Kapitel 4

Modellierung von Simulationen mit Autonomen Objekten

4.1 Einleitung

Nachdem im vorhergehenden Kapitel das Rahmensystem für die Simulation dynamischer Virtueller Welten konstruiert wurde und die Kernkonzepte *Autonomes Objekt* und *Feature* eingeführt wurden, soll nun dargestellt werden wie diese dazu verwendet werden können, um dynamisches Verhalten zu modellieren. Die hier präsentierten Features bilden ein hierarchisches baukastenartiges System welches in Abbildung 4.1 dargestellt ist. Die Stärke des Konzeptes liegt in der Möglichkeit komplexe Funktionalität durch Kombination von einfacheren Features zu realisieren. Im diesem Fall enstehen sogenannte virtuelle Features (in der Abbildung gestrichelt umrandet), welche keine eigene Implementierung benötigen, sondern sich aus mehreren Features zusammensetzen. So entsteht aus der bloßen Kombination des Features (*Visible* mit dem Feature *Controllable* ein neues (virtuelles) Feature *Grabbable* welches Autonome Objekte kennzeichnet, die vom Benutzer greifbar sind.

Andere Features implementieren eigene Funktionalität, greifen dabei aber auf andere, einfachere Features zurück, sodaß der zusätzliche Aufwand klein bleibt.

Die Pfeile zwischen den Features stellen eine Erweiterungsbeziehung dar, die gestrichelten Linen definieren Kommunikation zwischen Features.

Die weiter unten in der Abbildung dargestellten Features realisieren Grundkonzepte für die Realisierung von Beziehungen Autonomer Objekte untereinander, wie gegenseitige Kontrolle (*Controller Controllable* und und Verteilung von Nachrichten *Subscribable*.

Auf der nächst höheren Ebene werden Grafische Objekte und Logische

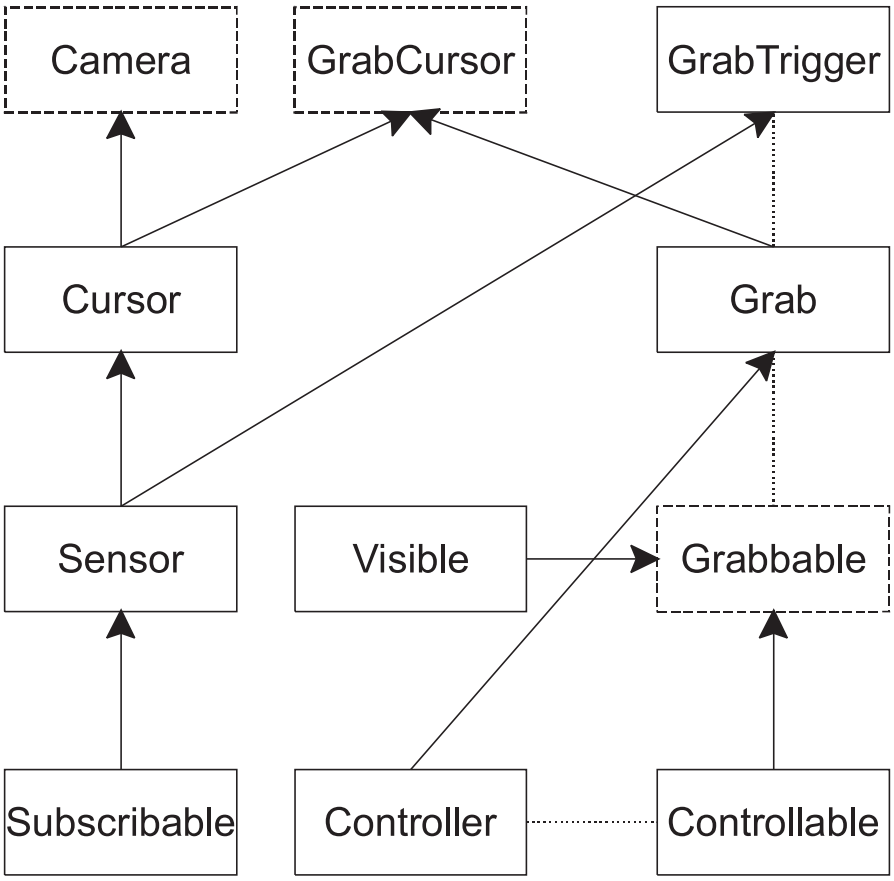

Abbildung 4.1: System von Features zur Modellierung von Verhalten

Geräte ((*Visible Sensor*) integriert.

Darauf aufbauend folgen Features zur Interaktion und Darstellung des Benutzers innerhalb der virtuellen Umgebung.

Der nächste Abschnitt führt zunächst eine Notation ein, in der Feature-Klassen definiert und grafisch dargestellt werden können. Diese Notation wird benutzt um in den darauf folgenden Abschnitten zunächst die Basis-Features einzuführen und das System anschließend auf logische Geräte, grafische Objekte und Interaktionsmetaphern zu erweitern. Schließlich wird wird ein Benutzermodell präsentiert, welches für die Interaktion in Virtuellen Umgebungen geeignet ist.

4.2 Eine Notation zur Darstellung von Feature-Klassen

Um eine Grundlage für die Darstellung von Feature-Klassen zu schaffen wird eine grafische Notation eingeführt. Diese Notation beschreibt alle Elemente der Features, wie sie im letzten Kapitel definiert wurden:

1. Signatur
2. Abhängigkeiten von lokalen und externen Features
3. Attribute
4. Protokolle

Die Notation kann nicht nur – wie im Rahmen der vorliegenden Arbeit – zur Dokumentation dienen, sondern eignet sich auch für die Darstellung in einem Autorensystem für Virtuelle Umgebungen.

Um nicht mit einem nichtssagenden Beispiel zu beginnen, beschreiben wir die Notation anhand zweier Features die tatsächlich existieren nämlich *Controller* und *Controllable*, welche in Abbildung 4.3 auf Seite 68 dargestellt sind. Die ausführliche Erklärung dieser Features erfolgt in Abschnitt 4.3.1.

Die Titelzeile des Diagramms enthält den symbolischen Namen der Feature-Klasse. Unter diesem Namen ist die Klasse dem System bekannt und kann in anderen Diagrammen referenziert werden.

Die rechte Seite des Diagramms definiert das Protokoll des Features, die linke Seite beschreibt alle übrigen Komponenten die zur Definition notwendig sind.

4.2.1 Abhängigkeit, Signatur und Attribute

Die folgenden Elemente zur Beschreibung einer Feature-Klasse werden im Diagramm auf der linken Seite dargestellt:

Using Aufzählung aller weiteren Features, die dem Autonomen Objekt zugeordnet sein müssen. Die Abhängigkeit ergibt sich aus dem gemeinsamen Gebrauch von Attributen.

Linked Aufzählung von Features, bei den Autonomen Objekten vorhanden sein müssen, mit denen kommuniziert wird. Im Falle des Features *Controlling* wird auf der Gegenseite die Feature *Controllable* erwartet.

Attributes Lokale Daten, die das Feature-Objekt instantiiert. Diese können von anderen Features des selben Autonomen Objektes gelesen werden, wenn dies nicht durch eine Umfassung mit eckigen Klammern ausgeschlossen wird.

Signature Die Signatur des Features also die Menge aller Nachrichten, die dieses Feature verarbeiten kann. Dabei werden Quittungen und Fehlerquittungen nicht explizit genannt, diese kann jedes Feature verarbeiten (siehe dazu Abschnitt 3.9.4).

Sends Nachrichten die das Feature seinerseits senden kann. Quittungen (ACK und NACK) werden nicht explizit aufgeführt, da sie von allen Features gesendet werden können.

4.2.2 Protokoll

Die Rechte Seite des Diagramms definiert das Kommunikationsprotokoll als Zustandsübergangsdiagramm. Die Darstellung lehnt sich an die UML-Notation an, weicht aber dort von ihr ab, wo dies zur Straffung der Darstellung dient.

Zustände sind als Ovale dargestellt. Sie enthalten den (unterstrichenen) symbolischen Namen des Zustandes und die auszuführenden Anweisungen in Form von Pseudocode. Einzelheiten zum Pseudocode werden im nächsten Abschnitt diskutiert.

Zwei Arten von Zuständen sind besonders markiert: Der *Startzustand* (dicke Umrandung) und die *Endzustände* (doppelte Umrandung).

Transitionen (Zustandsübergänge) sind als Pfeile gezeichnet. Neben dem Pfeil findet sich die Nachricht, bei deren Empfang der Übergang gewählt wird. Ein Pfeil ohne Nachricht stellt eine Transition dar, die augenblicklich und nicht als Reaktion auf den Empfang einer Nachricht, ausgeführt wird.

Eine von der UML abweichende Notation wird für die Verarbeitung von Quittungen (ACK und NACK, sie Abschnitt 3.9.4) gewählt, um die Darstellung zu vereinfachen. In vielen Fällen muß bei Eintreffen einer Fehlerquittung der Sender der dafür ursächlichen Nachricht informiert werden. Dies in jedem Fall explizit hinzuschreiben, würde die Darstellung unnötig verkomplizieren.

Die Definition für NACK gibt die folgende Abbildung 4.2, für ACK gilt das Gleiche.

4.2.3 Pseudocode

Für die Anweisungen die in den Zustandsdiagrammen dargestellt werden wird eine Notation ähnlich der Skriptsprache Python [Beaz01]) verwendet.

Abbildung 4.2: Erweiterung des UML Diagramms

Die Sprache ist gut lesbar (auch wenn man sie nicht beherrscht) und bietet mächtige Konzepte wie Mengen oder Listen. Wo immer nicht-triviale Konstrukte verwendet werden, werden sie im Text erklärt. Typen werden, um die Darstellung nicht aufzublasen, weggelassen, da sie sich ohnehin meist aus dem Kontext oder aus den Variablennamen ergeben.

Nachrichten werden durch Aufruf einer Systemfunktion `send` abgeschickt. Die Signatur von `send` lautet:

```
send(receiver=self,Message)
```

Der Parameter `receiver` enthält die Adresse des Autonomen Objektes, an welches die Nachricht gerichtet ist. Wird der Empfänger weggelassen, sendet das AO an sich selbst. Die Kurzform `answer(Message)` entspricht `send(sender, Message)`. Hierbei ist `sender` der Absender der zuletzt empfangenen Nachricht.

4.3 Grundlegende Konzepte

4.3.1 Kontrolle

In vielen Fällen üben Objekte in Simulationen Kontrolle aufeinander aus. Ein Objekt ist dabei der Kontrolleur, das andere Objekt wird kontrolliert. Dabei ist wichtig daß das kontrollierte Objekt nur höchstens einen Kontrolleur zuläßt. Dadurch werden Inkonsistenzen in der Simulation vermieden. Solche Beziehungen treten bei der Animation und bei der Benutzerinteraktion auf. So kann beispielsweise ein Objekt immer nur von einem Benutzer gegriffen und bewegt werden.

Realisiert wird diese Beziehung durch die Features *Controller* und *Controllable* die in Abbildung 4.3 dargestellt sind.

Das links dargestellte Feature *Controller* wird einem Autonomen Objekt assoziiert, welches ein anderes Objekt kontrollieren kann. Dies kann beispielsweise das grafische Echo eines Interaktionsgerätes sein, mit dem der Benutzer Objekte in der Szene greifen und bewegen kann. Letztere besäßen ein Feature *Controllable*.

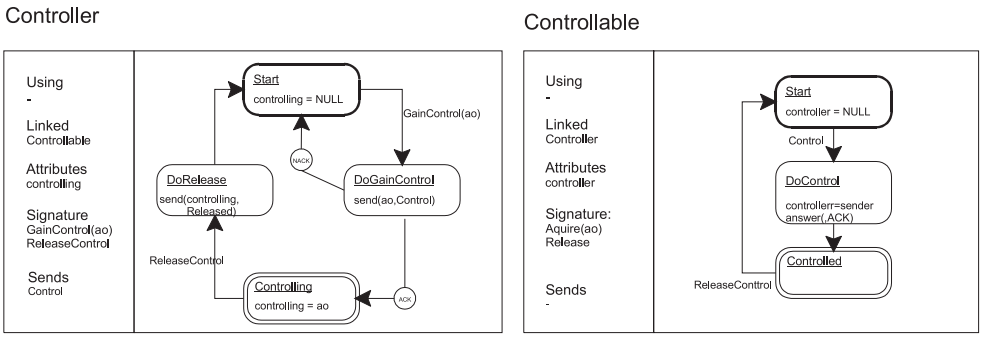

Abbildung 4.3: Features *Controller* und *Controllable*

Controller besitzt ein Attribut `controlling` welches die ID des kontrollierten AO enthält und ist zu begin leer (gleich `NULL`). *Controller* wird durch die Nachricht `GainControl(ao) ao` dazu veranlasst das Autonome Objekt `ao` von jetzt an zu kontrollieren und sendet diesem die Nachricht `Control`. Im Erfolgsfall (ACK) geht der Automat nun in den Zustand `Controlling` und legt `ao` im Attribut `controlling` ab. Wird andernfalls ein NACK empfangen, war `ao` entweder nicht kontrollierbar (hatte nicht das Feature *Controllable* oder wurde bereits durch ein anderes AO kontrolliert. Das NACK wird an den Sender der ursprünglichen Nachricht weitergeleitet. Durch die Nachricht `ReleaseControl` wird die Freigabe signalisiert worauf der Automat in den Startzustand übergeht.

Selbstverständlich wäre es möglich, *Controller* so zu gestalten, daß beliebig viele Objekte gleichzeitig kontrolliert werden können, aber dies wird für die weiteren Ausführungen nicht benötigt. Daher wurde der einfacheren Variante der Vorzug gegeben.

Controllable ist das Gegenstück zu *Controller* und kennzeichnet ein AO, welches exklusiv kontrolliert werden kann. Es ist höchstens ein Kontrolleur möglich. Dieser wird bei Empfang der Nachricht `Control` im Attribut `controller` registriert und der Automat geht in den über den Zustand `DoControl` in den Zustand `Controlled` über. Bei Empfang der Nachricht `ReleaseControl` wird wieder der Startzustand eingenommen und `controller` wieder gelöscht.

4.3.2 Verteilung von Daten: Abonnieren

In vielen Fällen werden Daten von einem Autonomen Objekt zu einer Menge weiterer Autonomer Objekte versendet. Etwa sollen mit einem Interaktionsgerät durch den Benutzer gleichzeitig mehrere grafische Objekte gesteuert werden. Diese Beziehungen können vor Beginn der Laufzeit feststehen, oder

erst später entstehen. Für beide Fälle kann das Feature *Subscribable* verwendet werden.

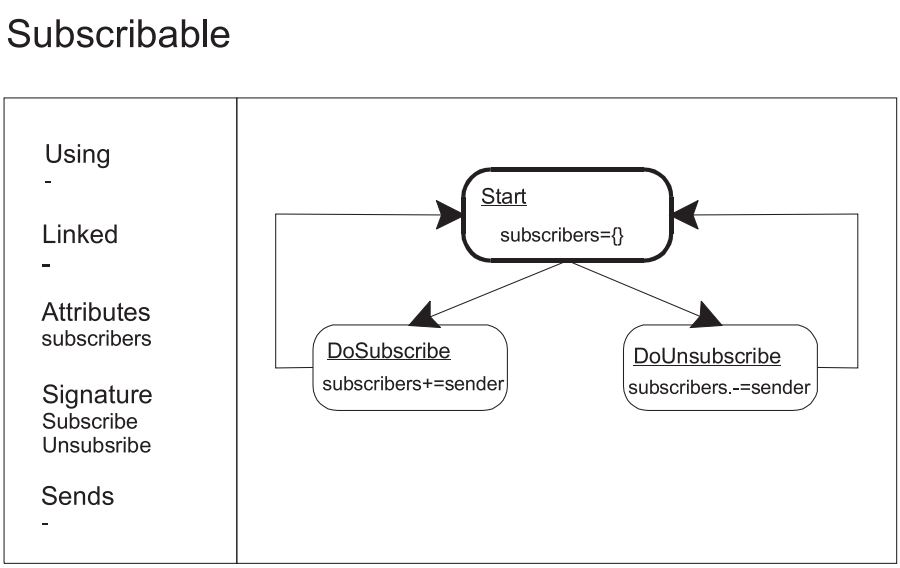

Abbildung 4.4: Feature Subscribable

Subscribable erweitert sein Autonomes Objekt um ein Attribut `Subscriber`, welches eine Liste der Autonomen Objekte enthält, die informiert werden sollen.

Subscribable stellt diese Liste anderen Features zur Verfügung und sorgt selbst nicht für die Verteilung der Daten.

Über die Nachrichten `Subscribe` und `Unsubscribe` können sich Autonome Objekte in die Liste ein und aus ihr austragen.

4.3.3 Grafische Objekte

Das Feature *Visible* wird Autonomen Objekten zugeordnet, die sichtbar sind, also die einem grafischen Objekt zugeordnet sind und dies kontrollieren. Die Darstellung ist nicht abhängig von einem bestimmten Renderer, sondern legt ein generisches Interface zugrunde, mit dem Attribute abgefragt `getAttr()` und gesetzt (`setAttr()`) werden können. Für die weiteren Ausführungen ist nur das Attribut `transform` wichtig. `transform` stellt die Transformationsmatrix eines grafischen Objektes (oder eines Teils des Szenengraphen) dar, mit deren Hilfe u.A. Position und Orientierung verändert werden können.

Mit Hilfe der Nachrichten `GetVisAttr` und `SetVisAttr` können andere Autonome Objekte die Attribute des grafischen Objektes setzen oder lesen.

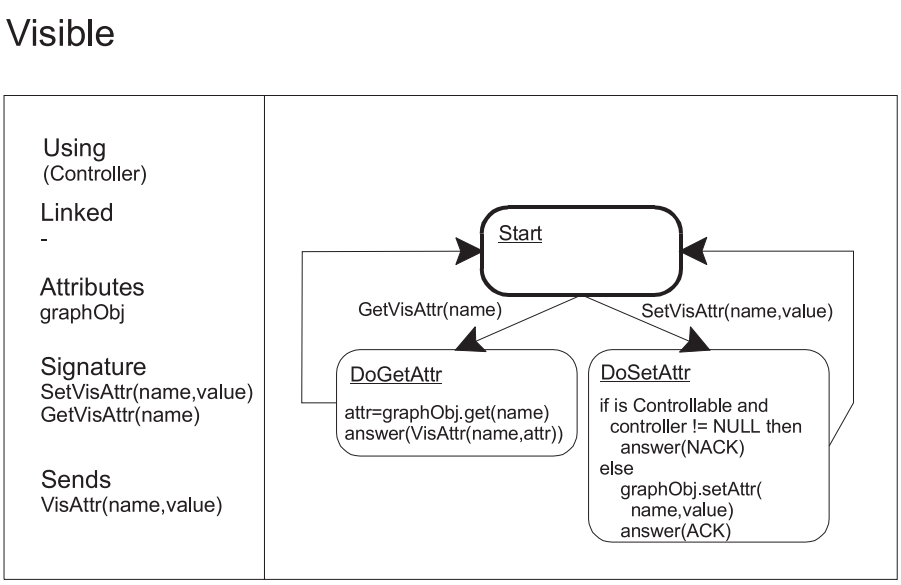

Abbildung 4.5: Feature *Visible*

Letzteres geschieht durch die Antwort mit der Nachricht `VisAttr` die *Appearance* als Antwort auf `GetVisAttr` sendet.

Wichtig ist der Zusammenhang mit dem Feature *Controllable*. Im Zustand `SetAttr` wird zunächst geprüft, ob das AO das Attribut *Controllable* besitzt und bereits ein Kontrolleur registriert wurde. In diesem Fall wird das Setzen des Attributes durch ein anderes AO verweigert und mit NACK beantwortet.

Wird *Appearance* ohne *Controllable* eingesetzt, kann jedes AO ungehindert die Attribute des grafischen Objektes setzen und schreiben. Durch die einfache Kombination mit *Controllable* wird exklusiver Zugriff realisiert.

4.3.4 Kollisionserkennung

Kollisionserkennung wird sowohl für die Benutzerinteraktion, als auch für die physikalisch-basierte Simulation benötigt. Da die Erkennung von Kollisionen zwischen komplexen polygonalen Objekten extrem rechenaufwendig ist, wird diese meist nicht auf der gesamten Szene, sondern nur zwischen Paaren entsprechend ausgezeichneten grafischen Objekten durchgeführt. Diese Auszeichnung wird durch Zuordnung des Features *Collidable* durchgeführt.

Bei der Kollisionserkennung werden immer Paare grafischer Objekte gegeneinander getestet. Typischerweise werden neben dem binären Ergebnis (Kollision liegt vor oder nicht) weitere Daten, wie etwa die Positionen der beteiligten Polygone etc. zurückgeliefert. Der Einfachheit halber beschränken

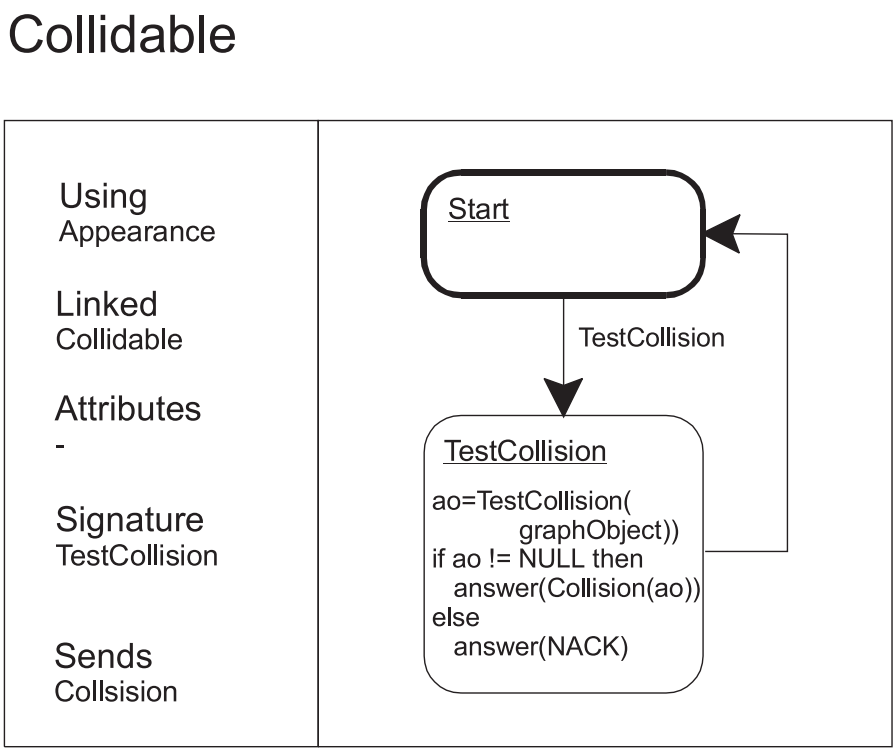

Abbildung 4.6: Feature *Collidable* zur Kollisionserkennung

wir uns hier auf das bloße Ereignis und ignorieren weitere Daten. Sie können für die praktische Anwendung leicht ergänzt werden, ohne das Vorgehen an sich zu ändern.

Ein AO mit dem feature *Collidable* ist in der Lage, Kollisionen des assoziierten grafischen Objektes und dem eines anderen AO zu testen, wenn es die Nachricht `TestColl(ao)` empfängt. Das zweite AO muß dieses Feature ebenfalls besitzen.

Die Funktion `TestCollision(ao)`, die den Kollisionstest durchführt, verschafft sich zunächst eine Liste aller Autonomen Objekte mit dem Feature *Collidable* über eine Anfrage and die WDB (siehe Abschnitt 3.12). Stellt die WDB effiziente Methoden zur räumlichen Suche zur Verfügung, kann der Suchraum auf das Hüllvolumen des dem anfragenden AO assoziierten grafischen Objektes eingeschränkt werden. Falls die zugrunde liegende Kollisionserkennung dies nicht schon ohnehin leistet, kann hier ein enormer Effizienzgewinn erzielt werden.

Wurde eine Kollision erkannt, wird dies dem Sender mit der Nachricht `Collision` mitgeteilt, andernfalls wird mit NACK geantwortet.

4.4 Benutzerinteraktion

Nachdem die grundlegenden Konzepte für die Modellierung von Verhalten mit Autonomen Objekten eingeführt wurden, werden nun weitere Autonome Objekte präsentiert, die es erlauben ein Benutzermodell baukastenartig zusammenzusetzen. Dazu werden Features zur Anbindung von Interaktionsgeräten, Steuerung von Cursoren sowie für Greif-Aktionen entwickelt.

4.4.1 Anbindung der Interaktionsgeräte

Zur Integration multidimensionaler Interaktionsgeräte (siehe Kapitel 5) dient das Feature *Sensor*. *Sensor* bildet die Schnittstelle zu dem System IDEAL , welches eine Klassenhierarchie logischer Geräte (siehe Abschnitt 5.3) realisiert. Logische Geräte unterscheiden sich im Typ der gelieferten Daten (z.B. Status einer Taste oder Position und Orientierung eines Trackers). Es muß aber nicht für jede Klasse ein eigenes Feature realisiert werden, da die Protokolle zur Bereitstellung der Gerätedaten identisch sind. Nur der Typ der Gerätedaten ist unterschiedlich.

Wie Abbildung 4.7 darstellt, wird im Startzustand das logische Gerät zunächst durch den Aufruf von `InitDevice` initialisiert. Der Parameter `name` bezeichnet den symbolischen Namen des Gerätes, der in der Sensor Datei von IDEAL definiert ist (siehe Abschnitt 5.4.5). Da IDEAL Fehler, die durch falsche Konfiguration oder durch ein nicht funktionierendes Gerät ausgelöst werden, können selbst behandelt, brauchen diese auf der Ebene der Simulation nicht berücksichtigt zu werden.

Nachdem das Gerät erfolgreich initialisiert wurde, werden nun im Zustand `PollDevice` die Daten kontinuierlich gelesen und über die Nachricht `SensorData` verteilt. Dabei wird auf das Feature *Subscribable* (siehe Abschnitt 4.3.2) zurückgegriffen. Alle Autonomen Objekte, die Sensordaten erhalten wollen, melden sich zuvor per `Subscribe` bei dem AO, welches den Sensor verwaltet an.

Autonome Objekte, die nur gelegentlich Daten abfragen wollen und nicht an einem kontinuierlichen Datenstrom interessiert sind können die Abfrage durch die Nachricht `RequestData` durchführen, welche im Zustand `DoRequestData` verarbeitet wird.

4.4.2 Cursor

Cursor kombiniert eine Reihe von Features, die bereits vorgestellt wurden. Es sind dies *Visible* und *Sensor*, welches seinerseits *Subscribable* verwendet.

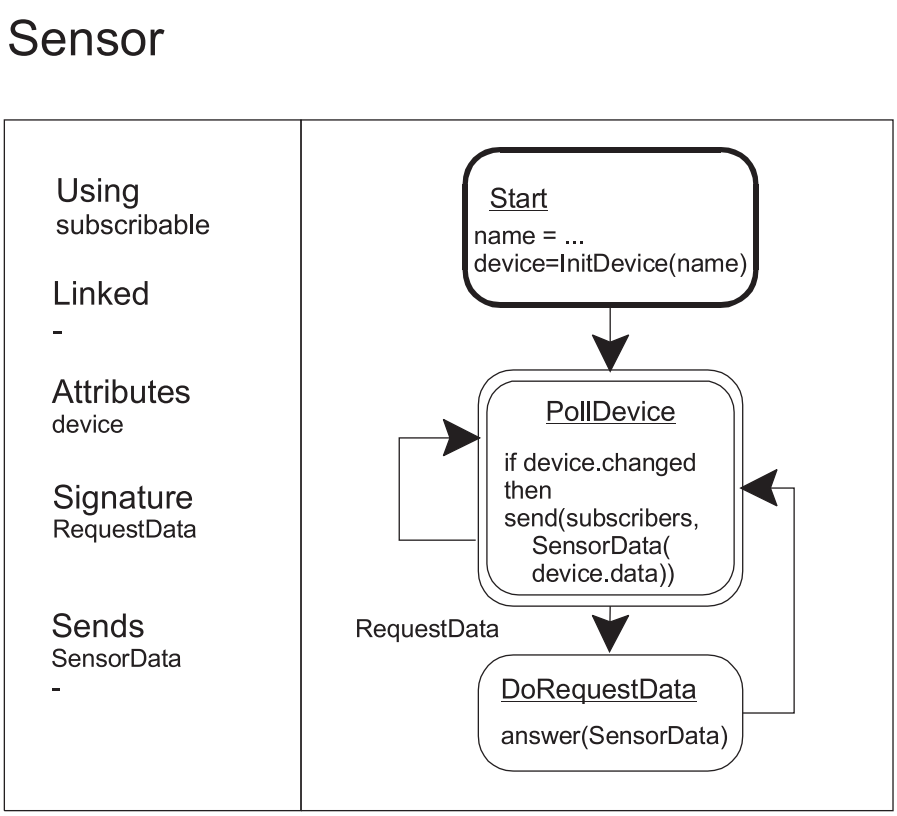

Abbildung 4.7: Feature zur Anbindung von Interaktionsgeräten

Hier kann die Orthogonalität des Konzeptes sehr schön demonstriert werden. Ein Cursor ist ein sichtbares (grafisches) Objekt, welches von einem Interaktionsgerät gesteuert wird.

Die Vorgehensweise ist, wie Abbildung 4.8 darstellt, recht simpel. Zunächst werden die eigenen Gerätedaten abonniert. Sobald diese eintreffen, wird das grafische Objekt bewegt.

Dies geschieht, indem die eintreffenden Gerätedaten `SensorData` ausgelesen werden und in Form einer NachrichtSetVisAttr gleich wieder an sich selbst geschickt werden. Das Feature *Visible*, wird daraufhin die Transformation (identifiziert durch den ersten Parameter der Nachricht) des grafischen Objektes setzen und den Cursor so entsprechend bewegen.

4.4.3 Greifen

Das Feature *GrabTrigger* erweitert die Funktion eines (Bool'schen) Sensors, also eines Buttons (siehe Abschnitt 5.4.4 auf Seite 93. Der Zustand des lo-

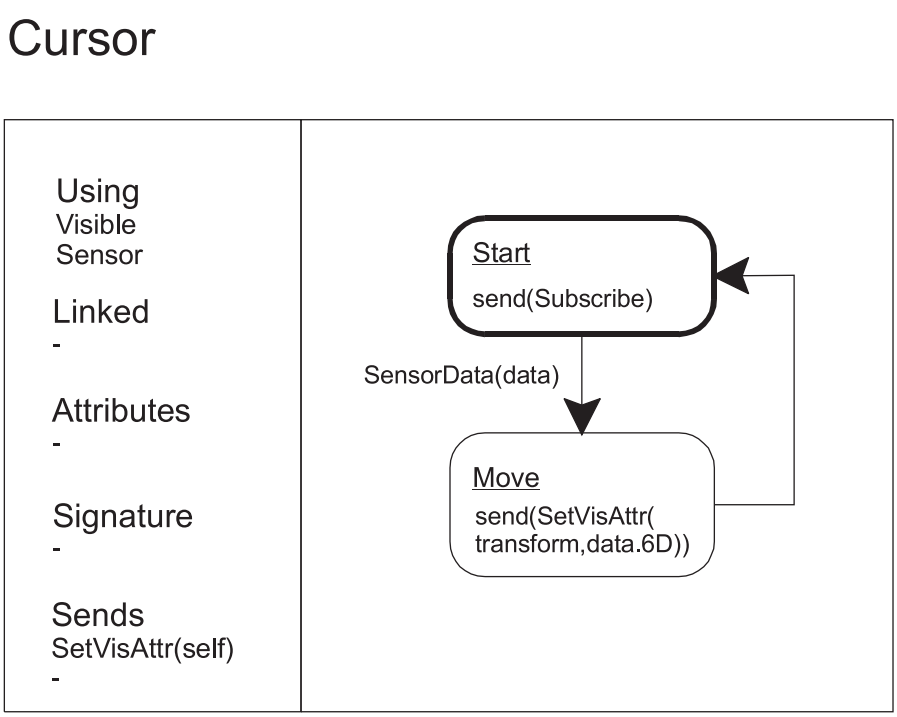

Abbildung 4.8: *Cursor* Feature

gischen Gerätes wird abgefragt und in Nachrichten für ein Grabber Feature umgewandelt (`Grab` und `Release`).

Dank dem mächtigen Konzept der logischen Geräte, welches IDEAL realisiert, kann *GrabTrigger* sehr einfach gehalten werden, ist aber dennoch sehr vielseitig einsetzbar. Das Feature funktioniert mit jeder Art von Eingabegeräten, neben Tasten von Keyboard und Maus kann auch die Gestenerkennung verwendet werden. Auf diese Weise können *GrabTrigger* Greifen über die Gestenerkennung in Verbindung mit einem Datenhandschuh realisiert werden.

4.5 Benutzer Modell

Aufbauend auf den in den letzten Abschnitten entwickelten Konzepten wird nun ein allgemeines Modell zur Darstellung eines Benutzers einer Virtuellen Umgebung angegeben. Einen Überblick gibt Abbildung 4.10. Das Modell beschreibt einen typischen Benutzer[1], ausgestattet mit

[1]Die Interaktionsgeräte werden in Abschnitt 5.2.1 eingeführt und beschrieben.

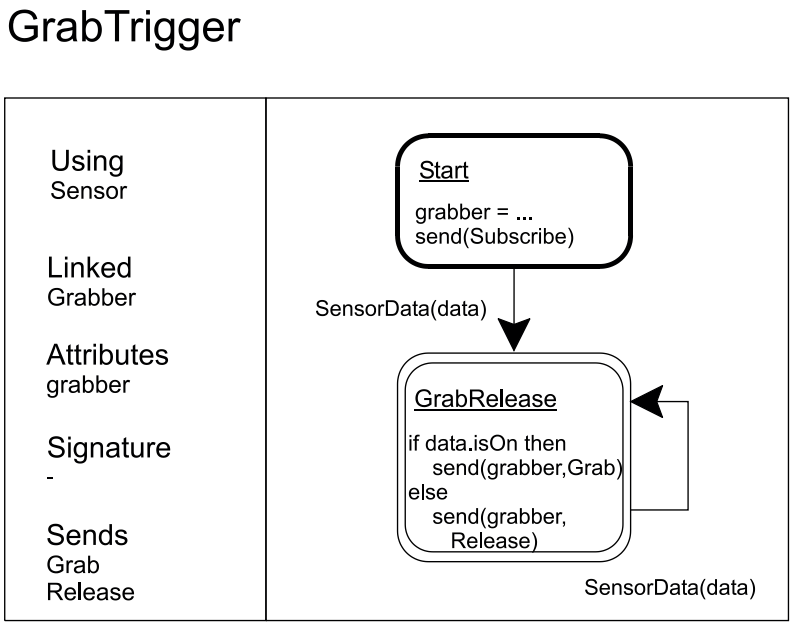

Abbildung 4.9: Feature GrabTrigger

- Headtracker zur Registrierung der Kopfbewegungen
- Datenhandschuh zur Steuerung des Cursors (sog. Handecho) und zum Greifen von Objekten
- Spacemouse zur Navigation.

Dieses Modell eignet sich zur Beschreibung einer Vielzahl von immersiven Anwendungen der Virtuellen Realität, insbesondere in der CAVE, aber auch für Installationen, in der eine Monitorbrille getragen wird.

Das Diagramm stellt das eigentliche Benutzermodell umfaßt von einem gestrichelten Rahmen dar. Zusätzlich ist ein gegriffenes Objekt zu sehen, welches sich außerhalb des Rahmens befindet und aus einem AO und einem grafischen Objekt besteht.

Das Benutzermodell ist von links nach rechts geordnet. Es sind in dieser Reihenfolge dargestellt:

1. Physische Geräte
2. Logische Geräte

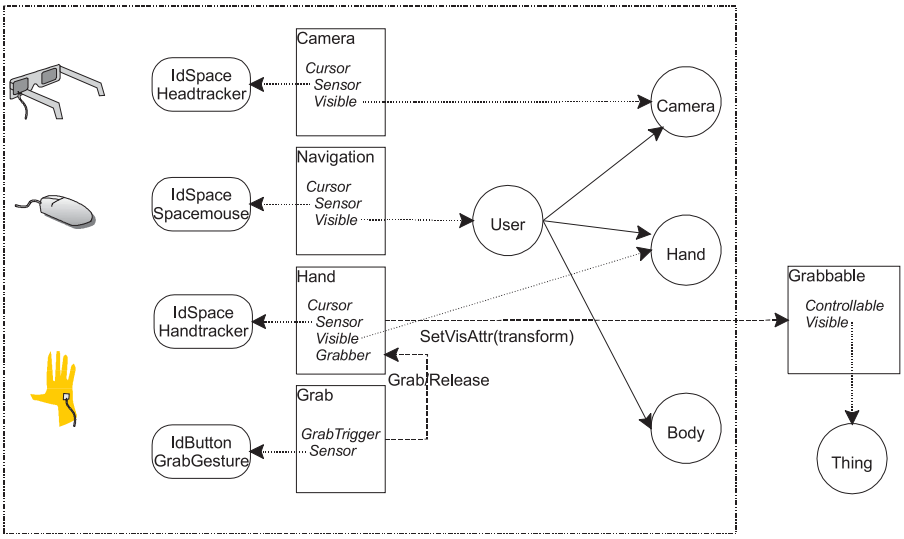

Abbildung 4.10: Modell des Benutzers

3. Autonome Objekte

4. Szenengraph

Die logischen Geräte abstrahieren zunächst die gerätespezifischen Eigenschaften und liefern logische Gerätedaten in einem standardisieren Format[2].

Die logischen Geräte werden von Autonomen Objekten kontrolliert, die wiederum in den Szenengraphen eingreifen, um die Interaktion zu realisieren. Für jede vorgesehene Form der Interaktion existiert ein Autonomes Objekt, welches diese implementiert. Da die Feature Klassen bereits in den letzten Abschnitten definiert wurden, genügt es an diese Stelle den Zusammenhang zwischen Autonomen Objekten und Interaktionstechniken darzustellen. Es werden die Namen der Autonomen Objekte in Abbildung 4.10 verwendet.

Camera Verarbeitet die logischen Gerätedaten des Headtrackers und steuert den Kamera Knoten. Kopfbewegungen des Benutzers wirken sich direkt auf die Perspektive der grafischen Darstellung der Szene aus, wie dies in immersiven Anwendungen üblich ist.

Navigation Der Benutzer steuert seine Position und Orientierung innerhalb der Szene mit der Spacemouse. Die Gerätedaten werden in die Transformation des Gruppenknoten *User* geschrieben. Da alle Kind-Knoten

[2]Siehe Abschnitte 5.3 und 5.4.4.

ihre Transformation von *User* erben, wird so die ganze Repräsentierung des Benutzers, inklusive der Darstellung des Körpers (*Body*), der Kamera und der Hand mitbewegt.

Hand Dieses Autonome Objekt kontrolliert den Tracker, der auf dem Datenhandschuh montiert ist und steuert damit den Cursor-Knoten innerhalb der Szene. Durch Handbewegungen kann dieser bewegt werden. Zusätzlich wurde dem Cursor die Feature *Grab* zugeordnet, um in Verbindung mit dem AO *Grab* das Greifen von Objekten zu ermöglichen.

Grab Realisiert das Greifen von Objekten, sobald eine Greifgeste am Datenhandschuh erkannt wird. Die Gestenerkennung übernimmt dabei ein logischer Button *GrabGesture*. Das Autonome Objekt setzt die logischen Gerätedaten des Buttons in Nachrichten an das *Cursor*-AO um (Grab/Release).

Auf der rechten Seite ist ein greifbares Objekt dargestellt, bestehend aus einem AO welches diese Funktion realisiert und einem grafischen Objekt zu dessen Darstellung. Das AO besitzt zwei Features, *Visible* und *Controllable* welches, wie bereits diskutiert genügt, um ein Objekt für den Benutzer greifbar zu machen.