Kapitel 3

Ein Simulationssystem für Autonome Objekte

3.1 Einleitung

Nachdem im vorherigen Kapitel der Stand der Wissenschaft und Technik dargestellt und diskutiert wurde, wird nun ein System eingeführt, welches als Komponente eines VR-Systems der dritten Generation zur Simulation von dynamischem Objektverhalten dient.

Das System trennt die Simulation von dynamischem Objektverhalten vollständig von der grafischen Darstellung der Virtuellen Umgebung. Damit ist die Parallelisierung des Systems mit geringem Aufwand möglich.

Zentrales Konzept des Systems sind sogenannte Autonome Objekte, die die Bausteine der Simulation bilden. Das Verhalten dieser Autonomen Objekte kann zur Laufzeit des Systems durch Eigenschaften definiert und erweitert werden. Dadurch wird der Entwurfsprozess für dynamische Virtuelle Umgebungen vereinfacht und beschleunigt.

Autonome Objekte kommunizieren miteinander über Nachrichten, die zu einem genau definierten Zeitpunkt zwischen den Simulationsschritten propagiert werden. Dies ermöglicht die saubere Modellierung von zeitdiskreten Simulationen.

Um dynamisches Objektverhalten zu realisieren, müssen nicht nur statische sondern dynamische Verbindungen zwischen Autonomen Objekten möglich sein. Damit solche dynamischen Verbindungen zur Laufzeit zustande kommen können, muß nach Autonomen Objekten gesucht werden können,

welche bestimmte logische Eigenschaften besitzen, oder sich an einem bestimmten Ort befinden.

In den folgenden Abschnitten werden zunächst die Konzepte eingeführt und im Zusammenhang erläutert. Anschließend wird genauer auf die technische Realisierung eingegangen. Danach wird das Prozeßmodell und die Verteilung der Simulation auf mehrere Prozesse beschrieben. Schließlich wird die Welt-Datenbasis eingeführt, die das Auffinden von Autonomen Objekten in komplexen Simulationen optimiert. Auf Grundlage der Welt-Datenbasis werden Verfahren zur Identifikation von Hindernissen und zur Suche von kollisionsfreien Pfaden entwickelt.

3.2 Entwurfsprozeß

Der Ablauf der Entwicklung einer Virtuellen Umgebung läuft meist in zwei Schritten ab. Zunächst werden die grafischen Elemente der Szene erstellt. Anschließend wird das Objektverhalten programmiert.

Den Entwurfsprozeß stellt Abbildung 3.1 genauer dar. Bestehende Szenen oder Komponenten können importiert oder neu erstellt werden. Hierbei ist die Struktur der grafischen Szenenbeschreibung von Bedeutung. Grafische Objekte, die Elemente der Simulation darstellen, also Verhalten besitzen, müssen als eigenständige Objekte im Szenengraphen abgebildet sein. Dabei werden dynamische Objekte typischerweise unterhalb der Wurzel des Szenengraphen eingehängt (vergl. Abschnitt 2.3.2). Komplexe Objekte mit Verhalten können aus mehren Teilobjekten bestehen, die zusammen einen Subgraph bilden. Wird die Szene neu modelliert, kann diese Struktur von vornherein erzeugt werden. Wird die Szene dagegen aus externen Quellen importiert (z.B. CAD-Programm), muß die Szenenhierarchie oft erst geeignet strukturiert werden.

Im nächsten Schritt wird das Verhalten der Objekte programmiert und in Testläufen verifiziert.

Für Systeme, die das Verhalten durch Programme darstellen, die in Maschinensprache übersetzt werden, stellt diese Schleife von Testläufen und Korrekturen einen Flaschenhals dar, deren Ablauf die linke Seite von Abbildung 3.2 illustriert. Zwar ist die Dauer eines Compilerlaufes für ein modular aufgebautes Programm, das nicht komplett sondern nur in Teilen übersetzt werden muß, gering (wenige Sekunden). Das VR-System muß aber nach dem Testlauf zunächst beendet werden. Dann wird das Programm modifiziert und neu übersetzt. Schließlich wird das VR-System neu gestartet, wobei die Szenenbeschreibung geladen und Interaktionsgeräte initialisiert werden. Dieser

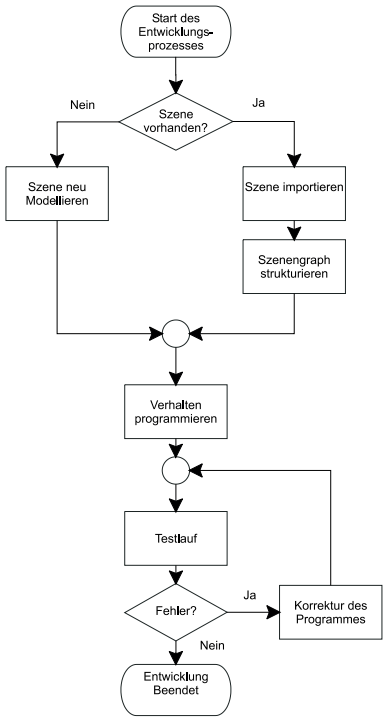

Abbildung 3.1: Ablauf der Entwicklung Virtueller Umgebungen

Vorgang kann für komplexe Systeme mehrere Minuten dauern. Daher ist die Programmierung von Verhalten für solche Systeme ein mühsamer und zeitaufwendiger Prozeß.

Wird das Verhalten in einer Skriptsprache definiert, ist es prinzipiell möglich, das Verhalten zur Laufzeit des Systems zu ändern, indem das Skript modifiziert wird und das VR-System lediglich in den Startzustand zurückversetzt wird, bevor ein neuer Testlauf erfolgt. Allerdings muß hier die wesentlich geringere Performanz – bedingt durch die Interpretation der Skriptsprache zur Laufzeit – in Kauf genommen werden. Dadurch lassen sich skriptbasierte Systeme schlecht zu komplexen Systemen skalieren.

Das hier vorgestellte System zur Simulation von Verhalten durch Autonome Objekte muß nicht nach jeder Fehlerkorrektur neu gestartet werden. Dennoch kann Verhalten in einer Programmiersprache formuliert werden, die in Maschinensprache übersetzt wird, sodaß es den Nachteil der skriptbasierten Systeme, zu langsam für komplexe Systeme zu sein, umgangen wird.

Wie die rechte Seite von Abbildung 3.2 darstellt, wird Verhalten zunächst modular in Form von Eigenschaften, die Objekte der Simulation aufweisen können, programmiert. Anschließend wird das VR-System gestartet. Eigen-

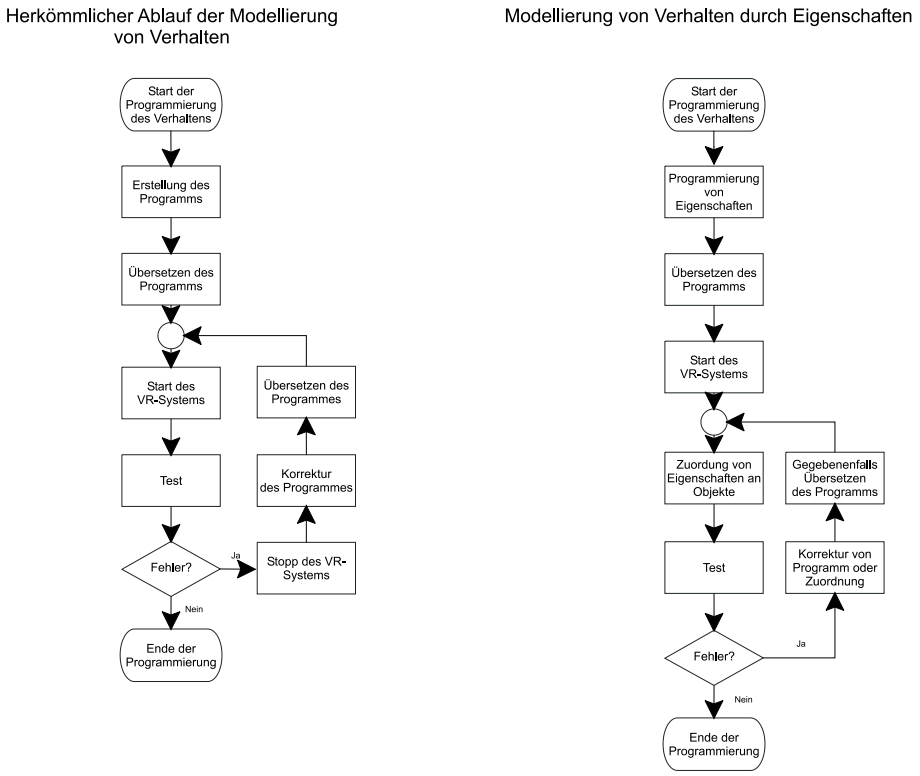

Abbildung 3.2: Ablauf der Entwicklung von Objektverhalten

schaften können nun den zu simulierenden Objekten dynamisch zugeordnet werden, während das VR-System läuft. Treten Fehler auf, können diese durch Änderung der Zuordnung oder durch Korrektur des Programmcodes behoben werden, ohne daß das VR-System neu gestartet werden muß.

3.3 Trennung von grafischer Darstellung und Simulation

Wie im vorangegangen Kapitel bereits ausgeführt wurde, ist die grafische Szenenbeschreibung als Grundlage für die Simulation aus den folgenden Gründen ungeeignet:

1. Die Datenstrukturen des Szenengraphen sind auf die effiziente grafische Darstellung optimiert, Mechanismen zum Auffinden von Objekten anhand logischer Kriterien fehlen.

2. Die Parallelisierung von grafischer Darstellung und Simulation wird erschwert.

3. Das Objektmodell für grafische Objekte ist meist schwer auf die Erfordernisse der Simulation zu erweitern.

4. Das Simulationssystem ist vom verwendeten grafischen Renderer abhängig.

Das im Rahmen dieser Arbeit vorgestellte System trennt daher die Datenstrukturen der grafischen Darstellung und des Simulationssystems. Die grafische Datenbasis wird vom Renderer in Form eines hierarchischen Szenengraphen gehalten.

Die Simulation verwaltet eine Liste von zu simulierenden *Autonomen Objekten*. Das Simulationssystem iteriert diese Liste und erlaubt nacheinander jedem Autonomen Objekt einen Simulationsschritt auszuführen. Autonome Objekte, die eine grafische Repräsentierung besitzen, können grafische Objekte manipulieren.

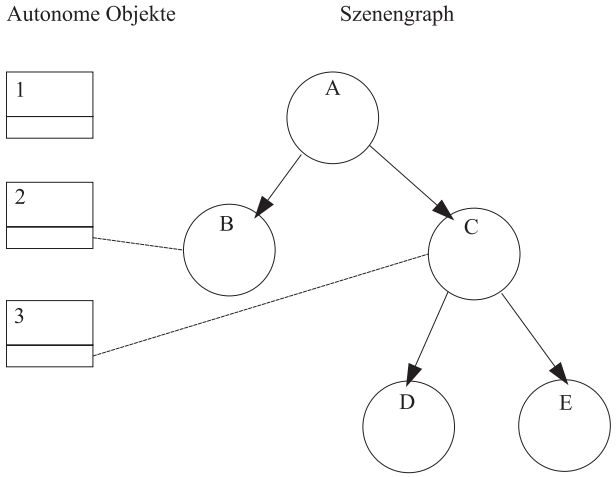

Abbildung 3.3: Relation Autonomer und grafischer Objekte

Die Relation von grafischen und Autonomen Objekten ist eindeutig, d.h. jedes sichtbare Autonome Objekt besitzt ein oder mehrere grafische Objekte *exklusiv*. Dadurch werden unerwünschte Seiteneffekte beim konkurrierenden Zugriff auf grafische Objekte vermieden und die Parallelisierung des Systems erleichtert.

3.4 Autonome Objekte

Aus der Trennung von grafischer Szenenbeschreibung und Simulationsmodell ergibt sich die Notwendigkeit, für die zu simulierenden Objekte ein eigenes Objektmodell einzuführen.

Die Elemente der Simulation werden als *Autonome Objekte* dargestellt. Wie bereits beschrieben, können diesen zur Laufzeit Eigenschaften zugeordnet werden, bei denen es sich wiederum um Objekte handelt.

Dieser Vorgehensweise wurde gegenüber der Modellierung des Verhaltens durch eine Klassenhierarchie bevorzugt. Dies soll im Folgenden begründet werden.

Wird der Quellcode einer Klassenhierarchie geändert, müssen mindestens die betroffenen Klassen neu übersetzt und gebunden werden. Dies ist aber zur Zeit unter den gängigen Betriebssystemen nur statisch möglich. Das heißt aber, daß das gesamte System neu gebunden werden muß, was einen anschließenden Neustart des Systems erforderlich macht, der aber gerade vermieden werden soll.

Klassenhierarchien ermöglichen Modifikation und Erweiterung von Objektverhalten durch Ableitung. Allerdings können immer nur alle Methoden und Element-Variable geerbt werden. Instantiierte Objekte können immer nur zu *genau einer* Klasse gehören. Mehrfachvererbung scheint einen Ausweg zu bieten, ist aber schwierig zu handhaben und fehleranfällig.

Das Decorator-Pattern [GHJV94], dessen Struktur in Abbildung 3.4 dargestellt ist, erlaubt es Objekten zur Laufzeit scheinbar ihre Klasse zu ändern.

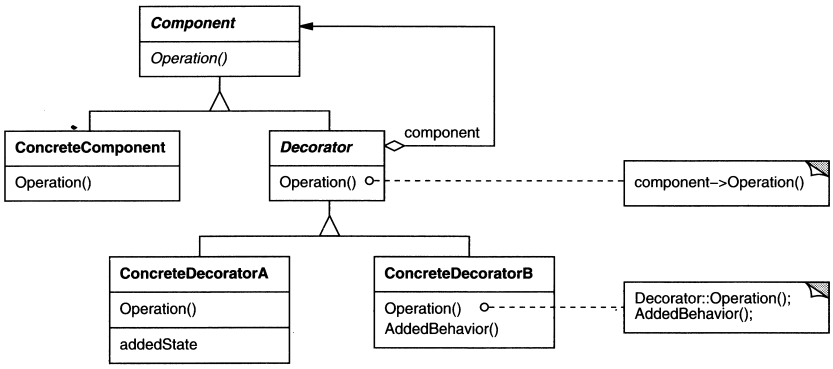

Abbildung 3.4: Decorator Pattern [GHJV94]

Als abstrakte Basisklasse fungiert *Component*. Davon abgeleitet werden die Klassen *Concrete Component* und *Decorator*. Die Basisfunktionalität (*Ope-*

ration) implementiert die Klasse *Concrete Component*. Die Klasse *Decorator* dient wiederum als Basisklasse für zusätzliche Funktionen. Indem einem *Component* Objekt zur Laufzeit Objekte der *ConcreteDecorator* Klassen zugeordnet werden, kann dessen Funktionalität erweitert werden. Damit können Objekte scheinbar ihre Klasse ändern.

Nachteilig am Decorator-Pattern ist es, daß das Interface der Decorator-Klasse immer identisch dem der Component-Klasse sein muß. Das Hinzufügen von Funktionen, die in Component nicht vorgesehen waren, ist später nicht möglich. Auch hier gehören Objekte immer zu genau einer Klasse. Ein flexibles System sollte aber die spätere Erweiterung von Objektverhalten ermöglichen.

Diese Überlegungen führen zu dem Schluß, daß es sinnvoll ist, das Verhalten Autonomer Objekte nicht in diesen selbst zu definieren, sondern es in eine eigene Klasse von Objekten auszulagern.

Intuitiv möchte man simulierten Objekten Eigenschaften zuordnen:

- die Zugehörigkeit zu einer oder mehreren logischen Klassen (Eine Taschenlampe ist etwa ein elektrisches Gerät, ein greifbarer Gegenstand und eine Lichtquelle gleichzeitig),
- Operationen, die auf dem Objekt ausgeführt werden können (die Taschenlampe kann an- und ausgeschaltet werden und als greifbares Objekt gegriffen und losgelassen werden)
- und schließlich eine grafische Repräsentierung (Die Taschenlampe wird durch die Geometrie ihres Gehäuses und eine Lichquelle dargestellt)[1].

Genau dies ist mit Autonomen Objekten und den sog. *Feature*-Objekten zur Laufzeit möglich. Da also das gesamte Verhalten eine Objektes durch die Eigenschaften festgelegt wird, wird nur eine einzige Klasse von Autonomen Objekten, gewissermaßen als Container für Feature-Objekte benötigt.

Wie Abbildung 3.5 illustriert, kann ein Autonomes Objekt beliebig viele Feature Objekte besitzen. Nachrichten, die das Autonome Objekt empfängt, werden an das passende Feature weitergeleitet und dort verarbeitet. Ein Simulationsschritt wird ausgeführt, in dem alle Features einen Simulationsschritt ausführen und die empfangenden Nachrichten verarbeiten. Sendet ein Feature selbst eine Nachricht aus, bedient es sich der Funktionalität "seines" Autonomen Objektes.

[1]Die Verbindung zwischen Simulationsmodell und Szenengraphen wird durch eine spezielle Klasse von Eigenschaften hergestellt (Klasse *Visible* siehe Abschnitt 4.3.3)

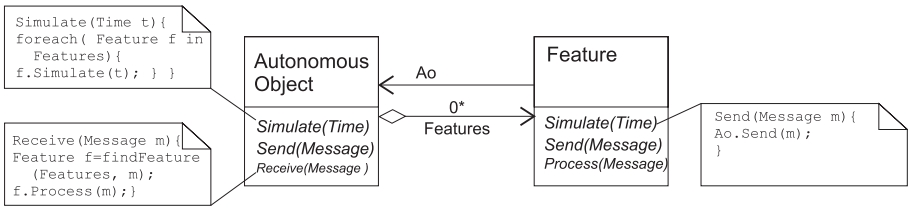

Abbildung 3.5: Autonome Objekte und Features

3.5 Kommunikation über Nachrichten

Es wurde bereits angedeutet, daß Autonome Objekte über Nachrichten kommunizieren. Da das Verhalten eines Autonomen Objektes durch seine Features bestimmt wird, ist die Schnittstelle eines Autonomen Objektes (also die Menge der Nachrichten, die es akzeptiert) ebenfalls abhängig von den zugeordneten Features. Diese *Signatur* wird durch die Vereinigungsmenge der Signaturen aller Features gebildet, die einem Autonomen Objekt zugeordnet sind. Autonome Objekte können also nicht direkt durch Aufruf von Methoden miteinander kommunizieren, sondern senden und empfangen Nachrichten-Objekte.

Die Kommunikation über Nachrichten ist noch aus einem weiteren Grund wichtig, sie können gepuffert und zu einem genau definierten Zeitpunkt übertragen werden. Die Simulation von kontinuierlichen Vorgängen muß nämlich in diskreten Schritten erfolgen:

Zunächst werden alle Berechnungen für einen Zeitpunkt t durchgeführt, erst wenn diese abgeschlossen sind, kann der nächste Simulationsschritt für den Zeitpunkt $t+\Delta t$ beginnen. Es muß unbedingt vermieden werden, daß für einen Teil des Systems noch der Zeitpunkt t gültig ist, während der andere Teil sich bereits in $t + \Delta t$ befindet. Andernfalls kann die Konsistenz der Simulation nicht garantiert werden.

3.6 Das Simulationssystem

3.6.1 Die Simulationsschleife

Aufgabe des Simulationssystems ist die Durchführung der Simulation und die Übertragung von Nachrichten. Außerdem stellt es eine globale Simulationszeit bereit.

Um die Simulation durchzuführen, iteriert das Simulationssystem das

Verzeichnis der Autonomen Objekte und übergibt die Kontrolle nacheinander an jedes AO. Dieses verarbeitet daraufhin - gemäß den ihm zugeordneten Eigenschaften - die Nachrichten in seinem Eingangspuffer und sendet seinerseits Nachrichten, die zunächst im Ausgangspuffer abgelegt werden. Die Reihenfolge, in der die Autonomen Objekte abgearbeitet werden, spielt keine Rolle. Für alle gilt während der gesamten Iteration dieselbe Simulationszeit.

Nachdem alle Autonomen Objekte abgearbeitet wurden, kann mit der Übertragung der Nachrichten begonnen werden. Dazu werden die Nachrichten in den Ausgangspuffern der Sender in die Eingangspuffer der Empfänger verschoben. Die Empfänger werden dabei durch das Empfänger-Attribut der Nachricht identifiziert.

Bevor nun als letztes die Simulationszeit weitergeschaltet wird, muß berücksichtigt werden, daß sich möglicherweise noch nicht alle Autonomen Objekte in einem gültigen Endzustand befinden. Es werden solange weitere Simulationszyklen durchgeführt, bis alle Autonomen Objekte einen Endzustand erreicht haben. Dieses Vorgehen ist notwendig, viele Kommunikationsprotokolle den Austausch mehrerer Nachrichten erfordern um einen Simulationsschritt zu beenden. Zeit vergeht aus konzeptioneller Sicht nur zwischen zwei Simulationsschritten, für das Senden und empfangen von Nachrichten wird keine Simulationszeit benötigt. Um das gesagte noch einmal exakt zu definieren, soll das folgende Pseudocodefragment dienen:

```
time = 0.0
loop:
    simlist = aolist
    while not simlist.empty():
        notfinished = []
        for ao in simlist:
            ao.sim()
            if not ao.finished:
                notfinished.append(ao)
        simlist = notfinished
        propagateMessages()

    time = time + dt
```

3.6.2 Struktur Autonomer Objekte

Abbildung 3.6 skizziert den Aufbau eines Autonomen Objektes. Ankommende Nachrichten werden zunächst in einem Eingangspuffer zwischengespei-

chert, bis das AO in der Simulationsschleife an die Reihe kommt. Die ankommenden Nachrichten werden auf die im AO enthaltenen Feature-Objekte verteilt. Falls eine Nachricht nicht verteilt werden kann, liegt ein Fehler vor (siehe dazu Abschnitt 3.9.4).

Die Features verarbeiten die Nachrichten gemäß den ihnen zugeordneten Protokollen. Dabei werden möglicherweise Nachrichten gesendet. Diese werden in den Ausgangspuffer abgelegt. Sobald alle Nachrichten aus dem Eingangspuffer verarbeitet wurden, geht die Kontrolle zurück an das Simulationssystem.

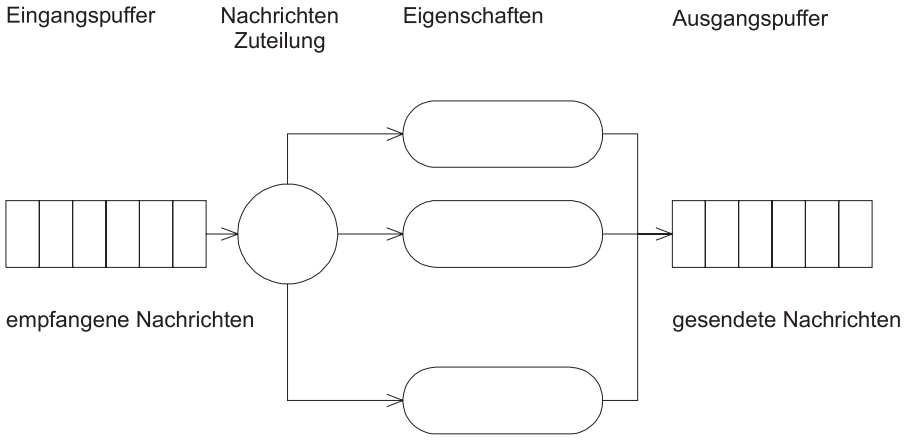

Abbildung 3.6: Aufbau eines Autonomen Objektes

Weiterhin besitzt jedes AO eine systemweit eindeutige ID. Diese ID wird bei der Erzeugung des Autonomen Objektes zugewiesen und und dient zu seiner Adressierung.

3.6.3 Nachrichten Kommunikation

Nachrichten dienen dem Informationsaustausch zwischen Autonomen Objekten. Nachrichten können ein einzelnes AO gerichtet werden (*unicast*) oder an eine Liste von Autonomen Objekten (*multicast*). Weiterhin enthalten sie eine Typkennung sowie beliebige weitere Daten.

Bei der Durchführung der Nachrichtenübertragung werden alle Nachrichten von den Ausgangspuffern der Autonomen Objekte in die Eingangspuffer der Empfänger verschoben. Da Nachrichten vom Empfänger nur gelesen und nicht verändert werden können, genügt es, eine Referenz auf das Objekt zu übertragen.

Für Nachrichten mit mehreren Empfängern wird die Referenz auf das Nachrichtenobjekt in den Eingangspuffer jedes Empfängers kopiert.

Neue Nachrichtenklassen werden mit Hilfe von reflexiven Containern. Diese wiederum werden über ein Container-Descriptor Objekt definiert (vergl. Abbildung 3.8 auf Seite 52). Das Container-Descriptor Objekt definiert, welche Daten welecher Typen unter welchem Schlüssel in einem Container Objekt enhtalten sind. Die Daten der Nachrichten sind in einem Container enthalten und über dessen Referenz auf das Descriptor Objekt, kann der Typ der Nachricht ermittelt werden.

3.7 Features

Features bestimmen die Reaktion eines Autonomen Objektes auf empfangene Nachrichten und definierten mit Hilfe des in ihnen enthaltenen Protokollobjektes deren Verarbeitung. Features bestehen aus Attributen, Operationen und einem Protokoll.

3.7.1 Attribute

Den inneren Zustand eines Features bestimmen Attribute. Attribute sind Objekte im Sinne der OOP und können beliebige Klassen haben. Die Vereinigungsmenge der Attribute aller Features eines Autonomen Objektes definiert dessen inneren Zustand.

Die Attribute eines Features werden in einem sogenannten reflexiven Container gehalten. Diese Container verhalten sich wie Dictionaries (In der STL Map genannt). Der Name eines Attributes bildet den Schlüssel beim Zugriff. Im Unterschied zu den STL-Maps, bei denen alle Daten einen einheitlichen Typ besitzen müssen, kann ein reflexiver Container Daten beliebiger unterschiedlicher Typen enthalten. Auch kann über einen Schlüssel abgefragt werden, welchen Typ das zugehörige Datum besitzt (daher der die Benennung reflexiver Container).

Attribute sind standardmäßig außerhalb des Features, zu der sie gehören für andere Features desselben AO sichtbar und lesbar, können aber durch nur durch Methoden des Features, zu dem sie gehören, geschrieben werden. Allerdings kann der Lesezugriff von außen explizit verboten werden. Der direkte Zugriff durch andere Autonome Objekte ist in keinem Fall möglich.

Features definieren einen Namensraum (*Namespace*), in dem die Namen der Attribute enthalten sind. Attribute anderer Features desselben AO können in der Form `feature.attribute` angesprochen werden.

Features werden durch Ableitung von einer abstrakten Basisklasse `Feature` definiert. Den einfachsten Fall stellt das *leere* Feature dar, die weder Attribute noch Operationen oder Protokolle definiert. Dieses leere Feature kann

benutzt werden, um Autonome Objekte zu markieren.

3.7.2 Operationen auf Attributen

Bei Operationen handelt es sich um meist kurze Programme, die eine Nachricht verarbeiten, und daraufhin den inneren Zustand der Eigenschaft verändern. Auch können Nachrichten an andere Autonome Objekte versendet werden. Insbesondere ist es möglich, daß Nachrichten an sich selbst (genauer an das Autonome Objekt dem die Eigenschaft zugeordnet ist) zu senden. Da in der Nachricht nur die Identifikation des empfangenden AO enthalten ist, hängt es vom Typ der Nachricht ab, welches Feature diese verarbeitet. Auf diese Weise ist es möglich, daß Features untereinander auf die selbe Art und Weise über Nachrichten kommunizieren, wie dies zwischen Autonomen Objekten möglich ist.

Das Feature-Objekt führt seine Operationen nicht selbst aus sondern delegiert diese Aufgabe an das in ihr enthaltene Protokollobjekt. Dieses stellt auch die Signatur zur Verfügung mit deren Hilfe das passende Feature-Objekt zur Verarbeitung einer Nachricht ausgewählt wird.

3.8 Protokolle

Während die Eigenschaftsobjekte Attribute und Operationen definieren, obliegt deren Ausführung dem Protokollobjekt, von dem jedes Feature höchstens eines besitzen kann.

Kommunikationsprotokolle können mit Hilfe deterministischer endlicher Automaten definiert werden. Die Menge aller erlaubten Nachrichtensequenzen entspricht damit der Menge der regulären Ausdrücke.

Formal ist ein deterministischer endlicher Automat durch ein Quintupel $(Q, \Sigma, \delta, q_0, F)$ definiert [HoUl88]. Dabei ist Q eine endliche Menge von Zuständen, Σ ein endliches Eingabealphabet δ eine Übergangsfunktion, q_0 der Startzustand und F die Menge der Endzustände.

Protokollobjekte definieren mit Hilfe eines solchen Automaten das Kommunikationsprotokoll für ein Feature. Das Eingabealphabet ist dabei die Menge von Nachrichten, die ein Feature verarbeiten kann, auch dessen Signatur genannt. Die Übergangsfunktion $\delta(q, a)$, definiert dabei den Folgezustand den der Automat einnimmt, wenn er im Zustand q die Nachricht a empfängt. δ wird als Tabelle $\Sigma \times Q$ dargestellt. Einem Zustand q kann ein Programm zugeordnet werden welches aus Operationen des übergeordneten Features besteht. Dieses wird mit der Einnahme des Zustandes ausgeführt. Das Eingabealphabet Σ wird von dem Feature referenziert und als Signa-

tur nach außen präsentiert. Weiter enthält das Protokollobjekt den Start- und die Endzustände. Die Endzustände sind für das Simulationssystem von Wichtigkeit, da ein neuer Simulationszyklus erst beginnen kann, wenn die Features aller Autonomen Objekte einen Endzustand erreicht haben. Der Startzustand wird genau dann eingenommen, wenn daß Autonome Objekt zum ersten mal die Kontrolle erhält (also im ersten Simulationszyklus). Im Startzustand können Ressourcen, wie Dateien oder Geräte belegt werden die zur Laufzeit benötigt werden.

Die Beschreibung von Features als Automat bietet eine Reihe von Vorteilen. Die Darstellung als Automat strukturiert den Programmcode von Eigenschaften in eine Reihe von Zuständen und Übergängen. Die Implementierung von Protokollen geschieht nicht in einem monolithischen Programmblock oder Skript sondern in wenigen Zeilen pro Zustand. Die Auswertung des Nachrichtentyps und die darauf folgende Auswahl des passenden Programmsegmentes übernimmt das System und vereinfacht so den Code weiter. Schließlich läßt sich ein Automat elegant in Form einer grafischen Notation darstellen. Dies erleichtert sowohl die Dokumentation als auch die Modellierung von Features, etwa mit einem grafischen Authoring-System.

3.8.1 Konfiguration Autonomer Objekte zur Laufzeit

Wie bereits zu Beginn dieses Kapitels ausgeführt wurde, ist die Möglichkeit Autonome Objekte zur Laufzeit des VR-Systems zu konfigurieren, eine wichtige Eigenschaft des hier vorgestellten Systems.

Die Zuordnung von Features an Autonome Objekte erfolgt durch Referenzen, zwischen Autonomen Objekten und Features (vergl. Abbildung 3.5 auf Seite 41). Diese kann durch den Benutzer über eine geeignete Benutzungsschnittstelle leicht modifiziert werden.

Ebenso können die Werte der Attribute zur Laufzeit geändert werden.

Die endlichen Automaten, die das Verhalten der Features und letztlich der Autonomen Objekte bestimmten bestehen aus zwei Komponenten: Der Definition des Automaten durch das Quintupel $(Q, \Sigma, \delta, q_0, F)$ und dem Programmcode, der innerhalb der Zustände des Automaten ausgeführt wird. Die Definition des Automaten kann in einer Datenstruktur gehalten werden, die durch eine Benutzungsschnittstelle vom Anwender zur Laufzeit problemlos manipuliert werden kann.

Übrig bleibt der eigentliche Programmcode. Handelt es sich um Code, der in einer Skriptsprache geschrieben ist, ist kein Aufwand zu treiben, nach der Modifikation wird einfach der neue Code interpretiert, ohne das ein Neustart nötig wird.

Liegt aber Code in einer Sprache vor, die in Maschinensprache übersetzt

wird, muß dieser anschließend an das VR-System gebunden (*linked*) werden. Wird ein statischer Linker verwendet, ist ein Neustart des Systems unausweichlich. Moderne Betriebssysteme bieten aber die Möglichkeit Teile eines Programmes, die als sogenanntes *Shared Object* (UNIX) oder als *Dynamic Link Library, DLL* (Microsoft Windows) vorliegen, an ein laufendes System zu binden. Der schematische Ablauf ist folgendermaßen:

```
SoPtr sharedLib = dlopen( "mylib.so" );
int *i = (int*) getsym( sharedLib,"counter" );
FnPtr f = (FnPtr) getsym(sharedLib, "function");
 ...
int *i = 4711;
f(...);
dlclose(sharedLib);
```

Zunächst wird das Shared Object unter Angabe des Names der Objektdatei geöffnet. Anschließend verschafft sich das Programm Zeiger auf die im Shared Object enthaltenen Symbole, hier eine Variable und eine Funktion. Diese können dann normal verwendet werden. Wird das Shared Object nicht mehr benötigt, kann es geschlossen und freigegeben werden. Die Zeiger auf die Symbole verlieren natürlich ihre Gültigkeit. Der Programmcode des Shared Objects kann zur Laufzeit des Hauptprogramms modifiziert und übersetzt werden. Anschließend wird das Shared Object, wie oben dargestellt neu gebunden und die Änderung werden im Hauptprogramm wirksam, ohne das dieses ebenfalls neu übersetzt werden muß.

Abbildung 3.7 illustriert, wie dieses Verfahren verwendet wird, um die Operationen zu definieren, die ein Protokollobjekt durchführt, wenn es Nachrichten verarbeitet. Die Tabelle mit den Zustandsübergängen $\delta(q, a)$ wird im Simulationssystem gehalten und kann problemlos zur Laufzeit verändert werden. Der Programmcode, der innerhalb der Zustände ausgeführt wird, ist in einem Shared Object enthalten. Dieses definiert für jeden Zustand q den der Automat annehmen kann eine Funktion `q(Message)`, die die empfangene Nachricht verarbeitet. Muß dieser Programmcode geändert werden, kann dies durch das oben beschriebene Verfahren geschehen, ohne das das Simulationssystem angehalten und neu gestartet werden muß.

3.8.2 Adressierung von Autonomen Objekten

Eine Voraussetzung für die Kommunikation über Nachrichten ist die Bekanntheit eines Empfängers. Der oder die Empfänger einer Nachricht werden durch Referenzen auf Autonome Objekte realisiert.

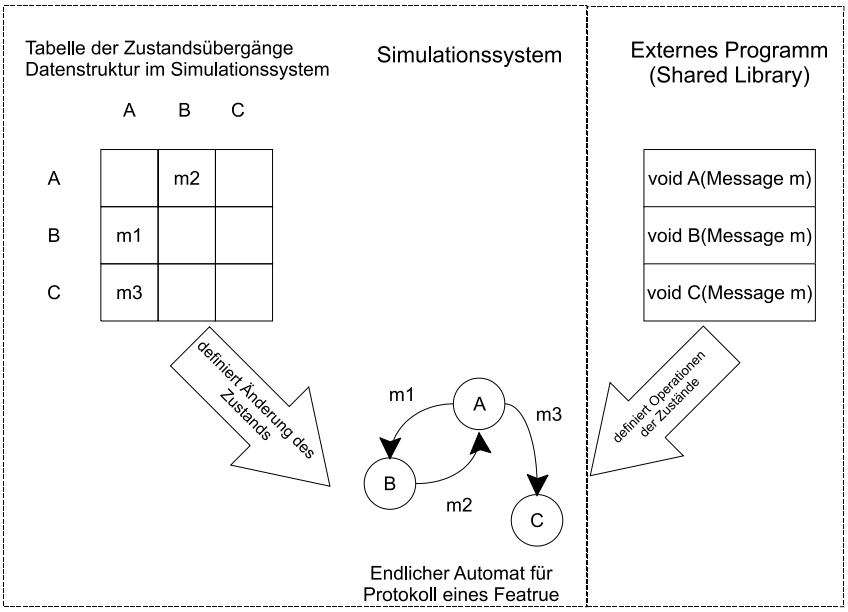

Abbildung 3.7: Shared Library definiert Operationen eines Protokolls

Für statisches Objektverhalten werden Autonome Objekte fest miteinander verbunden. Dieses Vorgehen ist analog zu den Verbindungen zwischen Feldern von Knoten durch Routes in VRML oder dem Trigger-Action Konzept von WorldUp. Für Autonome Objekte definiert ein Attribut welches eine oder mehrere IDs von Autonomen Objekten enthält den oder die Empfänger einer Nachricht.

In Simulationen mit dynamischem Objektverhalten können Sender und Empfänger wechseln. Hier dient eine Anfrage and die Welt-Datenbasis (siehe Abschnitt 3.12) dazu die Empfänger einer Nachricht zu finden. Diese Anfrage kann in jedem Simulationsschritt wiederholt werden, um die Liste der Empfänger zu aktualisieren. Empfänger können entweder Objekte mit bestimmten Eigenschaften sein, oder Objekte die sich in einem räumlichen Volumen, beispielsweise in der Nähe des Senders befinden.

3.9 Fehlerbehandlung

Bisher haben wir die Möglichkeit von Fehlern nicht berücksichtigt. Dies soll nun geschehen. Prinzipiell sind drei Arten von Fehlern zu klassifizieren:

1. Fehler im Programmcode
2. Fehler auf Protokollebene
3. Fehler bei der Zuordnung von Eigenschaften

Die folgenden Abschnitte diskutieren die Fehlerursachen und die Fehlerbehandlung.

3.9.1 Fehler im Programmcode

Hiermit sind alle Fehler gemeint, die bei der Programmierung von Code für Features gemacht werden können. Wird zur Programmierung eine Sprache verwendet, die vor der Ausführung in Maschinensprache übersetzt (compiliert) wird, wie etwa C oder C++ können die Fehler weitreichend sein und zu Fehlfunktionen oder sogar Absturz des Systems führen. Etwas mehr Sicherheit bieten interpretierte Sprachen wie JAVA oder Python. Hier können zumindest Abstürze weitgehend ausgeschlossen werden. Vorsicht muß auch die Anforderung von Ressourcen walten. Ein nicht funktionierendes Gerät kann das ganze System zum halten oder Absturz bringen. Abhilfe schafft in diesem Fall ein System zur Anbindung von Interaktionsgeräten, welches in Kapitel 5 vorgestellt wird.

Letztendlich kann ein Rahmensystem niemals so entworfen werden, daß es immun gegen die Programmierfehler seiner Anwender ist. Problematisch ist, daß ein Rahmensystem immer die den Programmablauf an von Anwendern programmierten Code übergeben muß, über den es keine Kontrolle hat. Abhilfe könnte hier eine Implementierung von Autonomen Objekten durch Threads schaffen. Dies würde erlauben, daß das Gesamtsystem weiterläuft, auch wenn einige Autonome Objekte in Endlosschleifen stecken oder sogar abstürzen.

3.9.2 Protokollfehler

Protokollfehler liegen dann vor, wenn Sender- und Empfängerprotokoll nicht völlig aufeinander abgestimmt sind. In gewisser weise handelt es sich hierbei ebenfalls um Programmierfehler, nur daß die Folgen für das Gesamtsystem weit weniger gravierend sind. Lediglich die beteiligen Autonomen Objekte werden nicht korrekt funktionieren.

Ein Protokollfehler liegt dann vor, wenn entweder

- ein AO keine Eigenschaft besitzt, in dessen Signatur der Typ der Nachricht vorkommt oder

- Das Feature, für die die Nachricht bestimmt ist, sich in einem Zustand befindet, aus der kein Übergang für diesen Typ existiert.

In beiden fällen reagiert das System mit der Rücksendung einer Fehlerquittung *Negative Acknowledgement, NACK* an den Absender. Das Protokoll des sendenden Features kann seinerseits auf eine Fehlerquittung reagieren, da es sich prinzipiell um eine normale Nachricht handelt (die Unterschiede die doch bestehen, werden in Abschnitt 3.9.4 diskutiert).

3.9.3 Zuordnungsfehler

Fehler dieser Art können in der Modellierungsphase eines zu simulierenden Systemes gemacht werden. Sie entstehen bei der falschen "Verdrahtung" des Systemes. Werden Autonome Objekte so verknüpft, daß ihre Features nicht zueinander passen, werden Nachrichten gesendet, die von der Gegenstelle nicht verarbeitet werden können. Beispielsweise wird versucht, ein Objekt zu greifen, daß gar nicht greifbar ist (d.h. dem das entsprechende Feature nicht zugeordnet wurde). Die Folge ist ein Protokollfehler (siehe letzter Abschnitt). Die Ursache ist aber kein Programmierfehler des Entwicklers der Eigenschaft sondern dieser liegt bei ihrem Anwender. Solche Fehler lassen sich glücklicherweise leicht beheben.

3.9.4 Quittungen und Fehlerquittungen (NACK)

Um das Funktionieren von Kommunikationsprotokollen zu gewährleisten werden Quittungen (Acknowledgement, ACK) und Fehlerquittungen versendet (negative Acknowledgement, NACK) . Mit ersterem quittiert der Empfänger einer Nachricht deren erfolgreiche Verarbeitung, letztere zeigen einen Fehler an.

Ein Fehler kann zum Beispiel vorliegen wenn die Ausführung einer Operation nicht erlaubt war, oder die Nachricht nicht verarbeitet werden konnte weil ein Zuordnungs- oder Protokollfehler vorlag.

Der Sender einer Nachricht kann für den Empfang von ACK und NACK eigene Zustände vorsehen und bei Fehlern entsprechend reagieren.

Sieht das Protokoll des (ursprünglichen) Senders im aktuellen Zustand des Protokolls einen NACK-Übergang vor, kann der Fehler behandelt werden. Insoweit ist NACK also eine ganz normale Nachricht. bei genauerer Betrachtung unterscheiden sich ACK und NACK aber doch von allen anderen Nachrichten:

Erstens werden ACK und NACK-Nachrichten ignoriert, wenn das Protokoll ihre Verarbeitung nicht vorsieht. Andernfalls würden weitere NACK-

Nachrichten erzeugt, und so nach kurzer Zeit das System von ihnen überflutet werden (die sogenannte NACK explosion).

Zweitens sind ACK und NACK die einzigen Nachrichten auf die mehr als eine Eigenschaft pro AO reagieren kann. Anders ausgedrückt können ACK und NACK in der Signatur beliebig vieler Eigenschaften eines Autonomen Objektes vorkommen. Aus diesem Grund enthalten diese Nachrichten ein weiteres Attribut, welches neben Sender und Empfänger AO auch die Eigenschaft identifiziert, für die die Fehlerquittung bestimmt ist.

Diese Unterschiede sind aber im System verborgen und haben keinen Einfluß auf den Umgang mit Nachrichten und Protokollen und der Gestaltung von Feature-Klassen und deren Protokollen.

3.9.5 Architektur des Simulationssystems

Zu simulierende Objekte werden als Autonome Objekte (AO) dargestellt. Diese sind völlig unabhängig von den grafischen Objekten der Szenenhierarchie. Das Simulationssystem enthält eine Liste aller Autonomen Objekte und arbeitet diese der Reihe nach ab.

Das Verhalten Autonomer Objekte wird von diesen getrennt, durch Feature-Objekte, kurz *Features* beschrieben. Beliebig viele dieser Features können Autonomen Objekten zugeordnet werden. Die Zuordnung kann zur Laufzeit erfolgen. Features dienen ebenfalls zur logischen Klassifizierung von Autonomen Objekten.

Features bestehen aus drei wesentlichen Komponenten, einer Signatur (*Signature*), einem endlichen Automaten, der ein Nachrichtenprotokoll definiert und einem reflexiven *Container*, der den inneren Zustand beschreibt.

Die Kommunikation zwischen Autonomen Objekten erfolgt über Nachrichten (*Message*). Zwar treten Autonome Objekte als Sender und Empfänger von Nachrichten auf, die eigentliche Verarbeitung und das Versenden von Nachrichten übernehmen aber ebenfalls die Features. Die zu transportierenden Daten halten Nachrichten in einem reflexiven Container.

Die Nachrichtenübertragung erfolgt gepuffert. Zunächst werden alle Nachrichten, die in einem Zyklus ein einem Puffer für jedes AO gesammelt. Nach Beendigung des Simulationszyklus wird werden die Nachrichten aus den Ausgangspuffern in die Eingangspuffer der empfangenden autonomen Objekte übertragen. Die grafische Darstellung wird aktualisiert und ein neuer Simulationszyklus beginnt.

Reflexive Container verhalten sich wie Dictionaries, sie bilden Schlüssel auf Daten ab. Diese Abbildung ist eindeutig. Ein Container kann beliebig viele Schlüssel-Daten-Paare enthalten. Die Daten selbst besitzen Typ und Wert. Dabei kann der Typ der Werte für jedes Element des Containers unter-

schiedlich sein und über den Schlüssel erfragt werden (daher die Bezeichnung *reflexiv*). Container dienen zur Speicherung der Attribute von Features und der Daten in Nachrichten.

Die Struktur eines Containers wird durch Deskriptoren *ContainerDescriptor*) festgelegt. Typ und Schlüssel jedes Eintrags. Jeder Container besitzt genau einen Deskriptor.

Signaturen (*Signature* beschreiben, welche Nachrichten ein Feature "versteht", also verarbeiten kann. Die Signatur besteht aus einer Liste von Schlüsseln (*Keys*, die aus Container Deskriptoren bestehen. Der Test erfolgt über den Vergleich mit dem Container Deskriptor der Nachricht. Die Signatur eines Autonomen Objektes ist die Vereinigungsmenge der Signaturen aller seiner Features.

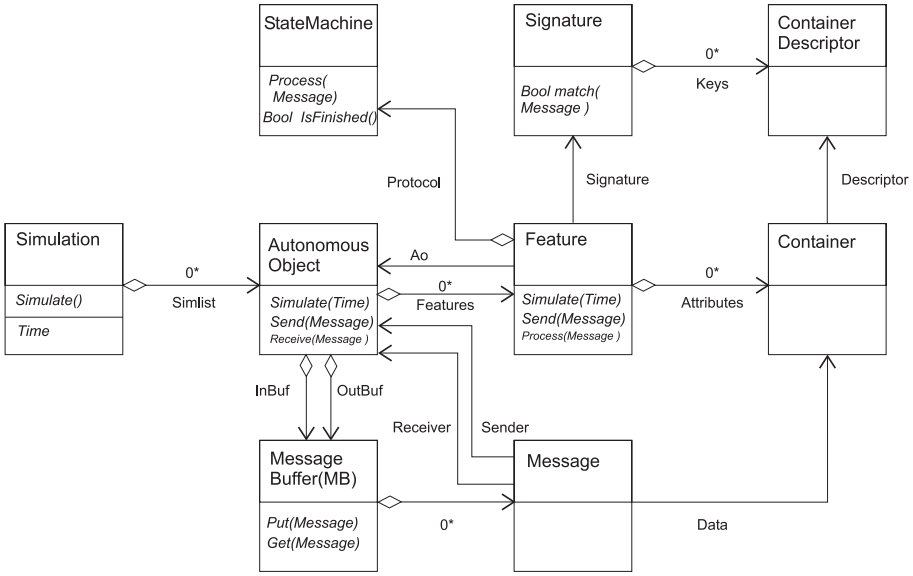

Abbildung 3.8: UML Diagramm der Systemarchitektur

3.10 Prozeßstruktur

Das System besteht aus zwei mindestens parallel laufenden Prozessen, dem Rendering Prozeß und dem Simulationsprozeß. Der Rendering Prozeß traversiert den Szenenhierarchie und stellt die Szene grafisch dar. Der Simulationsprozeß verarbeitet die Daten der Interaktionsgeräte und iteriert die Liste der zu simulierenden Autonomen Objekte. Diese wiederum greifen auf die grafischen Objekte in der Szenenhierarchie lesend und schreibend zu, steuern also die grafische Darstellung.

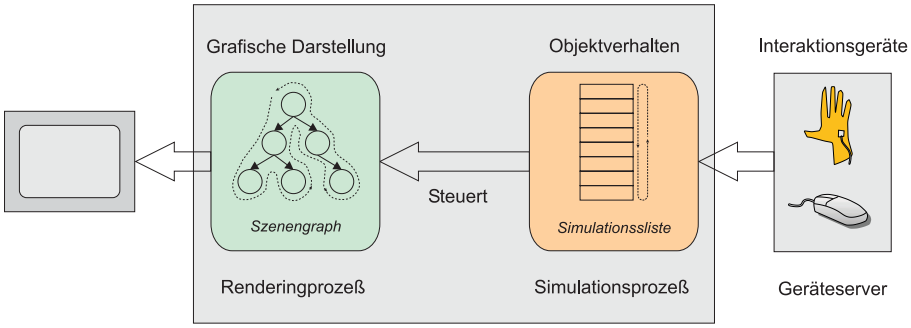

Abbildung 3.9: Parallele Prozesse für grafische Darstellung und Simulation

Die Hauptschleifen beider Prozesse laufen, unabhängig voneinander so schnell wie möglich ab. Um beide Prozesse zu synchronisieren, können die Änderungen an der grafischen Szenenbeschreibung gepuffert, und während eines Rendezvous beider Prozesse propagiert werden.

In in vielen Fällen kann auf eine Synchronisation verzichtet werden, da die Wahrscheinlichkeit, daß ein bestimmtes Attribut eines grafischen Objektes gleichzeitig von der Simulation geschrieben wird, während es der Renderer liest, äußerst gering ist. Selbst dann bleibt der resultierende Fehler in der Darstellung typischerweise unbemerkt, da seine Sichtbarkeit lediglich einen Sekundenbruchteil, nämlich die Dauer eines Frames beträgt.

Wie die Simulation selbst auf mehrere Prozesse verteilt werden kann, wird in Abschnitt 3.11 beschrieben.

3.11 Parallelisierung der Simulation

Stehen mehr als zwei Prozessoren zur Verfügung, kann die Simulation selbst parallelisiert werden. Abbildung 3.10 illustriert das Vorgehen anhand dreier Simulationsprozesse.

Jeder Simulationsprozeß fordert eine feste Zahl von Autonomen Objekten zur Simulation an. Diese Zahl sollte klein gegenüber der Gesamtzahl der zu simulierenden Objekte sein. Andererseits sollte sie deutlich über eins liegen damit der Aufwand und die Zahl der Anforderungen nicht zu hoch wird. In der Simulationsliste der Autonomen Objekte wird festgehalten, welches AO bereits abgearbeitet wurde, welche sich in Bearbeitung befinden und welche noch abgearbeitet werden müssen.

Sobald ein Simulationsprozeß alle Autonomen Objekte bearbeitet hat, gibt er diese Frei und fordert eine neue Reihe an. Dies geschieht solange, bis alle Autonomen Objekte sich in einem gültigen Endzustand befinden.

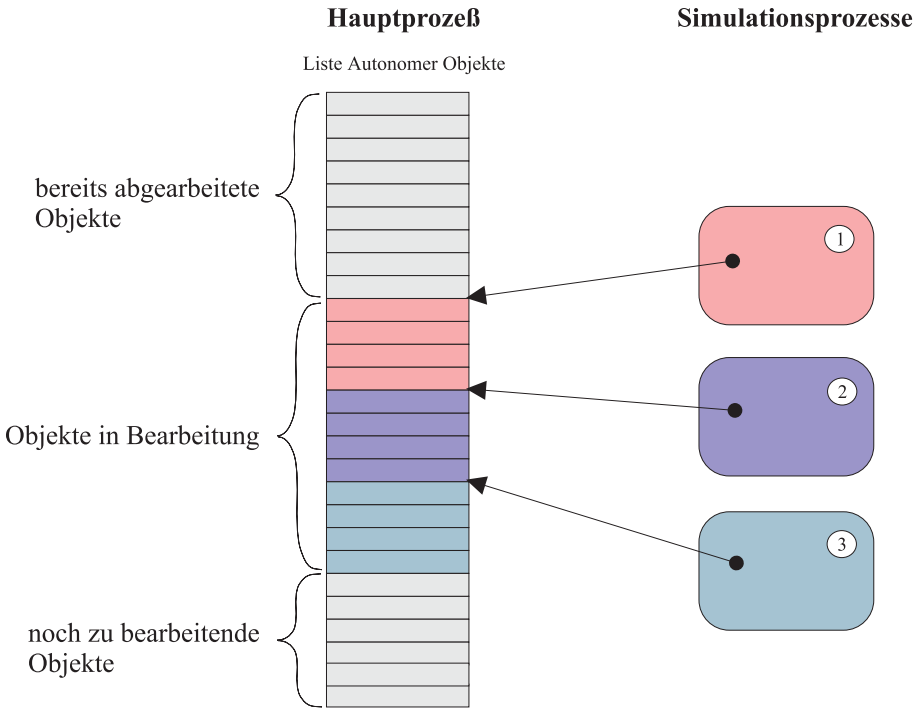

Abbildung 3.10: Verteilung der Simulation auf mehrere Prozesse

Daraufhin werden die Nachrichten übertragen und ein neuer Zyklus beginnt.

3.12 Die Welt-Datenbasis

Die Aufgabe der Welt-Datenbasis (WDB) ist es, Informationen bereitzustellen, die Autonome Objekte benötigen um miteinander zu interagieren. Dazu können Autonome Objekte Anfragen an die Datenbasis stellen, die Listen von Autonomen Objekten zurück liefern, die den Kriterien der Anfrage entsprechen.

Zwei Klassen von Suchkriterien können dabei unterschieden werden:

- Suche nach Eigenschaften
- Räumliche Suchkriterien

Anfragen and die WDB können aus mehreren Suchkriterien bestehen, die durch die binären Operationen verknüpft werden können. Auf diese Weise können Anfragen nach bestimmten Kombinationen von Eigenschaften und in Volumen suchen die aus Schnitt- oder Vereinigungsmengen einfacher Volumen bestehen.

Durch die Suche nach Features wird erreicht, daß die Anfrage nur solche Autonomen Objekte liefert, welche bestimmte Kommunikationsprotokolle verstehen. Dadurch kann das funktionieren der Kommunikation sichergestellt werden.

Räumliche Suchkriterien definieren Volumen, in denen nach Autonomen Objekten gesucht wird. Autonome Objekte selbst sind jedoch zunächst abstrakte Objekte, welche weder Position noch Ausdehnung im geometrischen Raum besitzen. Daher erfolgt die Suche zunächst nach grafischen Objekten, die die Suchkriterien erfüllen. Von diesen kann dann auf die assoziierten Autonomen Objekte zurück geschlossen werden.

Für die räumliche Suche in statischen Szenen kann auf die Operationen des verwendeten Renderers zurückgegriffen werden. Da diese ohnehin für die grafische Darstellung der Szene (Culling, Drawing) benötigt werden bieten die meisten Renderer eine Reihe von Schnittests zwischen primitiven Volumen und polygonaler Geometrie. Die Volumen umfassen beispielsweise Halbräume, Kugeln, achsparallele Quader sowie Frusta (Pyramidenstumpf).

Für dynamische Szenen, in denen sich viele Objekte bewegen ist der Szenengraph aber nicht die optimale Datenstruktur, um nach diesen Objekten zu suchen. Da die Szenenhierarchie auf einer statischen "ist enthalten in" Relation basiert, müßte die Szenenhierarchie für dynamische Szenen ständig angepaßt und umgestellt werden, da sie andernfalls entartet. Daher behilft man sich in der Praxis dadurch, daß Objekte, von denen bekannt ist, daß sie sich innerhalb der Szene bewegen, direkt unter die Wurzel des Szenengraphen einordnet. Damit degeneriert die die Suche nach solchen Objekten zu einer Iteration mit linearem Aufwand (vergl. Abschnitt 2.3.2).

Ausgesprochen ineffizient werden Operationen, die sich auf mehr als ein Objekt beziehen, hier steigt der Aufwand quadratisch mit der Zahl der beteiligen Objekte. Soll beispielsweise für jedes von n Autonomen Objekten dasjenige gesucht werden, welches diesem am nächsten ist, so wächst die Anzahl der Berechnungen auf $\frac{n^2}{2}$.

Dieses Problem trat etwa im Virtuellen Ozeanarium auf, wo für eine große Anzahl Fische Schwarmverhalten und Kollisionsvermeidung simuliert werden mußte (siehe Abschnitt 6.2). Das Problem konnte gelöst werden, in dem die WDB um eine Gitterstrukur erweitert wurde, in die die Autonomen Objekte eingeordnet wurden. Die Suche nach dem nächsten Nachbarn läßt sich dann in konstanter Zeit durchführen.

Die Welt-Datenbasis kann also neben dem Szenengraphen geeignete Datenstrukturen enthalten, die die effiziente Suche nach Autonomen Objekten im Raum unterstützt. Dies können, je nach Anwendungsfall Gitter, Quadtrees oder Octrees sein.

3.13 Einteilung der Welt-Datenbasis in ein Gitter

Um die oben diskutierten Probleme zu lösen, wird die Welt-Datenbasis in würfelförmige Zellen eingeteilt[2].

Zunächst kann dieses Gitter benutzt werden, um darin Autonome Objekte anhand ihrer Lage einzuordnen. Jede Zelle enthält eine Liste aller enthaltenen autonomen Objekte. Die Suche nach Autonomen Objekten innerhalb eines bestimmten Volumens wird dadurch wesentlich effizienter als die lineare Suche im Szenengraphen.

Die Lage eines Autonomen Objektes definiert sich durch die Position des zugeordneten grafischen Objektes innerhalb der Szene. Ist kein grafisches Objekt zugeordnet, ist die Lage eines Autonomen Objektes undefiniert und es kann nicht eingeordnet werden.

Den Zellen können auch Informationen über die statischen Teile der Szene bereithalten, um für dynamische Objekte schnell zwischen freien Gebieten und Hindernissen zu unterscheiden. Wie diese Information gewonnen wird, beschreibt der folgende Abschnitt.

3.13.1 Verfahren zur Erkennung von Hindernissen

Die statische Szene wird geladen und ein quaderförmiges, zum Koordinatensystem achsparalleles Hüllvolumen berechnet. Dieses wird in würfelförmige Zellen eingeteilt, deren Größe abhängig von der Anwendung eingestellt werden kann (siehe Abbildung 3.11 A)[3].

Daraufhin werden alle Zellen, die Polygone enthalten, als Hindernisse markiert (Abbildung 3.11 B). Im nächsten Schritt wird das Gitter "geflutet". Ausgehend von einem bekannten freien Punkt in der Szene (X-förmige blaue Markierung in Abbildung 3.11), werden alle Nachbarzellen, die nicht bereits als Hindernisse markiert sind, ebenfalls als "frei" definiert (C) . Zellen, die danach immer noch keine Markierung besitzen, liegen innerhalb von Hindernissen und werden daher ebenfalls als solche markiert (D). Auf diese Weise wird verhindert, daß Gebiete innerhalb von grafischen Objekten oder hinter außerhalb der Szene fälschlicherweise als "frei" erkannt werden. Auch werden Löcher in der Szenengeometrie, die von Modellierungsfehlern herrühren,

[2]Diese Einteilung in ein reguläres Gitter wurde gegenüber einem Ansatz mit Octrees [YaKF84, Tamm84] bevorzugt. Zwar bietet der Octree Vorteile im Hinblick auf den benötigten Speicherplatz, für Registrierung der dynamischen Objekte ist aber ein reguläres Gitter effizienter.

[3]Für die Aquarien des Virtuellen Ozeanariums (siehe Abschnitt 6.2 wurde eine Kantenlänge von einem halben Meter gewählt.

geschlossen. Beides hatte sich in in praktischen Anwendungen als problematisch erwiesen. Das so gewonnene Voxelgitter kann in Form einer Binärdatei gespeichert werden, die später beim Start des Systems ausgewertet wird und die Grundlage der Welt-Datenbasis bildet.

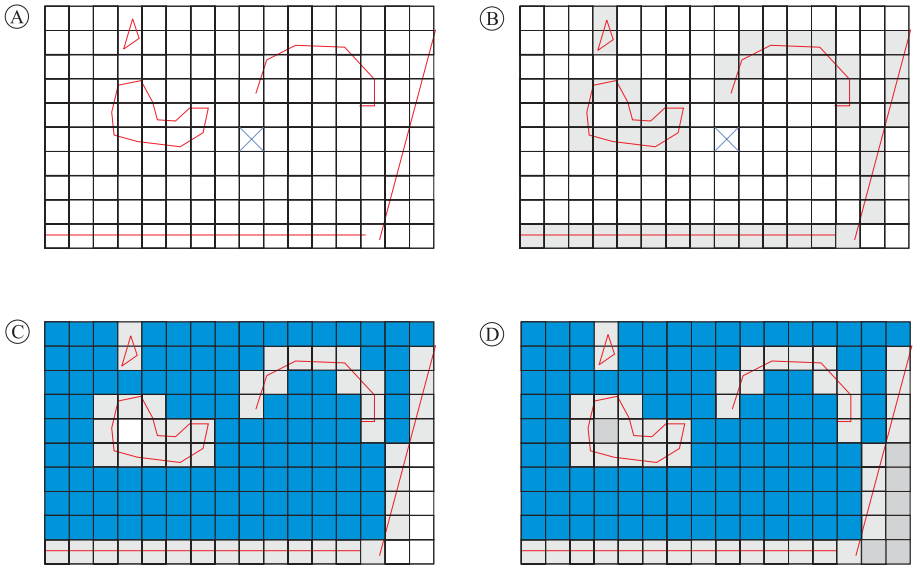

Abbildung 3.11: Schritte der Markierung der Zellen im Voxelgitter

Abbildung 6.14 auf Seite 135 illustriert den Zusammenhang zwischen grafischer Szene anhand einer Szene aus dem Virtuellen Ozeanarium.

3.13.2 Verfahren zur Pfadsuche in Virtuellen Umgebungen

Ein Problem, welches in dynamischen Virtuellen Umgebungen immer wieder zu lösen ist, ist die Suche nach Pfaden innerhalb der virtuellen Szene. Humanoide virtuelle Charaktere oder simulierte Lebewesen sollen sich auf natürliche Weise zielgerichtet innerhalb der virtuellen Welt bewegen ohne dabei mit Hindernissen zu kollidieren.

Für die Pfadsuche existiert bereits eine Reihe von Verfahren, die entweder auf Voxelgittern [BaTa98, Pint01, Stou99, Kuff98] oder Configuration Spaces [BaTa97, KuLV00] suchen. Diese werden in [FrKu02] vorgestellt und verglichen.

Das im Rahmen dieser Arbeit entwickelte Verfahren eignet sich besonders für simulierte humanoide Charaktere. Diese bewegen sich im wesentlichen in der Ebene, können aber auch über schiefe Ebenen oder Treppen in der Senkrechten bewegen. Für Objekte, die schwimmen oder fliegen kann es leicht

angepasst werden, indem einige Randbedingungen weggelassen werden, die auf der ebenen Bewegung beruhen. An den entsprechenden Stellen wird darauf eingegangen.

Aufbau des Suchgraphen

Nachdem die Szene in ein Voxelgitter eingeteilt wurde und die Hindernisse identifiziert worden sind, müssen für simulierte Fahrzeuge und Menschen zunächst die Ebenen identifiziert werden, auf denen die Fortbewegung möglich ist. Dazu wird über allen besetzen Zellen die "Deckenhöhe" berechnet, also die Anzahl der freien Zellen darüber. Nur dort wo die Decke hoch genug ist, ist Fahren oder Gehen möglich. Für fliegende oder schwimmende Objekte ist diese Unterscheidung nicht nötig.

Aus den Nachbarschaftsbeziehungen zwischen den Zellen kann nun ein Graph aufgebaut werden, in dem dann die eigentliche Suche stattfindet. Bei der Erzeugung dieses Graphen müssen für gehende oder fahrende Objekte zusätzlich die Höhenunterschiede, die Deckenhöhe an der Grenze zwischen den Zellen sowie in der Mitte der potentiellen Nachbarzelle berücksichtigt werden. Nur wenn der Höhenunterschied nicht zu groß ist, um überwunden zu werden und die Decke hoch genug ist, wird die Nachbarzelle als Knoten in den Suchbaum eingetragen.

Optimierte Heuristik für die A^* Suche

Zur Suche wird nun der klassische A^* Algorithmus [RuNo95] verwendet. Es handelt sich um einen Depth-First Suchalgorithmus, der von einer Heuristikfunktion gesteuert wird. Diese Heuristik bestimmt, welcher Knoten als nächstes untersucht wird. Eine typische Heuristikfunktion ist

$\Delta x + \Delta y + \Delta z$. Heuristikfunktion für A^*

Für gehende oder fahrende Objekte wird diese Heuristik modifiziert, um den Vorzug der Bewegung in der Ebene zum Ausdruck zu bringen:

$\Delta x + (\Delta y)^2 + \Delta z$. Modifizierte Heuristik für A^*

Durch diese Modifikation verliert der Algorithmus seine theoretische Optimalität; speziell für Menschen ist dies aber ein natürliches Verhalten, man vermeidet Steigung und nimmt dafür einen längeren Weg in Kauf. Abbildungen 3.12 und 3.13 zeigen den Effekt, den die modifizierte Heuristik auf die Suche nach einem Pfad vom ersten in den dritten Stock hat. Gesucht wird ein

Pfad zwischen den roten Kugeln im ersten und im dritten Stock. Die grünen Säulen markieren die untersuchten Zellen. Man kann deutlich sehen, daß mit der modifizieren Heuristik weniger Zellen untersucht werden, das Verfahren also schneller konvergiert.

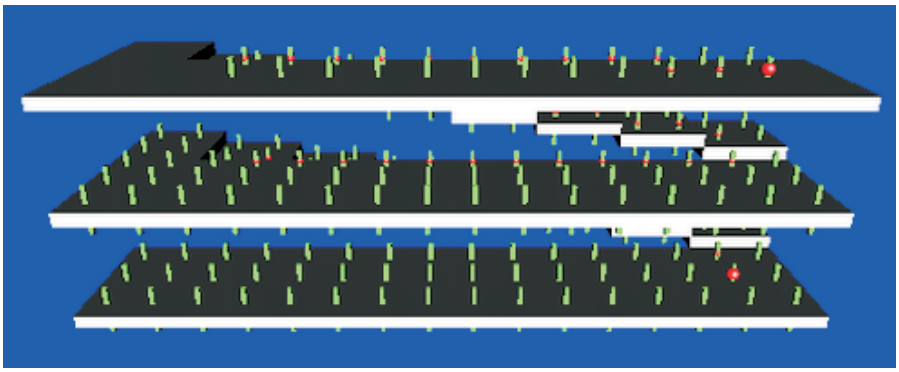

Abbildung 3.12: Suche mit der Standard Heuristik

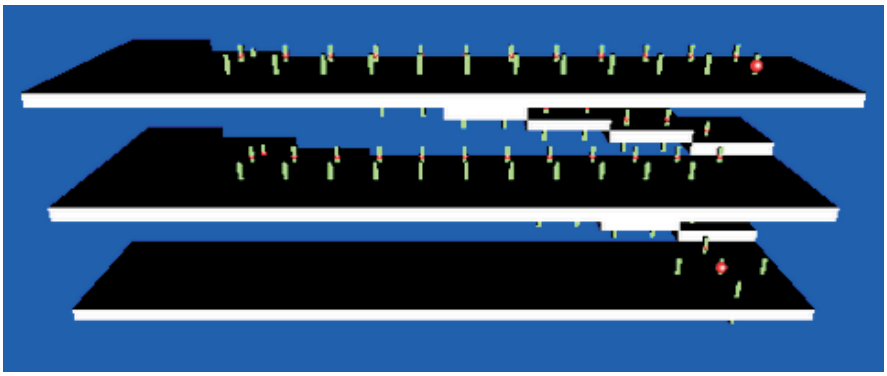

Abbildung 3.13: Schnellere Suche mit der modifizierten Heuristik

Für sehr komplexe Szenen stellt man fest, daß die Heuristik (auch die modifizierte) zu einer planlos scheinenden Suche führt. Betrachten wir die in Abbildung 3.14 dargestellte Szene, die aus 12 Räumen besteht die teilweise durch Flure miteinander verbunden sind. Jeder Raum und jeder Flur besteht aus einer großen Zahl von Zellen (nicht abgebildet). Die Aufgabe ist, vom Punkt A nach Punkt B zu finden.

Der Algorithmus wird zuerst die grau markierten Räume vergeblich durchsuchen. Für komplexe Szenen führt dies zu einer inakzeptablen Verlangsamung der Suche.

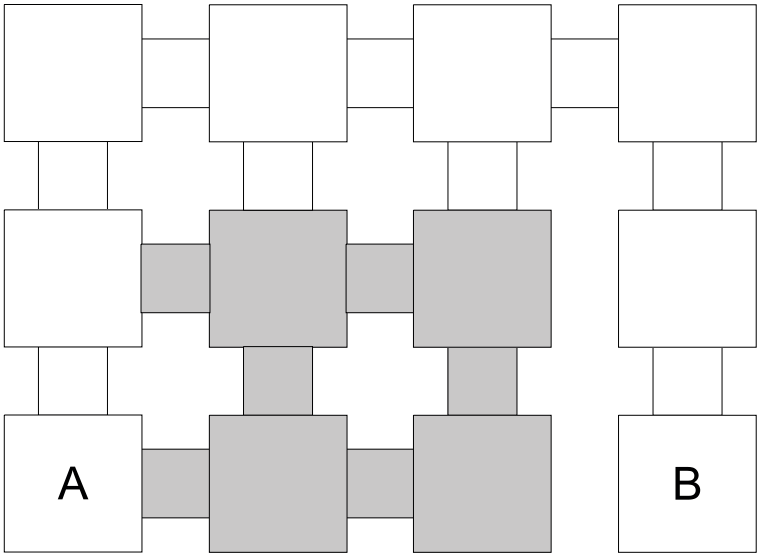

Abbildung 3.14: Problem mit großen Szenen

Automatische Partitionierung der Szene

Eine Lösung für das oben beschriebene Problem stellt die Partitionierung der Szene dar. Es wird versucht, zusammenhängende Gebiete zu Partitionen zusammenzufassen, die beispielsweise Räumen oder Fluren entsprechen. Zunächst wird ein Pfad zwischen den Partitionen gesucht. Ist dieser gefunden, wird zwischen den gefundenen Wegpunkten ein Pfad auf der Ebene der Zellen berechnet.

Zunächst bietet es sich an, Partitionen durch eine Suche auf den Zellen mit begrenzter Tiefe zu finden. Dies führt zu rautenförmigen Partitionen wenn vier Nachbarn berücksichtigt werden und runden Strukturen, wenn alle acht Nachbarn berücksichtigt werden. Ganze Räume werden nie zusammengefasst, wenn sie zu groß für die eingestellte Suchtiefe sind und die Form der Partitionen entspricht nicht der von Räumen oder Fluren in Gebäuden.

Daher wird ein anderer Ansatz verfolgt, der sich an der Raum-Flur Idee orientiert. Eine Partition wird hier aus Zellen zusammengesetzt, die sich gegenseitig "sehen" können. Damit ist gemeint, daß die Verbindung zwischen den Zellen frei von Hindernissen ist.

Das Verfahren arbeitet mit einer Breitensuche. Eine Zelle wird einer Partition nur dann hinzugefügt, wenn mindestens ein festgelegter Bruchteil der Zellen "gesehen" werden kann, die bereits zur Partition gehören. Sinnvolle Werte sind 70 bis 90 Prozent.

Das Vorgehen illustriert Abbildung 3.15. Die Zelle im Flur wird zu der

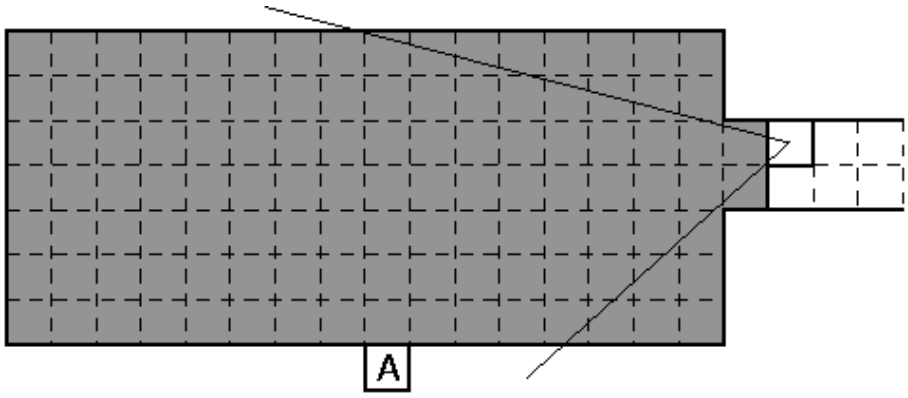

Abbildung 3.15: Partitionierung einer Szene

bestehenden Partition hinzugefügt, weil sie 99 von 114 Zellen "sieht". Der Algorithmus ist aufwendig, da er aber nur einmal vor der Laufzeit des VR-Systems ablaufen muß, kann dies zugunsten einer schnelleren Pfadsuche in Kauf genommen werden. Allerdings füht das Verfahren zu vielen kleinen Partitionen. Die Zelle in Abbildung 3.15, die mit A markiert ist, wird nicht zu der Partition hinzugefügt werden. Das Problem wird in einem zweiten Durchlauf gelöst, in dem kleine Partitionen mit angrenzenden Partitionen verschmolzen werden.

Generierung des endgültigen Pfades

Nachdem ein Pfad zwischen den Partitionen gefunden wurde, schließt sich die Suche auf der Ebene der Zellen an. Aus den Partitionen müssen zunächst Zellen als Wegpunkte für die weitere Suche generiert werden. Zwei Möglichkeiten dazu wurden untersucht.

Das erste Verfahren (siehe Abbildung 3.16) verwendet die Zellen in der Mitte der Partitionen als Wegpunkte. Zwischen diesen Zellen werden die Teilabschnitte des endgültigen Weges gesucht. Allerdings sind diese Pfade noch unnatürlich eckig. Es erfolgt eine Glättung, die zu den roten Abkürzungen führt.

Das zweite Verfahren sucht zwischen den Zellen die an der Grenze zweier Partitionen liegen. Diese sind in Abbildung 3.17 grau gefärbt. Hier ist keine weitere Glättung nötig.

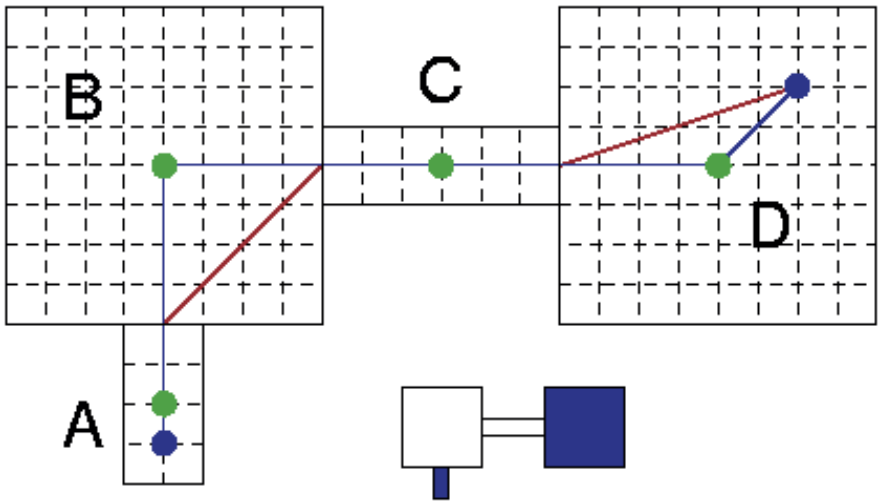

Abbildung 3.16: Suche zwischen den Mittelpunkten der Partitionen

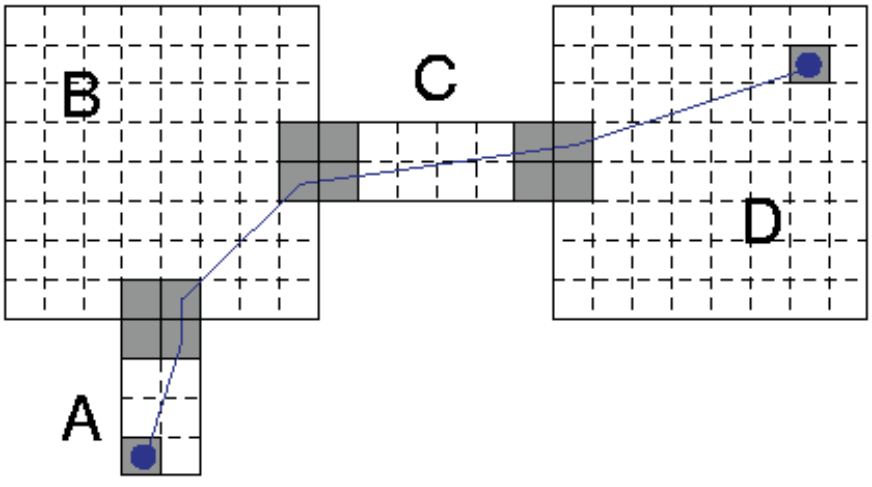

Abbildung 3.17: Suche zwischen den Grenzen der Partitionen