberhalb des Szenengraphniveaus möglich. Komponenten werden – wie bei Open Inventor Nodekits – als DLL/DSO realisiert. Szenen lassen sich mit Dokumenten in einem eigenen XML-Format beschreiben. Die für mehrere Plattformen und 3D-Formate implementierte Mehrschichtenarchitektur unterstützt auch API-Programmierung und Skripting für schnelle Prototyperstellung und ist damit sehr flexibel einsetzbar. Erweiterungskonstrukte auf jeder Abstraktionsebene gestatten künftige Entwicklungen. Als Nachteile dieser Architektur erweisen sich die fehlende Plattformunabhängigkeit und die rein imperative Szenengraphrealisierung mit Skripten bzw. C++-Code.

Bei den *Smart Virtual Prototypes* [Salme00] handelt es sich um eine verteilte Architektur zur Darstellung komplexer virtueller Produkte mit Simulationen und Verhalten im Web. Der Komponentenbegriff zieht sich dabei durch alle drei Ebenen der Schichtenarchitektur, die aus – mit VRML Prototypen realisierten – GUI-Objekten sowie Java-Klassen für Interaktorkomponenten zur Verhaltensbeschreibung auf der Clientseite und virtuellen Komponenten auf der Serverseite besteht. Die Trennung zwischen grafischer Repräsentation und Funktionalität bzw. Verhalten wird durch das logische Simulationsmodell auf Serverseite und das Visualisierungs- und Interaktionsmodell auf Clientseite erreicht. Nachteil dieser Architektur ist ihre Komplexität vor allem im Autorenprozeß und die Heterogenität der Beschreibungsmittel und eingesetzten Technologien.

4.3.5 Dokumentzentrierte Ansätze

Zu dieser Rubrik zählen Architekturen, die auf deklarativen Dokumentbeschreibungssprachen für Komponentenschnittstellen, -konfiguration, -assemblierung und -verknüpfung basieren. Mit *Jamal* [Rudol99a] wurde ein deklaratives Komponentenframework für die Beschreibung von Beziehungen und Interaktionen zwischen Komponenten einer 3D-Anwendung vorgestellt. Die Jamal-Beispielimplementierung nutzt JavaBeans bzw. CORBA-Komponenten als Komponentenmodell sowie BML als deklarative Syntax für Komponentenverknüpfungen. BML-Szenen werden vom *JamalPlayer* geparst und mit Hilfe von Java3D dargestellt. Die auf einem flexiblen *Component Interface Model* basierenden Jamal-Komponenten stellen Abstraktionen zu Szenengraphkonzepten dar. Die Komponentenarchitektur wurde im Rahmen des Spezifikationsprozesses für X3D vorgestellt. Rudolph konnte in [Rudol99b] auch zeigen, daß die Isomorphismen zwischen VRML-Prototypen, X3D-Dokumenten, Java Beans und der *Interface Definition Language (IDL)* es erlauben, Komponentenschnittstellen in abstrakter Weise zu beschreiben. Erst später erfolgt dann die Übersetzung in eine konkrete 3D-Realisierung. Dieser Ansatz, der sich an Standards orientiert und bei dem erstmals eine deklarative Syntax für Komponentenschnittstellen in der 3D-Domäne eingesetzt wurde, beeinflußte auch die Entwicklung der CONTIGRA-Architektur, die ebenfalls zu dieser Technologie-Gruppe gezählt werden kann.

4.3.6 Klassifikationsschema und Fazit

Es läßt sich feststellen, daß die Grenzen zwischen den aufgeführten Kategorien nicht scharf gezogen werden können. Insbesondere durch die Erweiterungen von Formaten – z.B. um deklarative Beschreibungsmöglichkeiten oder die Unterstützung standardisierter Komponententechnologien – läßt sich eine eindeutige Einordnung nicht immer vornehmen. Das Diagramm

in Abbildung 14 ordnet die vorgestellten Technologien auf eine alternative Art. Auf der X-Achse wird das nötige Programmierungs-Niveau von prozeduraler über objektorientierte Programmierung und Skriptsprachen bis hin zu deklarativen Dokumenten dargestellt. Die Y-Achse zeigt die Verwendung von Komponententechnologien an, wobei zwischen keiner, speziell entwickelter, standardisierter softwaretechnologischer und beliebiger (durch Abstraktion) unterschieden wird. Auch in diesem Diagramm kann die Position der einzelnen Technologien nicht exakt festgelegt werden, wobei durch die Schattierung die Bandbreite der Unterstützung bzw. Entwicklungen innerhalb des Ansatzes aufgezeigt werden. Gerade auf der X-Achse wird deutlich, daß häufig mehrere Programmierstrategien angeboten werden. Das wird insbesondere bei *i4d* und *3D Beans* deutlich. *Jamal* und CONTIGRA bieten Abstraktionen zu Komponententechnologien und 3D-Formaten an und stützen sich auf deklarative Dokumente. Natürlich benötigen auch sie objektorientierte Programmierung oder Skripte, um komplexe Anwendungslogik zu realisieren, die nicht von Standardkomponenten abgedeckt wird. Somit ist die Position einzelner Technologien im Diagramm nur als Näherung zu betrachten und repräsentiert vor allem die Kernidee der jeweiligen Architektur.

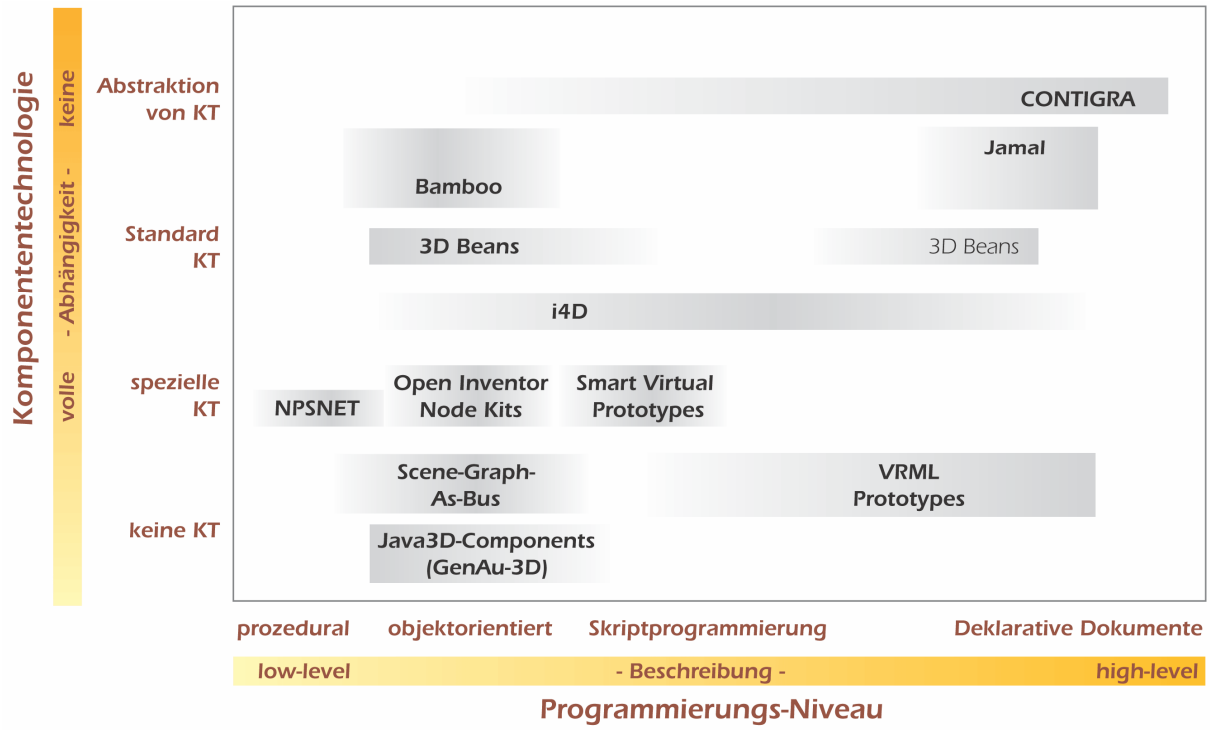

Abbildung 14: Klassifikation von 3D-Komponentenansätzen nach Programmierungs-Niveau und Abhängigkeit von Komponententechnologien

Die vorgestellten Ansätze haben eine Reihe von Stärken und eignen sich prinzipiell für die Erstellung von 3D-Komponenten. Neben dem Umstand, daß es sich häufig um abgeschlossene oder nicht weiterentwickelte Forschungsprojekte handelt, besitzen sie als Hauptnachteil zumeist bestimmte Abhängigkeiten von konkreten 3D-Formaten, Komponententechnologien, Programmiersprachen oder Realisierungsplattformen. Natürlich handelt es sich bei der unter 4.1 aufgestellten Forderung nach *Unabhängigkeit* um ein sehr schwieriges Kriterium, das sich immer nur teilweise erfüllen läßt. Die Anforderung, offene *Standards* zu verwenden, ist jedoch realisierbar, wird aber nur von wenigen vorgestellten Ansätzen erfüllt. Einige erfüllen zudem nicht das Kriterium der *High-Level-Sicht,* weil sie keinen konsequenten Abstraktionsansatz oberhalb des Szenengraphs anbieten.

Genau wie bei den traditionellen Softwaretechnologien ist trotz beinahe durchgängig verwendeter imperativer Ansätze der Trend zu externen deklarativen Beschreibungsdokumenten für Komponentenschnittstellen, aber auch 3D-Szenen insgesamt zu beobachten. Während bei einigen Arbeiten dafür spezialisierte XML-Auszeichnungssprachen verwendet werden, ist häufig jedoch eine Mischung verschiedener Beschreibungsformate zu finden, darunter IDL, *Data Sheets*, C-Header-Dateien, kurze textuelle Formate u.a. Dies verhindert eine einfache und konsistente Entwicklung und erfüllt nur teilweise die Anforderungen *Editierbarkeit, Interdisziplinäre Anwendungsentwicklung, Austauschbarkeit und alternative Werkzeuge*. Auch der Autorenprozeß wird nur bei den wenigsten Technologien durch eigene Werkzeuge oder etwa die Berücksichtigung verschiedener Entwicklerrollen unterstützt.

In Beantwortung der unter 4.1 gestellten Fragen und als Konsequenz der Analyse von allgemeinen Komponententechnologien und spezialisierten 3D-Ansätzen wurde für die technologische Basis ein eigener, neuartiger Ansatz gewählt, der sich zwar in Teilaspekten an existierenden Arbeiten orientiert, jedoch auf VRML97 bzw. X3D als 3D-Standardformat aufbaut und konsequent deklarativen Charakter hat. Konsistente Beschreibungssprachen auf XML-Basis ermöglichen auf verschiedenen Ebenen die Spezifikation, Konfiguration, Assemblierung und zu großen Teilen auch Implementierung von 3D-Widgets oder allgemein 3D-Komponenten. Auch der Autorenprozeß wurde konsequent an interdisziplinärem Arbeiten und visueller Erstellung ausgerichtet und ein prototypisches Werkzeug auf Basis der entstandenen Technologie realisiert. Der CONTIGRA-Lösungsansatz wird nachfolgend im Überblick dargestellt und in den Kapiteln 5 und 6 bezüglich des deklarativen Dokumentmodells und des Autorenprozesses mit seinen Werkzeugen im Detail präsentiert.

4.4 Der dokumentzentrierte Lösungsansatz

Nachdem Anforderungen an eine modulare Architektur aufgestellt, existierende Software-Komponentenmodelle untersucht sowie verwandte Arbeiten klassifiziert wurden, soll in diesem Unterkapitel zunächst der eigene Komponentenbegriff entwickelt und schließlich die mehrschichtige CONTIGRA-Architektur im Überblick erläutert werden (s.a. [Dachs01b] und [Dachs02]).

4.4.1 Der deklarative, dokumentzentrierte Komponentenbegriff

Alle existierenden Software-Komponentenarchitekturen bieten Beschreibungsmöglichkeiten für die Schnittstellen von Komponenten an, darunter *JavaBeans BeanInfo, EJB Deployment Descriptors* und *CORBA Component Descriptors*. Diese liegen in von Menschen lesbarer Form vor und basieren teilweise auf deklarativen XML-Formaten. Die Auslagerung der Komponentenbeschreibung in ein Markup-Dokument bietet einige Vorzüge. So können Schnittstelleninformationen und Komponentencode voneinander getrennt werden, womit eine separate Verarbeitung, Recherche und Selektion möglich wird. Außerdem werden Dokumentations- und Bearbeitungsmöglichkeiten von Komponenten vor allem auch durch die Anreicherung mit strukturierten Metadaten verbessert.

Mit der *Bean Markup Language* oder teilweise auch dem *CORBA Component Assembly Descriptor* ist zudem eine deklarative Beschreibung der Konfiguration, Kombination bzw. des Zusammenwirkens von Komponenten möglich. BML als XML-Grammatik gestattet beispielsweise die Beschreibung der Anwendungskomposition aus JavaBeans-Komponenten, wobei Komponenten instanziiert, *Properties* gesetzt und Ereignisquellen mit -zielen verknüpft werden können. Neben der deklarativen Beschreibung von Komponentenschnittstellen und deren Parametrisierung lassen sich Dokumente somit auch zur Konfiguration komponentenbasierter Anwendungen einsetzen.

Die Tendenz von rein imperativen Komponentenarchitekturen über die anteilige Beschreibung von Komponentenschnittstellen, -konfiguration und -verknüpfung läßt sich zu einem konsequent deklarativen Komponentenmodell fortführen. Dabei wird auch die Funktionalität bzw. Implementierung nicht mehr durch (eigenen) Programmcode, sondern deklarativ beschrieben. Selbstverständlich müssen solche Beschreibungen an einer Stelle in ausführbaren Code umgewandelt werden, so daß sich die programmierte Funktionalität komplett in Autorenwerkzeuge bzw. Player der entsprechenden Formate verlagert. Diese Entwicklung von programmzentrierten hin zu dokumentzentrierten Anwendungen läßt sich bereits seit längerer Zeit als Trend im Multimediabereich ausmachen. Letztlich können multimediale Dokumentformate, wie z.B. Flash oder Shockwave, durch geeignete Autorenwerkzeuge auch programmierunkundigen Entwicklern zugänglich gemacht werden. So bilden komplexe Dokumentformate und spezialisierte Auszeichnungssprachen immer häufiger die Basis multimedialer Anwendungen. Mit den XML-Standards *Synchronized Multimedia Integration Language (SMIL)* [SMIL@] für multimediale Präsentationen und *Scalable Vector Graphics (SVG)* [SVG@] für Vektorgrafiken wird dieser Trend besonders deutlich. Auch in [Encar00] wird statt algorithmischer Programmierung ein deskriptiverer, impliziter Ansatz für multimedial bestimmte Anwendungen vorgeschlagen.

Für die Erstellung interaktiver 3D-Grafikapplikationen wird somit ein dokumentzentrierter Komponentenansatz vorgeschlagen, der den Komponentendefinitionen von [Szype98] mit Ausnahme des Binärformates entspricht. Komponenten müssen also nicht notwendigerweise binäre Einheiten sein, sondern können auch ausschließlich durch strukturierte Dokumente beschrieben werden. Eine Realisierung der Komponentenfunktionalität kann in beliebiger Weise erfolgen, also z.B. prozedural, objektorientiert oder durch Skriptsprachen, möglichst

aber in deklarativer Form. Damit kommt als weitere Bedingung natürlich ein Ausführungsrahmen oder Player hinzu, der die in einzelnen oder mehreren Dokumenten beschriebenen Komponentenkonfigurationen, -aggregationen und -verknüpfungen zur Laufzeit umsetzen kann. Es wird eine *Blackbox*-Wiederverwendung vorgeschlagen, bei der ausschließlich die Komponentenschnittstelle bekannt ist und die Art der Realisierung verborgen bleibt.

Der hier beschriebene Komponentenbegriff hat seine Wurzeln im Projekt CHAMELEON, innerhalb dessen Formate und Werkzeuge für die Realisierung modularer, anpaßbarer und plattformunabhängiger Kursbausteine und Kursdokumente entwickelt wurden ([Wehne01], [Meißn01]). Während es sich bei diesem Projekt um die Anwendungsdomäne Lehr-/Lernanwendungen und bei der hier vorliegenden Arbeit um den Bereich interaktiver 3D-Applikationen handelt, wird im Projekt AMACONT eine Verallgemeinerung des Dokumentenmodells auf adaptive, aus konfigurierbaren Komponenten aufgebaute Webseiten vorgenommen [Fiala03].

Bei der CONTIGRA-Architektur handelt es sich also um ein 3D-Komponentenframework, welches komplett auf strukturierten Dokumenten basiert, mit denen sich von der Schnittstelle über die Komponentenimplementierung bis hin zur Konfiguration und Assemblierung zu komplexeren Szenen alle Details deklarativ beschreiben lassen. Dieser Lösungsansatz wurde entwickelt, weil die deklarative Beschreibung mehrere Vorteile bietet:

- Getrennte und trotzdem homogene Beschreibung von Geometrie, Erscheinungsbild, Verhalten, Komponenten und Konfigurationsmöglichkeiten sowie deren Zusammenwirken in zusammengesetzten 3D-Szenen.
- Einfaches Editieren von Hand oder mit syntaxgesteuerten Editoren bei gleichzeitiger Eignung für Autorenwerkzeuge.
- Dokumente können in einem offenen, von Menschen lesbaren Format zum Austausch zwischen verschiedensten Werkzeugen und Playern dienen.
- Große Ausdrucksmächtigkeit, Semantik und Kontrolle bereits in Dokumenten ohne Programmierung für einen größeren Kreis von Autoren möglich, damit Verlagerung der programmierten Interpretation der Formate in Player.
- Plattformübergreifend, durch XML-Nutzung auf Standards basierend oder verschiedene Standards integrierend.

4.4.2 Ebenen bei der Entwicklung komponentenbasierter 3D-Anwendungen

Die Abbildung 15 zeigt für die CONTIGRA-Architektur die Entwicklungsebenen komponentenbasierter 3D-Applikationen, die damit verbundenen Aufgaben, zugehörigen Dokumente und verwendeten Werkzeuge. Die deklarative CONTIGRA-Komponentenarchitektur orientiert sich dabei an der komponentenorientierten Softwareentwicklung. Bei dieser teilt sich der Lebenszyklus von Komponenten in die Hauptphasen Komponentenentwurf, Anwendungsentwurf (Assemblierung bzw. Komposition) und Laufzeit. Hier wurde noch die Phase der Distribution hinzugefügt.

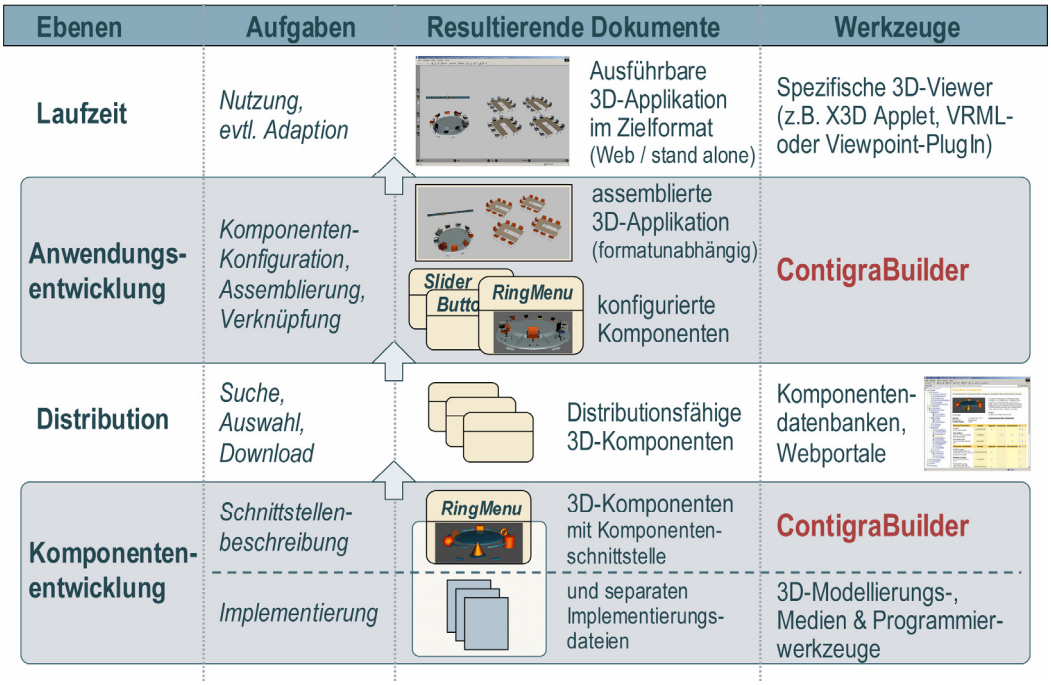

Abbildung 15: Komponentenentwicklungsebenen der CONTIGRA-Architektur

Die *Komponentenentwicklungsebene* umfaßt die Implementierung von 3D-Komponenten und die Spezifikation ihrer Schnittstelle für den Zugriff auf die Parameter bzw. Funktionalität der Komponente. Die Implementierung unterscheidet sich dabei insofern von der imperativen Programmierung konventioneller Software, daß Szenengraphen für die Beschreibung von Geometrie, Erscheinungsbild, Verhalten und auditiven Eigenschaften deklarativ erstellt werden. Lediglich für die Realisierung nicht als Baustein vorhandener oder durch Kombination von Komponenten erzielbarer Funktionalität muß diese imperativ programmiert werden. Dabei handelt es sich um einen vergleichsweise geringen Teil. Für die Erstellung der Implementierungsdateien kommen externe, allgemein gebräuchliche Werkzeuge und Editoren zum Einsatz, während das Zusammenwirken der verwendeten Szenengraphbestandteile und Subkomponenten CONTIGRA-spezifisch im Autorenwerkzeug CONTIGRABUILDER realisiert wird. Szenengraphexperten, Programmierer, aber auch 3D- und Audiodesigner sind an diesen Arbeiten beteiligt. Die Spezifikation der Komponentenschnittstelle erfolgt ebenfalls direkt über ein Beschreibungsdokument.

Die *Distributionsebene* umfaßt die Suche, Auswahl und Beschaffung von 3D-Komponenten über das Web. Distributionsfähige Komponenten liegen in kompakter Form vor, wobei lediglich ihre Schnittstelle als lesbares Dokument vorhanden ist und die Implementierung in – möglicherweise binärer oder geschützter – Paketform verborgen bleibt. Komponentendatenbanken auf Webservern können über Komponentenportale zugänglich sein. Auf dieser Ebene agieren nicht nur Programmierer, sondern auch Anwendungsentwickler, Konzeptionisten, Projektleiter, Designer u.a. Da diese Ebene auch entfallen kann, steht sie nicht im Mittelpunkt dieser Arbeit. Eine erste prototypische Lösung wird jedoch im Abschnitt 6.5.2.2 beschrieben.

Auf der Ebene der *Anwendungsentwicklung* erfolgt zunächst die Anpassung und Konfiguration ausgewählter Komponenten. Diese werden dann zu formatunabhängigen 3D-Anwendungen oder komplexeren Komponenten zusammengefügt, wobei neben Komponenten auch weitere Szenengraphbestandteile und zusätzliche Funktionalität ergänzt werden können. Komponenten werden außerdem untereinander und gegebenenfalls mit anderen Szenengraphbestandteilen verknüpft, was man als verbindungsorientierte deklarative Programmierung bezeichnen könnte. Diese Aufgaben können ebenfalls im Autorenwerkzeug CONTIGRA-

BUILDER durch Experten verschiedener Fachgebiete visuell und ohne Programmierkenntnisse bearbeitet werden. Möglich – jedoch untypisch – wäre auch die textuelle Bearbeitung von Hand, da für die deklarative Beschreibung aller genannten Aspekte ebenfalls ein eigenes Dokumentformat entwickelt wurde. Auch der CONTIGRABUILDER benutzt dieses Format intern zur Abspeicherung kompletter 3D-Applikationen.

Aus diesen Dokumenten kann im CONTIGRABUILDER oder über extern aufrufbare Transformationen schließlich eine auf der *Laufzeitebene* ausgeführte Applikation für ein konkretes 3D-Zielformat generiert werden. Alltagsnutzer oder Domänenexperten führen diese Anwendungen entweder *stand-alone* oder über das Web aus, wobei entsprechende Player bzw. Viewer für das jeweilige Format vorhanden sein müssen. Perspektivisch sind hierbei Adaptionen an konkrete Engeräte und Nutzerpräferenzen möglich. Auch die Laufzeitebene stand zunächst nicht im Vordergrund der Entwicklung, statt dessen jedoch die in der Abbildung 15 grau hinterlegten Ebenen *Komponentenentwicklung* und *Anwendungsentwicklung*. Durch die Trennung von Komponenten- und Anwendungsentwurf in Form verschiedener Dokumente (und der zugrundeliegenden Beschreibungssprachen) sowie durch das Autorenwerkzeug CONTIGRABUILDER wird Wiederverwendbarkeit explizit unterstützt. Das Werkzeug wird zusammen mit dem Autorenprozeß in Kapitel 6 ausführlich beschrieben.

4.4.3 Die deklarativen Beschreibungssprachen

Das Herz des CONTIGRA-Ansatzes stellen die auf der Basis der *Extensible Markup Language (XML)* [XML@] entwickelten Auszeichnungssprachen für die einzelnen Komponentenebenen dar, mit denen eine konsistente und durchgängig deklarative Beschreibung interaktiver 3D-Applikationen möglich wird. Diese werden im Detail in Kapitel 5 vorgestellt und hier nur im Überblick erläutert. Abbildung 16 zeigt noch einmal die Aufgaben für die unter 4.4.2 beschriebenen Ebenen, dazu die jeweiligen XML-Sprachen und ihre korrespondierenden Instanzdokumente, die wiederum verschiedene Szenengraphdateien und auch Dateien anderen Typs referenzieren.

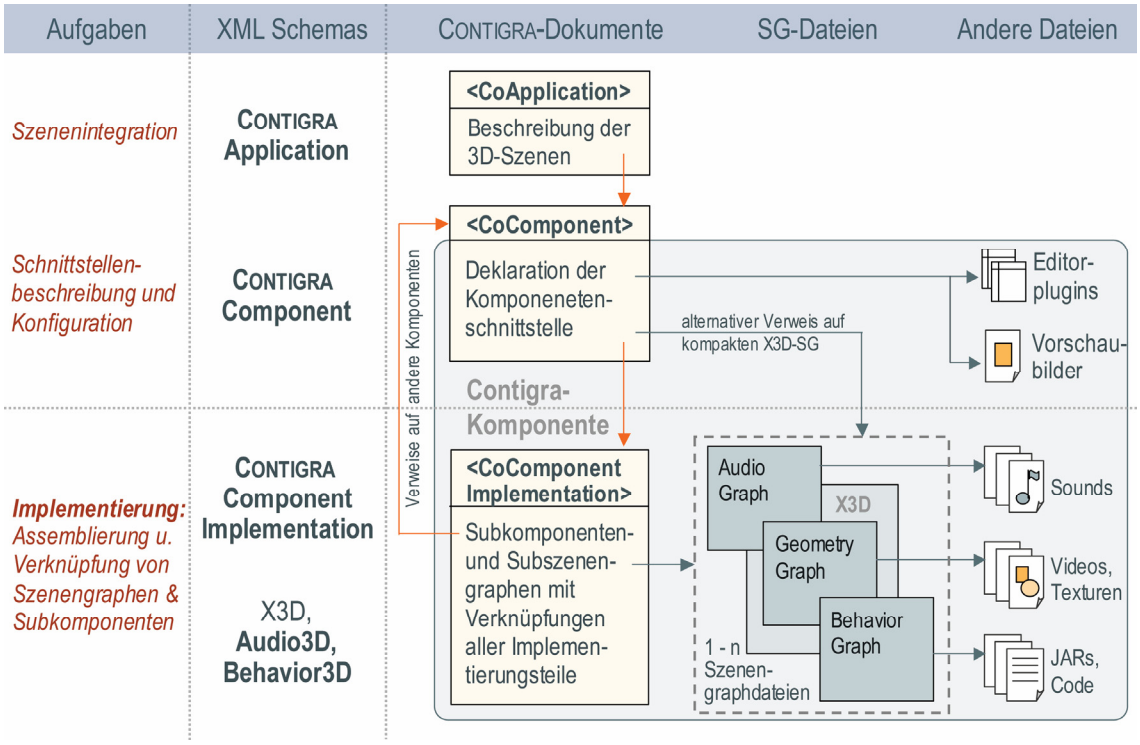

Abbildung 16: Die Auszeichnungssprachen und korrespondierenden Dokumente für die Ebenen der CONTIGRA-Architektur

Eine CONTIGRA-Applikation besteht aus einem Dokument, welches der unter 5.2 beschriebenen XML-Grammatik *CoApplication* genügt. Es enthält typische Szenenparameter, wie Kameraeinstellungen, Blickpunkte, Beleuchtung und globale Audioeigenschaften. Jede Applikation verweist auf eine *CoComponent*-Datei, die eine komplexe Komponente beschreibt, innerhalb derer Subkomponenten und weitere Szenengraphbestandteile die gesamte 3D-Anwendung darstellen. Alle Verweise von CONTIGRA-Instanzdokumenten untereinander sind in der Abbildung 16 als orange Pfeile dargestellt.

Eine CONTIGRA-Komponente (im Bild grau hinterlegt) besteht zunächst aus einem Schnittstellen- und einem Implementierungsdokument. Die Komponentenschnittstelle wird mit Hilfe der XML-Grammatik *CoComponent* beschrieben, die unter 5.3 näher vorgestellt wird. In einem Instanzdokument dieser Grammatik werden zahlreiche Metainformationen zu Hersteller, Dokumentation der Komponente, Anwendung im Autorenprozeß, Lizenzmodell etc. beschrieben. Dazu kommen für den Autorenprozeß noch Verweise auf Vorschaubilder, Icons und spezialisierte Editor-Plug-Ins. Den Hauptteil des Schnittstellendokuments bildet jedoch die Deklaration der konfigurierbaren Parameter, welche die Funktionalität der Komponente repräsentieren und bereits mit konkreten Werten versehen sind. Somit handelt es sich bei Schnittstellendokumenten um eine Art von Prototypen in Dokumentform, da sie kein reines Interface (wie z.B. eine Java Interface-Klasse oder ein C++-Header) darstellen, sondern eher als bereits instanziiertes Objekt mit konkreten Wertausprägungen aufgefaßt werden können. Eine Komponente läßt sich dadurch konfigurieren, daß für die im *CoComponent*-Instanzdokument angegebenen Parameter einfach andere Werte eingesetzt werden. Typisch ist somit ein Verhältnis von 1:n zwischen Implementierungs- und Schnittstellendokumenten, während das Verhältnis bei der objektorientierten Programmierung umgekehrt ist. Schließlich enthält das Schnittstellendokument einer Komponente auch einen Verweis auf ihre Implementierung. Im Falle einer distributionsfähigen Komponente handelt es sich um eine Referenz auf eine kompakte X3D-Szenengraphdatei, da X3D intern auch als Szenengraphbasis verwendet wird. Im Entwicklungsfall wird ein *CoComponentImplementation*-Dokument referenziert, welches den detaillierten Aufbau einer Komponente transparent für den Autorenprozeß beschreibt.

Das Implementierungsdokument genügt der unter 5.4 näher beschriebenen XML-Grammatik *CoComponentImplementation.* Darin befindet sich ein Subkomponentengraph in Form einer Transformationshierarchie von verwendeten Komponenten, die in der Regel als Referenzen auf weitere *CoComponent*-Instanzdateien integriert werden. Dazu kommt ein Subszenengraph als Transformationshierarchie mit Verweisen auf ein oder mehrere separate Szenengraphdateien. Dabei wird in einen *Geometriegraph, Verhaltensgraph* und *Audiograph* unterschieden. Die jeweils referenzierten Dokumente genügen im Falle der Geometrie dem standardisierten 3D-Format X3D bzw. für Audio und Verhalten den auf X3D abbildbaren Formaten *Audio3D* [Hoffm03] und *Behavior3D* [Dachs03a]. Diese Szenengraphformate werden unter 5.4.3 und 5.4.4 im Detail beschrieben.

Mediendateien, wie Texturen, Sounds oder Videos, werden stets separat von den Szenengraphdokumenten aus referenziert. Dazu kommen gegebenenfalls auch Java-Klassen bzw. Java-Archive, die Funktionalität für den Verhaltensgraph enthalten, welche sich nicht mit vorgefertigten Verhaltensbausteinen bzw. durch die Verknüpfung von Komponenten realisieren läßt.

4.4.3.1 Relation von CONTIGRA zu X3D

X3D wurde deshalb als internes Implementierungsformat gewählt, weil es sich im Prozeß der Standardisierung befindet, zahlreiche 3D-Modellierungsprogramme und Autorenwerkzeuge Exportfunktionalität für X3D anbieten, verschiedene Player existieren und die XML-Kodierung einfach in andere Formate konvertiert werden kann. Daher erfolgt auch für distributionsfähige Komponenten die Zusammenfassung sämtlicher Implementierungsdateien und

des *CoComponentImplementation*-Instanzdokuments zu einer großen und kompakten X3D-Szenengraphdatei, die dann zusammen mit dem Schnittstellendokument auf *CoComponent*-Basis eine Komponente darstellt.

Damit kann CONTIGRA auch als eine weitere Abstraktions- und Indirektionsebene zu X3D betrachtet werden. Es ergibt sich die Frage, ob das X3D-Wiederverwendungskonzept in Form von Prototypen nicht für die beabsichtigten Ziele dieser Arbeit als Lösungsansatz ausreichen würde. Einige Argumente werden im folgenden aufgeführt, warum der Komponentenansatz von CONTIGRA oberhalb von X3D seine Berechtigung hat.

- Die klare Abstraktion gegenüber der Szenengraphebene bietet neben der (auch mit Prototypen möglichen) Kapselung vor allem ein verbessertes Parameterkonzept (s. 5.3.3), das weit über die Möglichkeiten von Feldern eines X3D-Knotens hinausgeht (z.B. die 1:n – Abbildung eines CONTIGRA-Parameters auf verschiedene Szenengraphfelder).
- Das parametrisierbare Verknüpfungskonzept zwischen Subkomponenten (mit ihren Parametern) und Szenengraphknoten (mit ihren Feldern) hat kein Äquivalent in X3D (s. 5.4.5 für Details).
- CONTIGRA funktioniert als deklarative Komponentenarchitektur unabhängig von X3D und kann auch komplett in Java3D, OpenSG, Viewpoint oder andere Formate transformiert werden. Die immer noch nicht abgeschlossene Standardisierung von X3D und Unklarheiten in vielen Detailfragen (XML-Schema, SAI, Erweiterungsmechanismen etc.) rechtfertigen die unabhängige Entwicklung zusätzlich.
- CONTIGRA erlaubt auch die Integration weiterer multimedialer Bestandteile, die in X3D nicht ohne weiteres möglich sind. Ein Beispiel sind die erheblichen Raumklangerweiterungen durch das Format *Audio3D* (s. 5.4.3).
- Die Grammatik *CoComponent* funktioniert auch als Komponentenspezifikationssprache völlig unabhängig von jeglicher Implementierung und bietet neben der Beschreibung von Parametern und ihrer Konfiguration zahlreiche Metainformationen. Ein solches Konzept bietet X3D nicht.