VoxML Dokumentation

Mark Klement
Goethe Universität
Frankfurt am Main
s0058073@stud.uni-frankfurt.de

Abstract. Dies ist eine Dokumentation und Einführung in die Modellierungssprache VoxML, welche es erlaubt, semantische Informationen, Interaktionsmöglichkeiten und Eigenschaften von 3D Modellen zu speichern und zu verarbeiten. VoxML soll durch die Speicherung einer breit gefächerten Menge an semantischen Informationen in der Lage sein, die Einschränkungen anderer 3D Modellierungssprachen zu überwinden, und multimodale Simulationen von realistischen Szenarien zu ermöglichen. Im nachfolgenden soll ein Überblick über die Struktur, Syntax und Entity Typen von VoxML gegeben werden, der auch von bisher themenfremden Personen verstanden werden kann.

1. Einleitung

1.1 Problemstellung

Die Ausdrucksstärke natürlicher Sprache lässt sich nur schwer mit den uns zur Verfügung stehenden, computergestützten Mitteln visualisieren. Deshalb hat sich die Forschung bisher größtenteils darauf beschränkt, natürlichsprachige Ausdrücke in statische 3D Szenen umzuwandeln. Mit VoxML soll es jedoch nicht nur möglich sein, statische, sondern auch dynamische 3D Szenen aus natürlichsprachigem Text zu generieren. Dazu konzentriert man sich speziell auf die Modellierung von Verben, welche Bewegungen jeglicher Art implizieren.

1.2 Theoretischer Hintergrund

Im Wesentlichen lassen sich Bewegungsverben in zwei Komponenten auftrennen. Die erste Komponente, der „Pfad“ (engl. path), gibt eine räumliche Position und Richtung der Bewegung relativ zu einem Referenzobjekt an. Die zweite Komponente ist die „Art und Weise der Bewegung“ (engl. manner of motion), welche Auskunft über Dinge wie die Fortbewegungsform, Fortbewegungsmethode oder den Kraftaufwand gibt.

Ein weiterer, wichtiger Bestandteil zur korrekten Umwandlung eines Ausdrucks in eine Simulation ist der situationsbedingte Kontext und damit einhergehende Anforderungen an das Verhalten von Objekten. Der situationsbedingte Kontext entsteht durch die Gegebenheiten der Umwelt oder 3D Szene, und hat selbst Einfluss auf das Verhalten von Akteuren und Objekten, die sich in ihm befinden. Beispielsweise sollte man einen Löffel senkrecht und nicht waagerecht in einen Becher stellen, was allerdings nur dann möglich ist, wenn die Öffnung des Bechers im Weltkoordinatensystem nach oben zeigt.

Aus diesem Grund resultieren die Ausdrücke „Der Löffel fällt“ und „Der Löffel fällt in den Becher“ in verschiedenen Simulationen.

2. Dokumentenstruktur

Dieser Abschnitt befasst sich mit dem allgemeinen Aufbau und der Syntax von VoxML Dokumenten. Auf die Umsetzung Entity-Typspezifischer Einzelheiten wird in Abschnitt 3 genauer eingegangen.

2.1 XML als Grundlage

Die Grundlage von VoxML bildet die tagbasierte eXtensible Markup Language (XML). Informationen werden dabei in ineinander verschachtelten Tags gespeichert, wodurch für die Tags und Informationen eine Baumstruktur entsteht. Die erste Zeile eines VoxML Dokuments ist die Kopfzeile, welche eine XML Deklaration mit den Attributen „`version`“, „`encoding`“ und „`standalone`“ enthält. Das Attribut „`version`“ bezieht sich auf die verwendete XML-Version und kann auf den Wert „`1.0`“ gesetzt werden, da XML sich immer noch in der ersten Version befindet. Das Attribut „`encoding`“ beschreibt den verwendeten Zeichensatz, wobei der Standard für VoxML Dokumente „`us-ascii`“ ist. Das letzte Attribut der Kopfzeile „`standalone`“ ist optional und wird durch weglassen automatisch auf den wert „`no`“ gesetzt.

Eine typische XML Deklaration für VoxML Dokumente benötigt das optionale Attribut „`standalone`“ nicht, und sieht damit folgendermaßen aus:

```
<?xml version="1.0" encoding="us-ascii"?>
```

Beispiel 2.1.1: XML Deklaration

Unterhalb der XML Deklaration folgt eine weitere, für jedes VoxML Dokument notwendige Zeile, welche das VoxML-Tag öffnet und Definitionen für Namensräume enthält. Da das VoxML-Tag das Erste ist, das geöffnet, und das Letzte, das geschlossen wird, bildet das VoxML-Tag eine Art Body mit dem Inhalt des Dokuments:

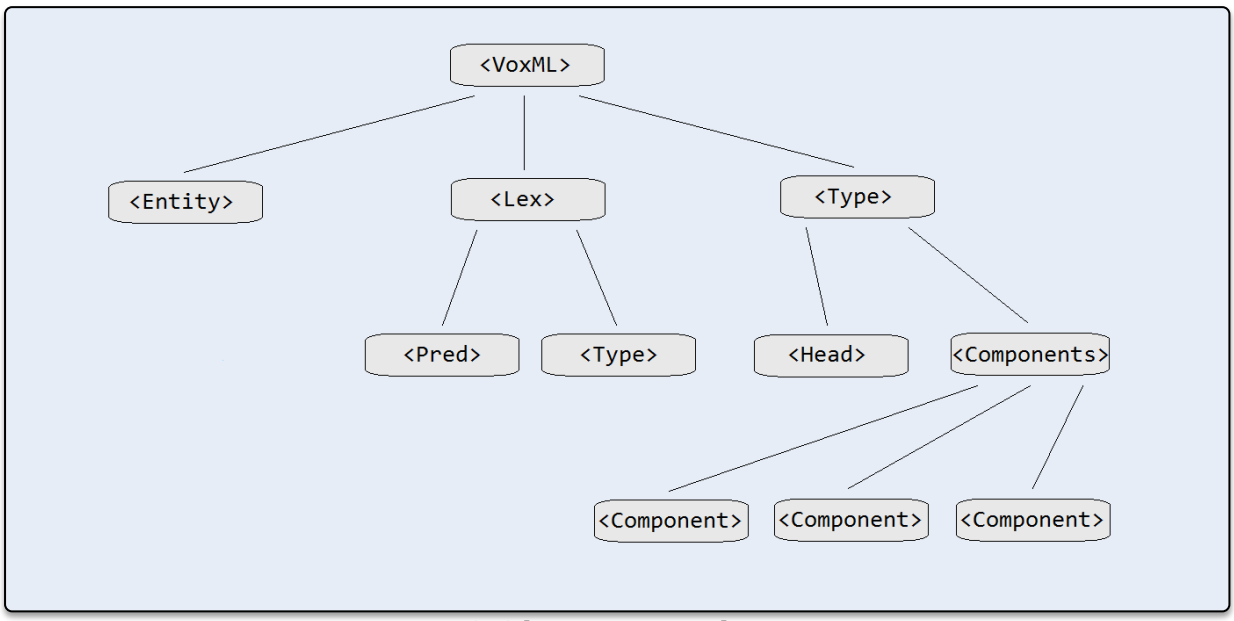
```
<VoxML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema">

… Inhalt …

</VoxML>
```

Beispiel 2.1.2: VoxML Body

Würde man die Struktur des Dokuments als einen Baum darstellen, so entspräche das VoxML-Tag der Wurzel dieses Baumes:



Beispiel 2.1.3: Baumstruktur

2.2 Syntax

Da jedes VoxML Dokument ein XML Dokument ist, stimmt auch die Syntax von VoxML mit der von XML überein. Das bedeutet, dass die einzelnen Elemente des VoxML Dokuments mit einem Start-Tag beginnen, und mit einem End-Tag enden. Die Länge der Namen der Elemente ist unbegrenzt und sie dürfen sämtliche alphanumerische Zeichen, sowie Unterstriche, Bindestriche und Punkte enthalten. Namen müssen jedoch zwingend mit einem Unterstrich oder einem Buchstaben beginnen und die Schreibweise ist case-sensitive. Alle Zeichen zwischen den Tags eines Elements sind dessen Inhalt.

```
<Pred>bowl</Pred>
```

Beispiel 2.2.1: Element Pred (Predicate) mit Inhalt „bowl“

Elemente müssen nicht zwangsläufig einen Inhalt haben, sondern können auch leer sein. Elemente ohne Inhalt werden in einem Tag, welches mit einem Slash endet, zusammengefasst. Auf diese Weise lässt sich unter anderem die Abwesenheit eines bestimmten Elements ausdrücken, ohne das Element komplett aus dem Code zu entfernen.

```
<RotatSym />
```

Beispiel 2.2.2: Element RotatSym ohne Inhalt

Elemente können an Stelle von/ zusätzlich zum Inhalt auch Attribute haben. Semantisch äquivalente Umstände können so auf syntaktisch unterschiedliche Weise dargestellt werden. Während XML es zulassen würde, die Komponenten eines 3D Objektes als Elemente mit Inhalt zu beschreiben, werden Komponenten in VoxML durch Elemente ohne Inhalt, aber mit Attribut dargestellt.

```
XML:      <Component>cabinet_frame</Component>

VoxML:    <Component Value="cabinet_frame" />
```

Beispiel 2.2.3: Element mit Inhalt vs. Element mit Attribut

Wie bereits erwähnt, ist es möglich die Elemente in einer Baumstruktur zu verschachteln. Das heißt, dass Elemente andere Elemente als Inhalt haben können. Wichtig ist dabei nur, dass ein Element das andere vollständig beinhaltet, also die Start- und End-Tags (oder Inhalte) zweier Elemente sich nicht überschneiden.

Erlaubt:

```
<Lex>
  <Pred>
    box_cabinet
  </Pred>
  <Type>
    physobj
  </Type>
</Lex>
```

Verboten:

```
<Lex>
  <Pred>
    box_cabinet
  </Pred>
  <Type>
    physobj
</Lex>
</Type>
```

Beispiel 2.2.4: zulässige vs. unzulässige Verschachtelung

Kommentare dürfen in VoxML Dokumenten überall dort stehen, wo normaler Text erlaubt ist. Das heißt, nach der XML Deklaration innerhalb der Elemente, nicht aber in deren Tags. Kommentare beginnen mit „<!---“ und enden mit „-->“.

```
<!--<Component Value="leg[2]+" />-->
```

Beispiel 2.2.5: Auskommentierte Komponente „leg“ des Objektes „table“

Zuletzt bleibt noch die Erwähnung der Zeichen mit besonderer Bedeutung. Diese Sonderzeichen dürfen nicht in Attributwerten oder Inhalten von Tags vorkommen. Möchte oder muss man diese Zeichen trotzdem verwenden, hat man zwei Möglichkeiten: Entweder, man arbeitet mit CDATA-Sektionen, in denen diese Sonderzeichen nicht als solche interpretiert werden, oder man verwendet deren Entity-Referenzen. Während sich CDATA-Sektionen hervorragend für lange Zeichenketten eignen, die besonders viele Sonderzeichen enthalten, eignet sich das Ersetzen der Sonderzeichen durch ihre Entity-Referenzen eher, wenn man nur wenige Sonderzeichen verwendet.

Entity-Referenz	Zeichen
&lt;	<
&gt;	>
&amp;	&
&quot;	„
&apos;	‘

Tabelle 2.2.6: Sonderzeichen und deren Entity-Referenzen

3. Entity Typen

Die Entity Typen von VoxML repräsentieren die wichtigsten Bausteine natürlicher Sprache (Nomen, Verben, etc.). Da die Modellierung eines Wortes in VoxML von seinem Entity Typ abhängig ist, soll in diesem Abschnitt auf die Umsetzung typspezifischer Einzelheiten eingegangen werden. Der Entity Typ wird in VoxML immer durch das Element „`Entity`“ mit dem Attribut „`Type`“ festgelegt.

```
<Entity Type="Object" />
```

Beispiel 3.0.1: Entity Typdeklaration

Entity-Typ	`<Entity Type="..." />`
Objekt	`Object`
Programm	`Program`
Attribut	`Attribute`
Relation	`Relation`
Funktion	`Function`

Tabelle 3.0.2: Entity Typdeklaration

Ein weiteres Element, das in jedem VoxML Dokument vorkommt, ist „`Lex`“ (lexical information), welches immer das Element „`Pred`“ (predicate) beinhaltet. Das Prädikat beschreibt möglichst kurz den Inhalt des VoxML Dokuments. In der Praxis wird als Prädikat einfach der Name, also die natürlichsprachige Bezeichnung des Gegenstandes, der Relation, etc. verwendet. Ein weiteres Element, das im Falle von Objekten und Programmen zu den lexikalischen Informationen gehört, ist „`Type`“. Zur Modellierung von Attributen, Relationen und Funktionen ist dieses Element nicht notwendig. An dieser Stelle ist allerdings Vorsicht geboten, da es noch ein weiteres Element mit dem Namen „`Type`“ gibt, welches aber nicht innerhalb der lexikalischen Informationen steht, und für jedes VoxML Dokument benötigt wird.

```
<Lex>
  <Pred>square_table</Pred>
  <Type>physobj*artifact</Type>
</Lex>
```

Beispiel 3.0.3: Lexikalische Informationen eines Objektes „square_table“

3.1 Objekte

Objekte werden in VoxML verwendet, um Nomen zu modellieren. Meist handelt es sich dabei um physische Objekte, mit denen entweder ein Agent, die Umwelt oder beide interagieren können. In VoxML werden Objekte schon durch das Element „`Type`“ in den Lexikalischen Informationen in zwei Kategorien unterteilt. Diese Unterscheidung richtet sich nach den Interaktionen, die mit dem Objekt möglich sind, und dem Zweck, den das Objekt erfüllt. Für Objekte, die keinen bestimmten Nutzen haben, und mit denen nur eine recht allgemeine Interaktion wie „aufheben“, „bewegen“ oder „halten“ möglich ist, bekommt das Element „`Type`“ den Inhalt „`physobj`“. Objekte mit spezifischem Nutzen und Interaktionsmöglichkeiten bekommen als „`Type`“ den Wert „`physobj*artifact`“ zugewiesen.

Natürlich gibt es dabei einen gewissen Interpretationsspielraum wo die Grenze zwischen allgemein und zielgerichtet liegt. Auf einem Tisch kann man Beispielsweise Dinge abstellen, was auch gleichzeitig die Hauptaufgabe des Tisches ist. Allerdings kann man viele Gegenstände verwenden, um Dinge darauf abzustellen, was diese Interaktion wieder recht allgemein erscheinen lässt. Es ist jedoch auch möglich, dass ein Objekt zwar eine stark zielgerichtete Funktion hat, das entsprechende Verb zur Erfüllung dieser Funktion aber noch nicht modelliert wurde. Das trifft beispielsweise auf alle Lebensmittel und das Verb „eat“, also „essen“ zu. In solchen Fällen ist die Verwendung beider Inhalte für das Element „`Type`“ möglich.

Objekt	Verb nicht modelliert	Verb modelliert
Tisch	`egal`	`egal`
Apfel	`egal`	`physobj*artifact`
Boden	`physobj`	`physobj`

Tabelle 3.1.1: Typ einiger Beispielobjekte

Da benutzbare Objekte meist aus mehreren Einzelteilen bestehen, die einzeln oder in kleinen Gruppen einen Zweck erfüllen, der zum Gesamtnutzen des Objektes beiträgt, sieht VoxML eine Zerlegung dieser Objekte vor. Diese Zerlegung findet innerhalb des „`Type`“ Elements für geometrische Informationen, das außerhalb der Lexikalischen Informationen liegt statt. Es ist möglich einzelne Komponenten mit einer Zahl in eckigen Klammern zu versehen, und so eine Art Verweis für den schnelleren Zugriff auf diese Komponenten zu kreieren. Besonders geeignet ist dieses Vorgehen für Funktionstragende Teile des Objekts. Komponenten, die mehrfach vorkommen, aber eine Einheit bilden und jeweils immer den gleichen Zweck erfüllen, werden mit einem „`+`“ versehen. Die geometrischen Informationen beinhalten neben der Zerlegung des Objektes in Komponenten auch ein übergeordnetes Element „`Head`“, welches versucht mit einer möglichst primitiven Form das Objekt oder dessen wichtigstes Teil zu beschreiben. Dazu kommen noch das Element „`Concavity`“, das die Wölbung eines Objektes beschreibt, und die zwei Elemente „`RotatSym`“ und „`ReflSym`“. „`RotatSym`“ meint jedoch nicht, dass das Objekt ein Rotationskörper sein muss, sondern lediglich, dass das Objekt durch Drehung um gewisse Winkel entlang einer Achse auf sich selbst abbildet. „`RotatSym`“ beinhaltet folglich einzelne Achsen der Weltkoordinaten und „`ReflSym`“ Ebenen bestehend aus zwei der drei Weltkoordinatenachsen.

```
<Type>
   <Head>sheet[1]</Head>
   <Components>
     <Component Value="surface[1]" />
     <Component Value="leg[2]+" />
   </Components>
   <Concavity>Flat</Concavity>
   <RotatSym>Y</RotatSym>
   <ReflSym>XY,YZ</ReflSym>
</Type>
```

Beispiel 3.1.2: Geometrische Informationen eines Objektes „square_table“

Ein weiteres Element zur Modellierung von Objekten ist „`Habitat`“, welches den situativen Kontext eines Objektes beschreibt. Es beinhaltet die intrinsischen Anforderungen, die allgemein an ein Objekt gestellt werden, und die extrinsischen Anforderungen, welche zur Ausübung bestimmter Interaktionen erfüllt sein müssen. Anforderungen, die zusammengehören können ähnlich, wie bei den Komponenten mit Zahlen in eckigen Klammern gekennzeichnet werden. Für viele Objekte beinhalten die Anforderungen die korrekte Ausrichtung in den Weltkoordinaten. Legt man beispielsweise einen Löffel irgendwo ab, so liegt er flach auf der Oberfläche (intrinsisch). Möchte man hingegen etwas mit dem Löffel umrühren, so muss er senkrecht in den Becher gestellt werden (extrinsisch). Die Parameter von „`align`“ sind die Achsen des Koordinatensystems des Objekts und des Koordinatensystems der Umgebung, in der sich das Objekt befindet.

```
<Habitat>
    <Intrinsic>
        <Intr Name="UP[4]" Value="align(Y,E_y)" />
        <Intr Name="TOP[4]" Value="top(+Y)" />
    </Intrinsic>
    <Extrinsic>
        <Extr Name="UP[5]" Value="align(Y,E_x|E_z)" />
    </Extrinsic>
</Habitat>
```

Beispiel 3.1.3: Habitat eines Objektes „spoon“

Zusätzliche Informationen und Restriktionen können in Form von constraints modelliert werden. So muss Beispielsweise die Dicke der Platte eines Tisches deutlich niedriger sein, als die Breite und die Tiefe. Die Beine eines Tisches sollten jedoch deutlich höher sein, als ihr Durchmesser.

```
<Intrinsic>
  <Intr Name="UP[3]" Value="align(Y,E_y)" />
  <Intr Name="TOP[3]" Value="top(+Y)" />
  <Intr Name="CONSTR[3]" Value="Y[1] &lt;&lt; Z[1], Y[1] &lt;&lt; X[1]" />
  <Intr Name="CONSTR[3]" Value="Y[2] &gt;&gt; X[2], Y[2] &gt;&gt; Z[2]" />
</Intrinsic>
```

Beispiel 3.1.4: Constraints für einen Tisch

Das Element „`Afford_Str`“ beinhaltet die Beschreibungen der Interaktionen eines Objektes. Die Interaktionsformeln beginnen immer mit dem Habitat, in dem sich das Objekt befindet, da dieses Entscheidend für den Ausgang einer Interaktion ist. Zusätzlich kann noch der aktuelle Zustand des Objektes einbezogen werden. Beispielsweise kann man mit einem Löffel nur dann etwas umrühren, wenn er sich Senkrecht in der Szene befindet, und bereits in einem Behälter steckt. Danach erfolgt die Parameterangabe für die anzuwendende Interaktion und schließlich ein optionales Endstadium des Objektes.

```
<Afford_Str>
   <Affordances>
      <Affordance Formula="H-&gt;[grasp(x, [1])]hold(x, [1])" />
      <Affordance Formula="H[4]-&gt;[put(x, in([3]))]contain([3], x)" />
      <Affordance Formula="H[4]-&gt;[grasp(x, [2])]" />
      <Affordance Formula="H-&gt;[lift(x, [1])]hold(x, [1])" />
      <Affordance Formula="H[5]-&gt;[put([1], in(x))]contain(x, [1])]" />
      <Affordance Formula="H[5],contain(x, [1])-&gt;[stir([1])]" />
   </Affordances>
</Afford_Str>
```

Beispiel 3.1.5: Interaktionsmöglichkeiten mit einem Löffel

Als letztes verfügt jedes Objekt über das Element „`Embodiment`“, in dem eine ungefähre Größenordnung im Verhältnis zu einem Agenten, und eine Bool-Angabe über die Bewegbarkeit enthalten sind

```
<Embodiment>
    <Scale>&lt;agent</Scale>
    <Movable>true</Movable>
</Embodiment>
```

Beispiel 3.1.6: Embodiment eines Löffels

3.2 Programme

Programme werden in VoxML genutzt, um Verben zu modellieren. Da das Einsatzgebiet von VoxML die automatische Generierung von bewegten 3D Szenen ist, liegt der Fokus insbesondere auf Verben, die Bewegungen von Agenten oder Objekten implizieren. Programme bilden neben den Objekten den wichtigsten Entity-Typ in VoxML.

Die Lexikalischen Informationen von Programmen beinhalten neben dem Prädikat noch ein Element, welches den Typ der Bewegung beschreibt. Unterschieden wird hier zwischen „`transition_event`“, „`process`“ und „`state`“. „`state`“ wird benutzt, um mehr oder weniger statische Zustände wie „hold“ zu Beschreiben. Als „`transition_event`“ werden solche Verben bezeichnet, die die Veränderung des Zustandes eines Objektes zum Ziel haben. Ein Beispiel hierfür ist „grasp“. Ein Objekt, welches vorher in der 3D Szene lag, befindet sich nach dem Aufheben durch den Agenten in dessen Hand, was gegebenenfalls neue Interaktionen zwischen Agent und Objekt ermöglicht. Verben, deren Typ „`process`“ ist, haben meist die übergeordnete Aufgabe Objekte zu bewegen, schließen Zustandsveränderungen allerdings nicht aus.

```
<Lex>
  <Pred>put</Pred>
  <Type>process</Type>
</Lex>
```

Beispiel 3.2.1: Lexikalische Informationen des Verbs „put“

Programme können als Argumente neben Agenten, Objekten und Koordinaten auch andere Funktionen und Programme entgegennehmen, oder diese als Subroutine aufrufen. Ein Beispiel für ein solches Programm ist „put“, welches zur Erledigung seiner Aufgabe die Programme „grasp“, „hold“, „move“ und „ungrasp“ aufruft.

```
<Type>
  <Head>process</Head>
  <Args>
    <Arg Value="x:agent" />
    <Arg Value="y:physobj" />
    <Arg Value="z:location" />
  </Args>
  <Body>
    <Subevent Value="grasp(x,y)" />
    <Subevent Value="while(hold(x,y)^!at(y,z)):
     move(x,y,z,'VoxSimPlatform.Pathfinding.AStarSearch.PlanPath',
     loc(y),z,y)" />
    <Subevent Value="if(at(y,z)):ungrasp(x,y)" />
  </Body>
</Type>
```

Beispiel 3.2.2: Aufbau des Verbs „put“

3.3 Attribute

Der Entity-Typ Attribut wird dazu verwendet, um adjektivische Modifikationen an Objekten abzubilden. Oftmals hat ein Attribut eine Menge an Werten, die die Ausprägung des Attributs beschreiben sollen. Je nach Anordnung und Menge dieser Werte lässt sich daraus eine Skala ableiten.

Skalentyp	Attribut	Werte / Ausprägungen
binär	`Härte`	`weich, hart`
nominal	`Farbe`	`grün, gelb, blau`
ordinal	`Größe`	`klein, mittel, groß`
intervall	`Temperatur`	`20°C, … ,100°C`
rational	`Gewicht`	`0Kg, 1Kg, 2Kg, etc.`

Tabelle 3.3.1: Skalentypen und Ausprägungen einiger Attribute

Die Skalentypen lassen sich nach ihrem impliziten Informationsgehalt sortieren. In aufsteigender Reihenfolge kommt als erstes die binäre Skala, welche lediglich zwei Ausprägungen für ein Attribut umfasst. Die nächste Stufe bilden die Nominal- und Ordinalskalen. Beide können beliebig viele Ausprägungen mit beliebiger Distanz aufweisen. Der entscheidende Unterschied ist jedoch, dass die Ausprägungen der Ordinalskala eine Reihenfolge haben. Die höchste Stufe bilden Intervall- und Rationalskalen, deren Gemeinsamkeit es ist, dass ihre Ausprägungen jeweils in gleich großen, regelmäßigen Abständen verteilt sind. Der Unterschied besteht darin, dass die Rationalskala zusätzlich einen natürlichen Nullpunkt beinhaltet.

Betrachtet man Beispielsweise das Attribut Farbe in der Ausprägung „`brown`“, so fällt auf, dass zur Attributmodellierung nicht viel Code benötigt wird. Nach den Lexikalischen Informationen folgt direkt das Element „`Type`“, welches Anzahl und Typ der Argumente, den Skalentyp des Attributs und die Angabe über die Notwendigkeit eines Referenzobjektes beinhaltet.

```
<Type>
  <Args>
    <Arg Value="x:physobj" />
  </Args>
  <Scale>nominal</Scale>
  <Arity>intransitive</Arity>
</Type>
```

Tabelle 3.3.2: Element „Type“ des Attributs „brown“

3.4 Relationen

Relationen beziehen sich meist auf zwei Objekte oder ein Objekt und einen Agenten. Sie dienen dazu, Eigenschaften ihrer Argumente, wie zum Beispiel deren Position in ein Verhältnis zu setzen. Der Inhalt des Elements „`Class`“ teilt die Relationen in zwei Gruppen: „`force_dynamic`“ und „`config`“. Zur Klasse „`force_dynamic`“ gehören Relationen, die in irgendeiner Weise mit Dynamik oder Krafteinwirkungen einhergehen, während zu „`config`“ statische Anordnungen gehören

Die Relation „left“ nimmt Beispielsweise zwei Objekte als Argumente entgegen und überprüft deren Koordinaten auf der X-Achse. Liegt ein Apfel links neben dem Teller, so ist die X-Koordinate des Apfels kleiner als die des Tellers. Da es bei dieser Relation um die statische, räumliche Anordnung zweier Gegenstände geht, gehört sie der Klasse „`config`“ an.

```
<Type>
  <Class>config</Class>
  <Value>RCC8.EC</Value>
  <Args>
      <Arg Value="x:physobj" />
      <Arg Value="y:physobj" />
  </Args>
  <Constr>X(x) &lt; X(y)</Constr>
  <Corresps />
</Type>
```

Tabelle 3.4.1: Element „Type“ der Relation „left“

3.5 Funktionen

Funktionen sind notwendig für das Zusammenspiel von Objekten und Programmen. Sie erledigen kleinere Aufgaben, die selbst kein Programm darstellen, ohne deren Erledigung aber keine Simulation möglich wäre. Ein Beispiel hierfür wäre die Funktion „the“, die nichts weiter tut, als das Objekt, auf das sie sich bezieht ohne Artikel wiederzugeben.

Zur Veranschaulichung der Umsetzung einer etwas komplexeren Funktion betrachten wir nun die Funktion „top“, welche beispielsweise Für die Ausrichtung von Objekten benötigt wird. Funktionen haben in VoxML außer dem Element „`Lex`“, welches lediglich das Prädikat der Funktion speichert nur noch das Element „`Type`“. Das Element „`Type`“ beinhaltet Informationen über „`Arg`“, das Argument bzw. Objekt, welches die Funktion entgegennimmt, wobei über das optionale Element „`Referent`“ noch spezifiziert werden kann, welcher Teil des Objektes angesprochen wird. Das „`Mapping`“ gibt an, mit welchen Dimensionen die Funktion arbeitet. Im Falle der „top“ Funktion wird die Dimension immer um eins verringert: Die Oberseite eines Würfels ist eine Fläche, die einer Fläche ist eine Linie, und die einer Linie ein Punkt. Das letzte, optionale Element „`Orientation`“ gibt Auskunft darüber, ob die Funktion in Welt- oder Objektkoordinaten arbeitet, welche Achse des entsprechenden Koordinatensystems verwendet wird, und ob die Funktion ein Referenzobjekt benötigt.

```
<Type>
  <Args>
    <Arg Value="x:physobj" />
  </Args>
  <Referent>x-&gt;Head</Referent>
  <Mapping>n:n-1</Mapping>
  <Orientation>
    <Space>world</Space>
    <Axis>+Y</Axis>
    <Arity>
       x-&gt;Habitat-&gt;Intrinsic[top(axis)]:intransitive
    </Arity>
  </Orientation>
</Type>
```

Beispiel 3.5.1: „Type“ Element der Funktion „top“

4. Zusammenfassung

In dieser Dokumentation wurde auf das Problem der Umwandlung natürlichsprachiger Statements in bewegte 3D Szenen eingegangen und im Zuge dessen eine Möglichkeit präsentiert, einen Teil des Problems, nämlich die strukturierte Speicherung semantischer Informationen, zu Lösen. Durch die modulare XML-Grundlage ist außerdem garantiert, dass VoxML Dokumente auf lange Zeit robust, verwendbar und erweiterbar bleiben. Weitere Informationen und eine VoxML Simulationsumgebung „VoxSim“ befinden sich auf der Webseite http://www.voxicon.net/.

5.Quellen

VoxSim Dokumentation:
https://www.aclweb.org/anthology/C16-2012.pdf

Zusatzinformationen VoxML und VoxSim Download:
http://www.voxicon.net/

VoxML Spezifikation:
https://arxiv.org/pdf/1610.01508.pdf

Sprachmodelle „Verb-framed“ und „Satellite-framed“:
https://www.wikiwand.com/de/Verb-framed_Language

Begriffsklärung Affordance:
https://www.wikiwand.com/en/Affordance

XML Grundlagen:
https://entwickler.de/online/korrektes-xml-115728.html

http://wwwlgis.informatik.uni-kl.de/archiv/wwwdvs.informatik.uni-kl.de/courses/seminar/WS0203/ausarbeitung1.pdf