

MaxControl – ein objektorientiertes Werkzeug zur automatischen Erstellung von 3D-Computeranimationsfilmen und dessen Integration in eine professionelle 3D-Animationssoftware

Dissertation
zur Erlangung des Doktorgrades (Dr. rer. nat.)
der Technischen Fakultät
der Christian-Albrechts-Universität
zu Kiel

vorgelegt von
Dipl.-Inf. JAN PAUL

Kiel,
24.01.2008

1. Gutachter: Prof. Dr. Gerhard Weber

2. Gutachter: Prof. Dr. Peter Kandzia

Datum der mündlichen Prüfung: 4.1.2008

Danksagung:

Mein Dank gilt Herrn Professor Dr. Weber für die Betreuung dieser Arbeit, die Unterstützung auf fachlichem Gebiet und wertvolle Hilfestellungen bezüglich der Literaturlarbeit. Ich danke Herrn Professor Dr. Kandzia für die Ermöglichung einer Promotion im Bereich Multimedia an seinem Lehrstuhl für Datenbanksysteme und für die Zeit, die er auch nach seiner Emeritierung regelmäßig für die Betreuung dieser Arbeit aufgewendet hat. Bei Herrn Dr. Simon bedanke ich mich für die Möglichkeit, durch seine Vermittlung und Betreuung im Bereich 3D-Computeranimation arbeiten zu können, beginnend bei meiner Studienarbeit, über meine Diplomarbeit bis hin zu dieser Doktorarbeit. Auf dem Gebiet der Physik, deren Teilbereich Mechanik in dieser Arbeit eine wichtige Rolle spielt, stand mir Herr Dr. Härtel, wie schon bei meiner Studien- und Diplomarbeit, stets beratend zur Seite und lieferte auch die Aufgabenstellungen für einige Beispielanwendungen dieser Arbeit zum Thema „Didaktik der Physik“.

Inhaltsverzeichnis

| | | |
|-------|--|-----|
| 1 | Einleitung | 7 |
| 1.1 | EINORDNUNG DIESER ARBEIT IN FILMGENRES UND FILMPRODUKTIONSTECHNIKEN..... | 7 |
| 1.2 | 3D-ANIMATIONSPROGRAMME IM VERGLEICH MIT DIESER ARBEIT..... | 17 |
| 1.3 | THESEN..... | 20 |
| 2 | Aufgabenstellung und Zusammenfassung dieser Arbeit..... | 21 |
| 2.1 | AUTOMATISIERUNG DER ERSTELLUNG VON ANIMATIONEN: VORSTELLUNG DES ENTWICKELTEN LÖSUNGSKONZEPTES | 21 |
| 2.2 | ARCHITEKTUR VON MAXCONTROL | 33 |
| 3 | Begriffsdefinitionen..... | 38 |
| 3.1 | 3D-OBJEKT, OBJEKT..... | 38 |
| 3.2 | TRANSFORMATION..... | 38 |
| 3.3 | 3D-SZENE, SZENE, SZENENZUSTAND..... | 39 |
| 3.4 | 3D-ANIMATION, ANIMATION, ANIMIERBAR, FRAME | 44 |
| 3.5 | KEY, ANIMATIONS-KEY, KEYFRAME-ANIMATION | 44 |
| 3.6 | RENDERING | 47 |
| 4 | Anforderungen an MaxControl auf der Basis des aktuellen Stands der Technik..... | 49 |
| 4.1 | EXISTIERENDE UND MÖGLICHE TECHNIKEN ZUR AUTOMATISCHEN ERSTELLUNG VON ANIMATIONEN ... | 49 |
| 4.1.1 | SYSTEME ZUR ANIMATION NATÜRLICHER PHÄNOMENE..... | 49 |
| 4.1.2 | PARTIKELSYSTEME..... | 50 |
| 4.1.3 | SIMULATION VON MECHANIK | 54 |
| 4.1.4 | TOOLS ZUR AUTOMATISCHEN ANIMATION VIELER 3D-FIGUREN | 57 |
| 4.1.5 | PLUGINS ALS MÖGLICHE TECHNIK | 59 |
| 4.1.6 | SKRIPTSPRACHEN | 59 |
| 4.1.7 | PROPRIETÄRE SPEZIELLE LÖSUNGEN..... | 60 |
| 4.1.8 | JAVA ALS MÖGLICHE TECHNIK..... | 61 |
| 4.1.9 | ZUSAMMENFASSENDE BETRACHTUNG WEITERER NACHTEILE BISHER EXISTIERENDER TECHNIKEN | 62 |
| 4.2 | VERGLEICH VON MAXCONTROL MIT TECHNIKEN AUS DEM BEREICH DER COMPUTERSPIELE..... | 63 |
| 4.3 | INTEGRATION VON MAXCONTROL IN INDUSTRIELL ÜBLICHE PRODUKTIONSPROZESSE | 65 |
| 5 | Überblick über die in MaxControl verwendeten Werkzeuge..... | 68 |
| 5.1 | BETRIEBSSYSTEM MICROSOFT WINDOWS XP | 68 |
| 5.2 | 3D-ANIMATIONSWERKZEUG 3D-STUDIO-MAX | 68 |
| 5.3 | TONERZEUGUNG MIT FOLEY STUDIO MAX..... | 70 |
| 5.4 | RENDERING MIT FINAL RENDER | 71 |
| 5.5 | VOLUMETRISCHES RENDERING MIT AFTERBURN..... | 71 |
| 5.6 | MECHANIKSIMULATION MIT MACROMEDIA SHOCKWAVE | 72 |
| 5.7 | DEFINITION VON VERHALTENSWEISEN MIT JAVA | 74 |
| 5.8 | TRANSFORMATIONEN MIT JAVA-3D | 74 |
| 5.9 | MICROSOFT INTERNET EXPLORER ALS BASIS-SOFTWARE | 75 |
| 6 | Simulationskonzept von MaxControl..... | 76 |
| 6.1 | VEREINFACHENDE ANNAHMEN | 76 |
| 6.2 | BASISPRINZIPIEN..... | 77 |
| 6.3 | SIMULATIONSEIGENSCHAFTEN (SPs)..... | 78 |
| 6.4 | ZWEI DATENEbenen FÜR DIE SIMULATION | 81 |
| 6.5 | ZUSTANDSÜBERGÄNGE IM VERLAUF DER SIMULATION | 82 |
| 6.6 | MANUELLE KONTROLLE VON SIMULATIONSEIGENSCHAFTEN | 87 |
| 6.7 | ZUSTANDSÜBERGÄNGE UNTER BERÜCKSICHTIGUNG DER MANUELLEN KONTROLLE..... | 91 |
| 7 | Datenmodell und Designrichtlinien..... | 95 |
| 7.1 | EINFÜHRUNG | 95 |
| 7.2 | PRINZIPIEN FÜR DEN AUFBAU VON OBJEKTYPEN UND VERHALTENSWEISEN | 99 |
| 7.2.1 | RICHTLINIEN FÜR DIE STRUKTURIERUNG VON OBJEKTYPEN | 100 |
| 7.2.2 | WIEDERVERWENDUNG VON VERHALTENSWEISEN | 103 |
| 7.2.3 | VERERBUNG VON VERHALTENSWEISEN UND OBJEKTYPEN | 104 |
| 7.2.4 | EINBINDEN VON SIMULATIONSEIGENSCHAFTEN..... | 107 |
| 7.3 | NAMENSKONVENTIONEN FÜR KLASSENBEZEICHNER UND DABEI RELEVANTE BASISKLASSEN | 111 |
| 8 | Grundaufbau wichtiger Klassen, Möglichkeiten für deren Nutzung und Erweiterung..... | 113 |
| 8.1 | MC_ENTITY | 113 |
| 8.1.1 | INFORMATIONEN ÜBER DIREKTE UND INDIREKTE BESITZER | 114 |

| | | |
|----------|---|-----|
| 8.1.2 | BERÜCKSICHTIGUNG UND STEUERUNG DES ZEITABLAUFS | 114 |
| 8.1.3 | ZUGRIFF AUF DIE SIMULATIONSOBJEKTE EINER SZENE | 116 |
| 8.2 | MC_PROPERTY | 116 |
| 8.2.1 | ZUORDNEN EINER SIMULATIONSEIGENSCHAFT ZU IHREM BESITZER | 117 |
| 8.2.2 | ZUGRIFF AUF DIE WERTE VON SIMULATIONSEIGENSCHAFTEN | 119 |
| 8.3 | MC_OWNER | 120 |
| 8.3.1 | DYNAMISCHE BINDUNG VON SPs UND VHs AN IHREN BESITZER | 120 |
| 8.3.2 | INITIALISIERUNG VON VERHALTENSWEISEN UND FESTLEGUNG VON STANDARDWERTEN FÜR SIMULATIONSEIGENSCHAFTEN | 123 |
| 8.3.3 | BAUMSUCHE IN DER ZUGEHÖRIGEN SIMULATIONSOBJEKT-HIERARCHIE NACH VERHALTENSWEISEN UND SIMULATIONSEIGENSCHAFTEN | 124 |
| 8.3.4 | ZUGRIFFSPFADE FÜR SIMULATIONSEIGENSCHAFTEN UND VERHALTENSWEISEN | 134 |
| 8.3.5 | BEQUEMERE ZUGRIFF AUF DIE WERTE VON SIMULATIONSEIGENSCHAFTEN | 138 |
| 8.3.6 | SUCHE NACH KINDOBJEKTEN UND DEREN VERHALTENSWEISEN | 140 |
| 8.3.7 | BEEINFLUSSUNG DER MECHANIKSIMULATION UND LESEN VON KOLLISIONSEREIGNISSEN | 141 |
| 8.4 | MC_OBJECT | 143 |
| 8.4.1 | INFORMATIONEN ÜBER DIE SZENENHIERARCHIE | 144 |
| 8.4.2 | AUFRUF DER VERHALTENSWEISEN EINES SIMULATIONSOBJEKTES | 145 |
| 8.4.3 | ZUGRIFF AUF DIE ERSTELLTEN FEDERN | 145 |
| 8.5 | MCO_DYNAMIC | 145 |
| 8.5.1 | ZUGRIFF AUF DIE TRANSFORMATIONSEIGENSCHAFTEN EINES SIMULATIONSOBJEKTES | 146 |
| 8.6 | MC_BEHAVIOUR | 146 |
| 8.6.1 | AKTIVIEREN UND DEAKTIVIEREN VON VERHALTENSWEISEN | 146 |
| 8.6.2 | ZUORDNEN EINER VERHALTENSWEISE ZU IHREM BESITZER | 147 |
| 8.6.3 | DEFINITION EINER VERHALTENSWEISE | 149 |
| 9 | Festlegung und Diskussion der Simulationsvorschrift | 161 |
| 9.1 | PRINZIPIEN | 161 |
| 9.2 | AUSFÜHRUNGSREIHENFOLGE DER VERHALTENSWEISEN | 161 |
| 9.3 | SIMULATIONSZYKLUS | 168 |
| 9.4 | PROBLEME UND BESONDERHEITEN DER MAXCONTROL-SIMULATION SOWIE DARAUS RESULTIERENDE DESIGN-VORSCHRIFTEN | 173 |
| 9.4.1 | ZUR WIEDERHOLBARKEIT VON SIMULATIONEN | 173 |
| 9.4.2 | AUS DEN ANFORDERUNGEN RESULTIERENDE DESIGNVORSCHRIFTEN | 174 |
| 9.4.3 | WIRKUNG DER AUSFÜHRUNGSREIHENFOLGE DER VHs AUF DAS SIMULATIONSERGEBNIS | 175 |
| 9.4.4 | AUSFÜHRUNGSREIHENFOLGE DER SOS AUFGRUND VON ELTERN-KIND-BEZIEHUNGEN DES SZENENGRAPHEN | 179 |
| 9.4.5 | DESIGNVORSCHRIFTEN BEZÜGLICH DER AUSFÜHRUNGSREIHENFOLGE VON VERHALTENSWEISEN | 181 |
| 9.4.6 | SIMULATION NACH DEM GESAMTSCHRITTVERFAHREN ALS MÖGLICHE ALTERNATIVE | 183 |
| 9.5 | SIMULATION VON MECHANIK ALS TEILBEREICH DER PHYSIK | 184 |
| 10 | Die technische Umsetzung | 186 |
| 10.1 | TECHNISCHE DETAILS WICHTIGER KLASSEN | 186 |
| 10.1.1 | MC_PROPERTY: DEFINITION VON GUI-REPRÄSENTATIONEN FÜR SPs | 186 |
| 10.1.2 | MC_OWNER: ZUGRIFF AUF DIE TRANSFORMATIONEN VON SIMULATIONSOBJEKTEN | 187 |
| 10.1.3 | MCH_TRANSFORM3D (TRANSFORMATIONEN) | 188 |
| 10.1.4 | MCH_SPRING (FEDERN) | 188 |
| 10.1.5 | MCH_COLLISIONINFO (INFORMATIONEN ÜBER KOLLISIONEN) | 189 |
| 10.2 | SIMULATION | 190 |
| 10.2.1 | VORBEREITENDER SCHRITT BEZÜGLICH DER SIMULATIONSEIGENSCHAFTEN | 191 |
| 10.2.2 | SIMULATIONSZYKLUS AUS TECHNISCHER SICHT MIT ZEITABLAUFKONTROLLE UND EINBINDUNG DER MECHANIKSIMULATION | 191 |
| 10.2.3 | TECHNIK DER SIMULATION VON MECHANIK | 195 |
| 10.2.3.1 | VORGEHENSWEISE DER MECHANIKSIMULATION | 195 |
| 10.2.3.2 | PROBLEME UND PROBLEMLÖSUNGEN BEI DER SIMULATION VON MECHANIK | 197 |
| 10.2.3.3 | SIMULATION VON FAHRZEUGPHYSIK | 202 |
| 10.2.4 | AUTOMATISCHE ERZEUGUNG VON TÖNEN | 204 |
| 10.3 | EINBINDUNG DER VERWENDETEN WERKZEUGE | 205 |
| 10.3.1 | KOMMUNIKATION MIT 3D-STUDIO-MAX UND IHRE OPTIMIERUNG | 205 |
| 10.3.1.1 | PRINZIPIEN DER OPTIMIERUNG | 205 |
| 10.3.1.2 | ZUSTANDSÜBERGÄNGE BEI DER KOMMUNIKATIONSOPTIMIERUNG | 210 |
| 10.3.1.3 | ALGORITHMEN ZUR KOMMUNIKATIONSOPTIMIERUNG MITTELS KEY-TECHNIK | 213 |
| 10.3.2 | KOMMUNIKATIONSMETHODEN ZUR SICHERUNG DER SKALIERBARKEIT | 231 |

| | | |
|----------|---|-----|
| 10.4 | PRAKTISCHE ANWENDUNGEN DES ENTWICKELTEN WERKZEUGS | 232 |
| 10.4.1 | ANIMATIONSFILME ZUM THEMA „VISUELLE EFFEKTE IN SPIELFILMEN“ | 232 |
| 10.4.1.1 | AUTOMATISIERTE ANSTEUERUNG VIELER LICHTQUELLEN | 232 |
| 10.4.1.2 | ANIMATION SECHSBEINIGER ROBOTER | 233 |
| 10.4.1.3 | SIMULATION EINER GRÖßEREN ANZAHL VON ROBOTERN | 234 |
| 10.4.1.4 | SIMULATION VON MECHANIK AUF DER BASIS DER ANIMATION SECHSBEINIGER ROBOTER | 234 |
| 10.4.1.5 | SIMULATION EINES AUTORENNENS | 235 |
| 10.4.1.6 | FAHRZEUG UND ROBOTER IN EINER SIMULATION | 238 |
| 10.4.2 | ANIMATIONSFILME ZUM THEMA „DIDAKTIK DER PHYSIK“ | 239 |
| 10.4.2.1 | AUTOMATISCHE ANIMATION DATENREPRÄSENTIERENDER OBJEKTE | 239 |
| 10.4.2.2 | SIMULATION VON BILLARDKUGELN | 240 |
| 10.4.2.3 | SIMULATION VON SONNE, ERDE UND MOND | 241 |
| 10.4.2.4 | DARSTELLUNG VON POTENTIALVERLÄUFEN IN SCHALTUNGEN | 242 |
| 10.5 | GEPLANTE WEITERENTWICKLUNGEN VON MAXCONTROL | 243 |
| 11 | Diskussion | 246 |
| 12 | Abbildungsverzeichnis | 250 |
| 13 | Stichwortverzeichnis | 252 |
| 14 | Literaturverzeichnis | 254 |
| 15 | Auszug aus der Klassenstruktur von MaxControl | 270 |
| 16 | Anlagen | 275 |

1 Einleitung

1.1 Einordnung dieser Arbeit in Filmgenres und Filmproduktionstechniken

Filme sind ein wichtiger Bestandteil unserer Kultur geworden. Sie werden für verschiedene Zwecke eingesetzt, z.B. in Form von Lehrfilmen, um Lerninhalte über die Möglichkeiten von Büchern und einzelnen Bildern hinaus zu vermitteln. Sie werden ebenso zum Erzählen von Geschichten oder zur Dokumentation aktueller und vergangener Ereignisse verwendet. Dies sind nur einige wichtige Beispiele für die Einsatzmöglichkeiten von Filmen.

Technisch können Filme als eine Weiterentwicklung der Fotografie gesehen werden. Indem mehrere Einzelbilder pro Sekunde in festen zeitlichen Abständen aufgenommen werden, können so festgehaltene Ereignisse durch das Abspielen dieser Einzelbilder in der korrekten Reihenfolge und Geschwindigkeit zu einem späteren Zeitpunkt visuell wiedergegeben werden. Die naheliegendste Verwendung für Filme ist also das visuelle Speichern und Wiedergeben von Ereignissen und Abläufen.

Da das Sehen einer der wichtigsten Sinne des Menschen ist, können Filme einen wichtigen Teil dessen, was Menschen in ihrem Umfeld an Geschehnissen wahrnehmen, überzeugend wiedergeben. Da ein wichtiger weiterer Sinn des Menschen das Gehör ist, folgte als logische Weiterentwicklung des Films die gleichzeitige Aufnahme bzw. Wiedergabe von Tonspuren. Der so entstandene Tonfilm ist auch heute noch die Basistechnologie für die meisten Filme, auch wenn sich Aufnahme- und Wiedergabeverfahren ständig weiterentwickelt haben. Die anfänglich genutzten Schwarz-Weiß-Aufnahmen wurden durch Farbbilder ersetzt, so entstand der Farbfilm. Der wichtigste neuere Schritt war sicherlich der Übergang von analoger zu digitaler Speicherung und Bearbeitung von Filmen (siehe auch [Giesen00k]), was sich unter anderem am Erfolg des Mediums DVD für Filme zeigt. Bei den Tonspuren gab es eine Entwicklung von Mono- über Stereo- bis hin zu Surround-Tonspuren (siehe [Holman00]) für mehrere im Raum angeordnete Lautsprecher, die ein räumliches Klangerempfinden ermöglichen. Analog dazu wurden und werden verschiedene Verfahren zur Ermöglichung des räumlichen Sehens von Filmen entwickelt (siehe [Umble06]).

An der Weiterentwicklung des Films für weitere Sinne des Menschen wird weiterhin gearbeitet. Es gibt Versuche für den Geruchssinn (z.B. [Naka06]), und von Jahrmärkten und Vergnügungsparks sind Simulatoren bekannt, die synchron zu einem Tonfilm passende Bewegungen des Zuschauerraums erzeugen, um so die zusätzliche Illusion von Bewegung für den Zuschauer zu erzeugen, so genannte *Motion-Rides* (siehe [Kanev02, Moriya01]).

Eine einführende Darstellung zum Medium Film und seiner technischen Entwicklung gibt [Borstnar02].

Das übliche Vorgehen im Umgang mit Filmen ist, die Einzelbilder in einer bestimmten Geschwindigkeit aufzunehmen, der so genannten Framerate, die in Einzelbildern pro Sekunde angegeben wird, und sie dann in derselben Framerate wieder abzuspielen. So entsteht für das menschliche Auge eine Illusion, die gefilmte Vorgänge scheinbar wiederholt. Der Film bleibt jedoch eine Illusion, zumal es nicht zwingend erforderlich ist, die Einzelbilder in der Geschwindigkeit oder Reihenfolge aufzunehmen, in der sie später wiedergegeben werden.

Häufig verwendet wird z.B. ein Filmerzeugungsverfahren, das einzelne Bilder in beliebigen Zeitabständen entgegennehmen kann, die dann in einer festen Framerate abgespielt werden

können. Eine der einfachsten relativ früh dazu einsetzbaren Techniken ist eine Filmkamera, die per Knopfdruck jeweils nur ein einzelnes Bild aufnimmt.

Nimmt man mit einer solchen Kamera z.B. Einzelbilder eines realen Modells auf, das über Gelenke verfügt, die zwar beweglich sind, aber selbständig ihre aktuelle Stellung halten können, so kann das Modell in verschiedene Bewegungsposen gebracht werden, die dann mit der Kamera einzeln aufgenommen werden. Es ist nun möglich, so nacheinander Posen an dem Modell einzustellen und aufzunehmen, dass der entstehende Film in einer konstanten Framerate abgespielt den Eindruck einer fließenden Bewegung erzeugt. Diese Technik wird als Stop-Motion (siehe [Hubb03]) bezeichnet.

Es folgen als Beispiel einige Einzelbilder eines Stop-Motion-Films:

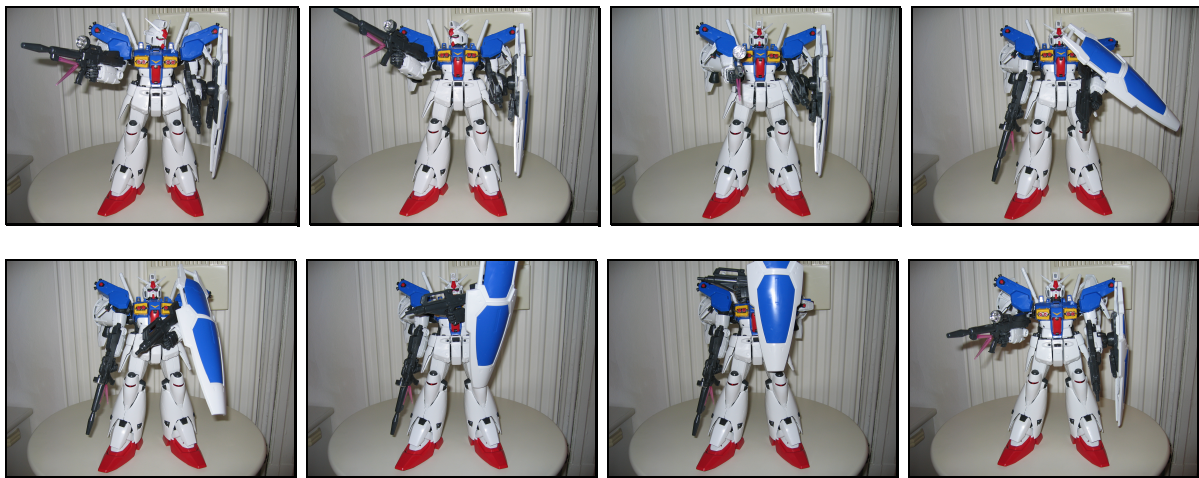



Abbildung 1: Stop-Motion-Film

Der zugehörige Film ist auf der beiliegenden DVD in folgendem Pfad zu finden:

 Movies\Gundam_FB_1_V2_10fps.avi

Dieser durch Stop-Motion entstandene Film enthält 108 Einzelbilder einer bewegbaren realen Figur. Die Einzelbilder werden mit einer Framerate von 10 Bildern pro Sekunde abgespielt, um den Eindruck einer Bewegung der Figur zu erzeugen. Dadurch ergibt sich eine Filmdauer von etwa 10 Sekunden.

Stop-Motion-Filme können durch digitale Verfahren verbessert werden. Durch die Aufnahme einzelner Bilder ohne Bewegung während der Bildbelichtung entsteht zusammen mit in der Regel recht niedrigen Bildraten bei dieser Technik oft der Eindruck von nicht fließenden ruckartigen Bewegungen. [Bros01] entwickelte dafür ein System, das die Änderungen zwischen zwei Stop-Motion-Bildern durch Computer-Vision-Techniken erkennen und auf dieser Basis den Einzelbildern nachträglich Bewegungsunschärfe hinzufügen kann. Bewegungsunschärfe entsteht bei herkömmlichen Filmaufnahmen durch die Einzelbildbelichtungszeit automatisch und erzeugt somit einen flüssigeren Eindruck besonders bei schnellen Bewegungen. Zu Bewegungsunschärfe siehe auch Kap. 10.2.3.2.

Ein mit Stop-Motion vergleichbares Verfahren ist der Zeichentrick. Hier werden nicht reale Objekte aufgenommen, sondern gezeichnete Einzelbilder. Zeichentrick hat den Vorteil, dass die darzustellenden Welten und Objekte nicht real existieren müssen, um sie z.B. wie beim Stop-Motion-Verfahren aufzunehmen. Die Möglichkeiten des Darstellbaren erhöhen sich dadurch, allerdings auf Kosten des Herstellungsaufwandes, da jedes Einzelbild gezeichnet

werden muss. Durch das Übereinanderlegen durchsichtiger bemalter Folien kann dieses Verfahren jedoch vereinfacht werden, da Hintergründe dann beispielsweise für mehrere Einzelbilder unverändert bleiben können und nur bewegte Objekte im Vordergrund für jedes Einzelbild neu gezeichnet werden müssen (siehe dazu [Jack06]). Zum Zeichnen der Bilder und zum Übereinanderlegen solcher Ebenen werden zunehmend Computerprogramme verwendet, die den Arbeitsaufwand ebenfalls senken können [Hubb03].

Seit 3D-Modelle von Körpern durch einen Computer ausreichend realistisch als Bild dargestellt werden können und solche computergenerierten Bilder als Film abgespielt werden können, wurden **3D-Computeranimationsfilme** erzeugt, denen das **Thema dieser Arbeit** zuzuordnen ist. 3D-Computeranimationsfilme sind technisch mit Zeichentrickfilmen vergleichbar, da auch hier Einzelbilder ohne die direkte Verwendung realer Objekte künstlich erzeugt werden. Vom Computer auf der Basis von 3D-Modellen erzeugte Bilder haben gegenüber Zeichentrick den Vorteil, dass ein im Computer gespeichertes 3D-Modell aus beliebigen Blickwinkeln automatisch neu dargestellt werden kann, wogegen ein Künstler bei der Erstellung eines Zeichentrickfilms für jeden Blickwinkel ein eigenes Bild zeichnen müsste. Kamerafahrten oder Objektbewegungen sind also mit Hilfe von 3D-Computeranimationen weitaus einfacher herstellbar als im Zeichentrick-Verfahren. Auf der Basis der physikalischen Gesetze der Optik existieren für die meisten bekannten optischen Phänomene wie Beleuchtung, Schatten, Spiegelung, Lichtbrechung, Tiefenunschärfe und Bewegungsunschärfe mathematische Modelle. Mit Hilfe dieser Modelle oder mit vereinfachten Verfahren kann im Computer für ein 3D-Modell automatisch ein sehr realistisches Bild des darzustellenden Körpers erzeugt werden. Je komplexer die erforderlichen Berechnungen für ein Phänomen der Optik jedoch sind, desto mehr Rechenzeit wird für ein Einzelbild bei der Umsetzung dieses Phänomens benötigt.

Die Berechnung spezieller Phänomene kann so aufwändig sein, dass diese bisher noch gar nicht oder erst seit kurzem berücksichtigt werden. Zu solchen erst jetzt zunehmend in 3D-Animationsprogrammen berechenbaren Phänomenen gehört z.B. die Lichtausbreitung in halbtransparenten Volumina, wie z.B. der menschlichen Haut, das so genannte *Sub-Surface-Light-Scattering* (siehe [Jens02]). Ein weiteres solches Problem ist die komplexe Ausbreitung von Photonen im Raum durch diffuse Lichtreflexion an Oberflächen, was auch bei nur einer einzigen Lichtquelle zu einer Beleuchtung von Stellen im Raum führen kann, die eigentlich im Schatten dieser Lichtquelle liegen. Die Umsetzung dieses Phänomens für Computergrafiken wird als *Global Illumination* bezeichnet. Nach [Bung02] verwendeten erste Ansätze für die Berücksichtigung der Globalen Illumination das *Raytracing*, um nicht diffuse Reflexion von Licht an glatten Oberflächen und die Lichtbrechung in durchsichtigen Körpern wie Glas zu simulieren, siehe z.B. die Arbeit [Whit80]. Die Lichtreflexion an diffusen Oberflächen wurde jedoch erst durch *Radiosity*-Lösungen möglich [Bung02], die nach [Foley97] zuerst von [Nish85] und [Gora84] entwickelt wurden. Nach [Bung02] konnten jedoch so genannte Kaustiken erst durch die Arbeit [Jens96] berechnet werden. Kaustiken sind Lichtmuster, die durch gekrümmte spiegelnde oder transparente Körper durch Bündelung des Lichtes auf diffus reflektierenden Oberflächen entstehen, wie z. B. die beweglichen Lichtmuster am Boden eines Swimmingpools, die durch Wellen auf der Wasseroberfläche hervorgerufen werden, die wie dynamisch veränderliche Linsen wirken. Da schon ein bewegtes Objekt die globale Beleuchtung aller anderen Objekte im Raum ändern kann, ist die Berechnung von Global Illumination besonders aufwändig bei Animationen, in denen sich nicht nur die Kamera, sondern auch Objekte bewegen. Denn dann muss die globale Beleuchtung für jedes Einzelbild erneut berechnet werden. Um Rechenzeit einzusparen, lässt [Yee01] dabei auf der Basis bestimmter Einschränkungen des menschlichen Sehverhaltens Berechnungsfehler zu, die einem Betrachter weniger auffallen. [Mysz02] wählt einen vergleichbaren Ansatz, um mit einer Messfunktion für die visuelle Qualität einer Animation

und mit Hilfe von bildbasierten Rendering-Techniken sogar eine Echtzeitleistung für Animationen mit Global Illumination zu erreichen. [Tole02] ermöglicht globale Beleuchtung in Echtzeit, indem die Berechnungsergebnisse eines aufwändigeren Rendering-Systems zwischengespeichert werden, um darauf basierend mit Hilfe von Grafikkhardware Szenen mit globaler Beleuchtung darzustellen. Dabei werden auch bewegte Lichtquellen und Objekte unterstützt. Auch [Ward99] bietet eine Echtzeitleistung, die auch nicht-diffuse Spiegelungen und Lichtbrechungen unterstützt.

Computergrafiken sind Zeichentrickdarstellungen in ihrem Realismus mittlerweile in der Regel überlegen, da die entsprechenden Verfahren automatisch realistische Bilder und daraus aufgebaute Filme erzeugen können, während bei Zeichentrickfilmen zumindest für die bewegten Objekte aufgrund des nötigen Aufwandes meistens nur relativ vereinfachte Darstellungen gezeichnet werden.

Computer bieten mit Hilfe entsprechender *Compositing*-Software (siehe Kap. 4.3) vielfältige Möglichkeiten zur Kombination von Bildern und Filmen aus verschiedensten Quellen. Durch ältere Techniken, die ohne den Einsatz von Computern auskommen, waren auch früher schon unterschiedliche Arten der Kombination von Filmmaterialien und Bildquellen möglich, wie z.B. durch Rückprojektion, Travelling Mattes, Blue-Screen-Verfahren über einen optischen Printer sowie Front- oder Aufprojektion [Giesen00e] oder Spiegeltricks [Giesen00s]. Dadurch können die hier angesprochenen Filmtechniken relativ frei miteinander kombiniert werden.

So werden Zeichentrickfilme zunehmend durch Computeranimationen ergänzt, z.B. wurde eine große flüchtende Herde in „*König der Löwen*“ (1994) durch eine automatisierte Computeranimation umgesetzt [Guzdial06, Sullivan99]. Es zeichnet sich durch Filme wie „*Ice Age 1+2*“ (2002, 2006), „*Findet Nemo*“ (2003), „*Madagaskar*“ (2005) und „*Final Fantasy VII: Advent Children*“ (2005) ab, dass Zeichentrickfilme in Zukunft weitgehend durch 3D-Computeranimationsfilme ersetzt werden. Dabei können die Übergänge sehr fließend sein, statt z.B. Computeranimationen in Zeichentrickfilme einfließen zu lassen, können umgekehrt auch Zeichentrick-Komponenten in Computeranimationen eingebunden werden. Bei „*Macross Zero*“ (2002) werden beispielsweise im Zeichentrickverfahren animierte Darstellungen von Menschen in 3D-Computeranimationen eingesetzt, z.B. gezeichnete Piloten im Cockpit eines als 3D-Computeranimation realisierten Flugzeugs in einer ebenfalls computergenerierten Umgebung.

Damit Computeranimationen, die in Zeichentrickfilme eingefügt werden, sich nicht zu auffällig von den im herkömmlichen Zeichentrickverfahren realisierten Teilen unterscheiden und auch bei weitgehender Verwendung von Computeranimation der visuelle Charakter eines Zeichentrickfilms nicht verloren geht, wurden Darstellungstechniken entwickelt, die eine Computergrafik wie eine typische manuell erstellte Zeichentrickgrafik aussehen lassen. Diese Technik wird als *Cel-Shading* bezeichnet und kommt wegen ihres ansprechenden Stils sogar in Computer- und Videospiele zum Einsatz. Bei der Verwendung des *Cel-Shading* bleiben Vorteile der 3D-Computeranimationen wie weiche und perspektivisch korrekte Bewegungen von Objekten erhalten. Cel-Shading ist eine Form des nicht-photorealistischen Rendering, für das es verschiedene Anwendungen und Varianten gibt. [Kow99] entwickelt eine Methode, mit der auf der Basis von 3D-Modellen Fell, Gras und Bäume durch Rendering so dargestellt werden können, dass sie wie gezeichnet wirken. Dabei soll die Komplexität der Szene, also viele Haare bei Fell, viele Grashalme bei Gras und viele Blätter bei Bäumen nur durch wenige gut platzierte Striche angedeutet werden, in Anlehnung an die typische vorgehensweise von Comiczeichnern. In [Praun01] werden 3D-Objekte mit künstlich erzeugten Strichen dem Lichteinfall nach schattiert, so wie Schattierungen z. B. in Bleistiftzeichnungen realisiert werden können. [Petrov00] ermöglicht die automatische Erzeugung von Schatten, die eine

handgezeichnete Figur auf eine 3D-Szene wirft, die ebenfalls auf der Basis von Handzeichnungen entsteht. [Corr98] entwickelte ein System, das handgezeichnete Objekte nachträglich mit einer Textur versehen kann, indem ein texturiertes dreidimensionales Grundmodell entsprechend der aktuellen Form der Handzeichnung deckungsgleich verformt und dann mit seiner Textur dargestellt wird. Einen Überblick über solche Techniken gibt [Stroth02].

Auch Filme, die auf Aufnahmen realer Schauspieler oder Umgebungen basieren, so genannte *Live-Action-Filme*, werden mit Stop-Motion, Zeichentrick und zunehmend auch mit Computeranimationen kombiniert. Der Film „*Falsches Spiel mit Roger Rabbit*“ (1988) kombiniert z.B. Live-Action mit hinzugefügten Zeichentrickfiguren, wobei die Interaktion der Zeichentrickfiguren mit der realen Welt und insbesondere den Schauspielern besonders kunstvoll realisiert wurde. Bei vielen Live-Action-Filmen des Genres Science-Fiction, wie z.B. den älteren¹ „*Star Wars*“ Episoden IV-VI² (siehe [Giesen00e]), wurden Spezialeffekte durch das dem Zeichentrick sehr verwandte *Rotoscoping* [Hubb03] hinzugefügt, um z.B. Energiestrahlen, Lichtschwerter oder Düsenstrahlen darzustellen. Der Roboter ED-209 wurde im Live-Action-Film „*Robocop*“ (1987) durch Stop-Motion-Techniken in den Film eingefügt [Giesen00]. Die neuesten Star-Wars-Filme und ähnliche neuere Science-Fiction-Filme wie „*Krieg der Welten*“ (2005) binden sehr intensiv 3D-Computeranimationen in die Live-Action-Sequenzen ein. Bei vielen neueren Filmen mit intensiver Einbindung visueller Spezialeffekte, insbesondere auch bei den neuesten Star-Wars-Filmen, kippt zunehmend die Richtung der technischen Kombination von Live-Action mit 3D-Computeranimationen so, dass eher Live-Action-Elemente in Computeranimationen eingebunden werden als umgekehrt. Diese Entwicklung wird auch in [Giesen00k, Giesen00e] beobachtet. Bei dem Film „*Sky Captain and the World of Tomorrow*“ (2004) sind lediglich die Schauspieler real, sämtliche Umgebungen entstanden als Computeranimationen, in welche Filmsequenzen der Schauspieler eingefügt wurden. Bei diesem Vorgehen sind spezielle Probleme bei der Beleuchtung zu beachten. Während bei der Einbindung von Computeranimationen in Live-Action-Szenen die virtuelle Beleuchtung der Computergrafiken den Beleuchtungsverhältnissen der Live-Action-Szene angepasst werden muss, ist es umgekehrt erforderlich, bei einer Integration von gefilmten Schauspielern in virtuelle Umgebungen die Beleuchtung der Schauspieler an die Beleuchtung der Computergrafik-Szene anzupassen. Die Arbeit [Debev02] bietet dafür eine interessante Lösung, indem die Lichtverhältnisse einer virtuellen Umgebung durch eine mit hellen roten, grünen und blauen LEDs bestückte Hohlkugel reproduziert werden, um so einen Schauspieler in der Mitte dieser Kugel gemäß den Lichtverhältnissen der virtuellen Szene zu beleuchten.

Es ist also nicht für jeden Film entscheidbar, mit welcher der hier angesprochenen Techniken er produziert wurde, da die Übergänge mittlerweile sehr fließend sind und die meisten Techniken sehr gut miteinander kombiniert werden können. Die hier angesprochenen Techniken repräsentieren außerdem bei weitem nicht alle verfügbaren Techniken in all ihren Abstufungen und Weiterentwicklungen, sondern sollen eher zeigen, aus welchen älteren Techniken sich die 3D-Computeranimation entwickelt hat.

Die Techniken lassen sich auch schwer Filmgenres zuordnen, da fast jede Technik in fast jedem Genre Verwendung finden kann und auch die Genres selbst fließend ineinander

¹ Die Episoden I-III wurden später produziert (Erscheinungsjahre 1999-2005) und stützen sich zu einem weitaus größeren Teil auf Computeranimationen.

² „Krieg der Sterne“ war für die zuerst gedrehten Episoden IV bis VI (Erscheinungsjahre 1977-1983) der deutsche Haupttitel von „Star Wars“. Ab der später gedrehten Episode I wurde auch für die deutschen Versionen der englische Haupttitel „Star Wars“ übernommen. Offizielle Webseite: <http://www.starwars.com/>, 11.02.2007

übergehen. Beispiele für Misch-Genres sind z.B. Science-Fiction-Horrorfilm („*Alien – Das unheimliche Wesen aus einer fremden Welt*“ (1979, siehe auch [Giesen00])), Science-Fiction-Actionfilm („*Total Recall*“ (1990)), Science-Fiction-Komödie („*Spaceballs*“ (1987)), Action-Komödie („*Rush Hour*“ (1998)) und Grusel-Komödie („*Scary Movie*“ (2000), „*Meine teuflischen Nachbarn*“ (1989)). Zudem unterliegen Filmgenres nach [Borstnar02] einem ständigen Wandel. Dies erschwert zusätzlich eine Zuordnung von Filmen und umso mehr von Techniken zu einem bestimmten Genre.

Zeichentrick, Stop-Motion und Computeranimationen sind Techniken, mit denen phantastische Objekte und Umgebungen dargestellt werden können. Abgesehen vom Zeichentrick werden diese Techniken deshalb tendenziell am häufigsten für Fantasy- und Science-Fiction-Filme verwendet, Zeichentrick wird dagegen häufig auch in Kinderfilmen eingesetzt. Jedoch können alle diese Techniken in fast jedem Genre Verwendung finden.

In Japan produzierte Zeichentrickfilme, so genannte Animes, werden dort mit einem anderen Selbstverständnis gesehen als westliche Zeichentrickfilme, die meistens Kinderfilme sind. Animes dagegen verwenden Zeichentrick lediglich als Technik mit einem allerdings künstlerisch sehr eigenen Stil, der sich stark von westlichen Zeichentrickfilmen unterscheidet. Dies zeigt, dass der Begriff „Zeichentrick“ nur als Filmtechnik, nicht aber als Filmgenre gesehen werden darf. Diese Ansicht wird auch in [Borstnar02] vertreten. Inhaltlich sind die Animes auf kein Genre beschränkt und die Verteilung auf die Genres ist in etwa mit der Verteilung westlicher Live-Action-Filme auf dieselben Genres vergleichbar. So gibt es bei Animes auch Jugendfreigaben bis „FSK18“ bzw. „keine Jugendfreigabe“. Bei Animes ist jedoch auch eine noch offenere Vermischung der Genres bemerkbar, so dass es dort viele Vertreter von Mischgenres gibt. Beispiele hierfür sind „*Chobits*“ (2002) (Science-Fiction-Liebesfilm), „*Martian Successor Nadesico*“ (1996) (Science-Fiction-Komödie), „*Das wandelnde Schloss*“ (2004) (Fantasy-Liebesfilm) und „*Hellsing*“ (2001) (Horror-Actionfilm), wobei es sich hierbei meistens nicht um abgeschlossene Filme sondern um Serien handelt, was ein weiteres typisches Merkmal für Animes ist. Zudem besitzen fast alle Animes ein komödiantisches Element.

Auch Computeranimationen können in fast jedem Genre Verwendung finden. Vorwiegend werden sie jedoch tatsächlich in Science-Fiction- und Fantasy-Filmen verwendet und haben dort die meisten älteren Techniken für Spezialeffekte wie Rotoscoping, Modelltricks (siehe [Giesen00e]), Puppenspiel und Stop-Motion weitgehend verdrängt. Sie sind damit für die Filmindustrie im Bereich „visuelle Spezialeffekte“ sehr wichtig geworden.

Aber auch Dokumentationen und Spielfilme mit historischem Hintergrund verwenden zunehmend Computeranimationen, um vergangene Welten und Ereignisse überzeugend darzustellen. So wird im Film „*Gladiator*“ (2000) (siehe auch [Giesen00]) das Leben im alten Rom dargestellt, wobei Gebäude als intakt gezeigt werden, die heute höchstens noch als Ruinen existieren. Der Film „*Troja*“ (2004) zeigt die Eroberung Trojas und in „*Pearl Harbor*“ (2001) (siehe auch [Giesen00]) wird der Angriff auf Pearl Harbor überzeugend dargestellt.

Ein weiteres Feld, in dem Computeranimationen zunehmend eingesetzt werden, sind Lehrfilme, die Lerninhalte veranschaulichen sollen. Wurden dazu in der Vergangenheit bisher häufig Zeichentricktechniken verwendet, um mathematische Formeln oder physikalische Gesetze und Vorgänge zu verdeutlichen, so werden auch in diesem Bereich zunehmend Computeranimationen eingesetzt, da sie z.B. durch das oben genannte *Cel-Shading* auch für abstrakte Darstellungen verwendet werden können. Aber auch realistische Darstellungen können hier sinnvoll sein, um z.B. im Bereich Biologie den Aufbau einer Zelle überzeugend darzustellen und die komplexen Vorgänge in einer Zelle anschaulich wiederzugeben.

Auch reine Actionfilme, die nicht der Science-Fiction zuzuordnen sind, können Computeranimationen verwenden, um beispielsweise Stunts zu ermöglichen, die in der Realität nicht machbar oder zu gefährlich wären. Ein Beispiel hierfür ist die letzte Verfolgungsjagd im Film „*Hart am Limit*“ (2004) (Originaltitel: „*Torque*“). Ebenso kann mit Hilfe von Computeranimationen die Zerstörung von Gebäuden und Bauwerken dargestellt werden, ohne diese Gebäude wirklich zu beschädigen oder Modelle zu verwenden. Ein Beispiel hierfür ist der einstürzende Staudamm in dem Fernsehfilm „*Die Todeswelle - Eine Stadt in Angst*“³ (2000). Dieser Film wird zwar dem Genre „Thriller“ zugeordnet, jedoch kann eine solche Technik auch sinnvoll in einem Actionfilm eingesetzt werden.

Horrorfilme können Computeranimationen verwenden, um Monster glaubhaft und mit realistischen Bewegungen darzustellen. Beispiele hierfür sind die Filme „*Van Helsing*“ (2004), „*Resident Evil 1+2*“ (2002, 2004) und „*Underworld 1+2*“ (2003, 2006).

Computeranimationen finden jedoch auch ihren Platz in Filmgenres, in denen man ihre Anwendung nicht unmittelbar erwarten würde. In Komödien oder Comedyserien können komödiantische Elemente metaphorisch überzeichnet werden („*Ally McBeal*“ (1997-2002)) oder Spezialeffekte der Hauptträger des komödiantischen Inhalts sein („*Die Maske*“ (1994), siehe auch [Giesen00, Giesen00e]). Als beispielsweise die Hauptfigur Ally McBeal in der gleichnamigen Serie jemandem „ihre liebe Freundin Ling“ vorstellt, die in der Serie eher als aggressive gefühlskalte Person gilt, verwandelt diese sich prompt mit Hilfe einer hochwertigen Computeranimation in die Alien-Königin⁴ aus den Alien-Filmen. Hierbei wurde wahrscheinlich eine spezielle Art der **2D**-Computeranimation eingesetzt, das so genannte Morphing, welches im folgenden Absatz näher erläutert wird. Die Persönlichkeit von Ling wurde also in Form einer visuellen Metapher verdeutlicht, wobei diese unerwartete realitätsferne Überzeichnung das komödiantische Element dieser Szene ist. In einer anderen Szene hat Ally eine Halluzination von einem tanzenden Baby, als sie unbewusst darüber nachdenkt, Mutter zu werden. Dieser Effekt wurde im Gegensatz zum erstgenannten Effekt als **3D**-Computeranimation realisiert. Der Protagonist in „*Die Maske*“ kann sich mit Hilfe einer magischen Maske verwandeln, wobei diese Verwandlungen stets betont humorvoll gestaltet sind und dennoch für die technischen Maßstäbe des Erscheinungsjahrs 1994 sehr aufwändig umgesetzt wurden.

Das oben genannte Morphing kann sowohl auf zweidimensionale Bilder als auch auf 3D-Modelle angewendet werden. Dabei wird ein fließender Übergang von einem Bild oder 3D-Modell zu einem anderen erzeugt, wobei sich Form und Farbe kontinuierlich der Ziel-Erscheinungsform annähern. Diese Technik wird in [Jack06] erläutert. Nach [Sura01] wurde zweidimensionales Morphing z.B. in [Beier92] behandelt, Morphing für Polygone und Polylinien in [Seder93] und [Alexa00], für Freiform-Kurven in [Samoil98] und für Voxelbasierte volumetrische Repräsentationen in [Cohen98]. [Sura01] selbst behandelt das Morphing zweidimensionaler Netze aus Dreiecks-Polygonen. Das Verfahren aus [Beier92] wurde im Musikvideo „*Black or White*“ von Michael Jackson eingesetzt, um Gesichter verschiedener Personen sogar mitten in der Bewegung ineinander zu transformieren. [Zhang02] ermöglicht Morphing für bildbasierte 3D-Objekte, die nur schwer durch klassische Hilfsmittel wie Polygone repräsentiert werden können.

Die Erstellung von Computeranimationsfilmen erfordert im Wesentlichen drei Schritte. Zunächst muss die 3D-Szene erstellt werden, dabei müssen unter anderem 3D-Objekte

³ http://www.scanline.de/_company/Awards.htm, 12.02.2007

⁴ Die Alien-Königin tauchte zuerst im zweiten Teil „*Aliens – Die Rückkehr*“ (1986) auf.

erzeugt werden, welche die darzustellende Welt repräsentieren. Danach müssen Bewegungen oder andere Formen von zeitlichen Veränderungen wie z.B. Verformungen für die 3D-Objekte festgelegt werden, möglich sind auch Änderungen der Beleuchtung oder Kamerafahrten. Dieser Vorgang wird als Animieren bezeichnet. Erst auf der Basis einer solchen Animation ist es sinnvoll, die Szene aus der Sicht einer virtuellen Kamera in aufeinander folgenden Einzelbildern darzustellen und zu speichern. So entsteht ein abspielbarer Computeranimationsfilm. **Das Thema dieser Arbeit ist die weitgehende Automatisierung des zweiten Schrittes, also des Animierens.**

Wie oben erläutert wurde, können Computeranimationen nahezu beliebig mit anderen Filmherstellungstechniken kombiniert werden. Computeranimationen können theoretisch in jedem Filmgenre eingesetzt werden. Aus diesem Grund ist diese Arbeit weder auf eine bestimmte Filmtechnik noch auf ein bestimmtes Filmgenre spezialisiert. Es sollen zwar vornehmlich Animationen für reine Computeranimationsfilme erstellt werden, jedoch können diese Filme auch mit verschiedensten anderen Filmtechniken kombiniert werden. Bei entsprechender Modellierung ist es auch möglich, dass mit anderen Filmtechniken erstellte Komponenten, wie gefilmte Umgebungen oder Schauspieler, beim automatischen Erstellen der Animation berücksichtigt oder sogar animiert werden.

Wie in Abbildung 2-Abbildung 9 gezeigt, können z.B. Aufnahmen von realen Schauspielern als eigene Objekte in die 3D-Szene einer Computeranimation eingesetzt werden. Damit können diese Aufnahmen beim automatischen Erstellen einer Animation berücksichtigt und bewegt werden.

Um Schauspieler und reale Gegenstände in Computeranimationen einzubinden, muss eine Maske erstellt werden, die bestimmt, welche Bildbereiche eines Fotos oder einer Live-Action-Filmsequenz in die Computergrafik einfließen sollen. Diese Maske, auch Alpha-Kanal genannt, kann mit Compositing-Techniken für eine solche Einbindung verwendet werden (siehe dazu Kap. 4.3). Die Rendering-Systeme von 3D-Animationsprogrammen können in der Regel ebenfalls diesen Alpha-Kanal berücksichtigen, so dass z.B. Live-Action-Aufnahmen von Schauspielern wie zweidimensionale Pappaufsteller in eine 3D-Szene eingebunden werden können. So können sich die Schauspieler in reflektierenden Gegenständen der 3D-Szene spiegeln (siehe Abbildung 9, Spiegelung auf dem Kopf des Roboters). 3D-Objekte aus der Szene können Schatten auf die Schauspieler werfen (siehe Schatten der Leitplanke auf dem linken Hosenbein in Abbildung 2 oder Schatten der Straßenlaterne auf Schauspieler in Abbildung 3). Scheinwerferlichtkegel können die Silhouette eines Schauspielers projizieren (siehe Abbildung 3) und Schauspieler können vor einigen Teilen einer 3D-Szene und gleichzeitig hinter anderen Teilen derselben Szene erscheinen (siehe Kopf des Piloten in Abbildung 4).

Der Film zu Abbildung 4 - Abbildung 5 ist auf der beiliegenden DVD in folgendem Pfad zu finden:


 Movies\Space_Headspin.avi



Abbildung 2: Schatten auf Schauspieler



Abbildung 3: Silhouette in Lichtkegel

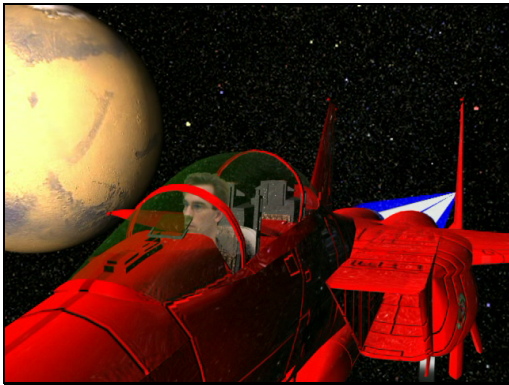


Abbildung 4: Pilot in Cockpit 1



Abbildung 5: Pilot in Cockpit 2



Abbildung 6: Schauspieler in 3D-Welt 1



Abbildung 7: Schauspieler in 3D-Welt 2



Abbildung 8: Schauspieler in 3D-Welt 3

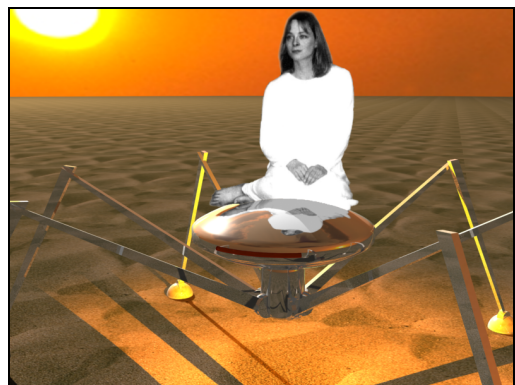
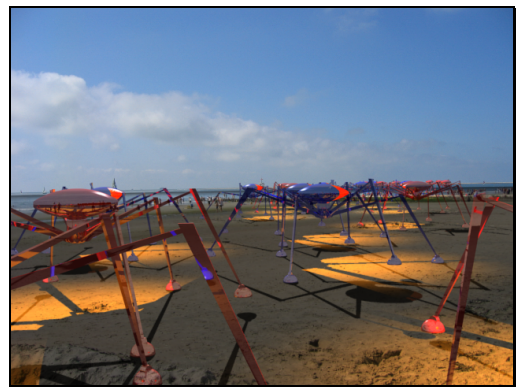


Abbildung 9: Spiegelung

Um für Live-Action-Elemente Alpha-Masken zu erstellen, findet meistens das so genannte Blue-Screen-Verfahren Verwendung (siehe [Hubb03]), bei dem inzwischen häufiger grüne als blaue Hintergründe eingesetzt werden. Viele Arbeiten beschäftigen sich mit der Verbesserung dieses Verfahrens. Das Verfahren aus [Chuang00] kann sogar das Brechungsverhalten z.B. von Glas und das Reflexionsverhalten spiegelnder Objekte erfassen und beim Compositing entsprechend berücksichtigen. Das Verfahren aus [Matus02] kann nicht nur eine Maske für diffus begrenzte Objekte erstellen, die z.B. Fell haben, sondern erfasst dabei auch deren dreidimensionale Struktur und ihr Beleuchtungsverhalten, so dass diese Objekte durch Image-Based-Rendering (siehe Kap. 5.5) aus verschiedenen Blickwinkeln und mit veränderten Beleuchtungsverhältnissen vor neuen Hintergründen dargestellt werden können. Das Verfahren aus [Chuang02] erleichtert das Erstellen einer Maske für Vordergrundelemente, die vor natürlichen Hintergründen gefilmt wurden, die nicht nur eine durchgängige Farbe haben wie z.B. beim Blue-Screen-Verfahren.

Umgekehrt können Computeranimationen in Live-Action-Filme eingebunden werden, wie die folgenden Bilder zeigen. Die Roboter und Fahrzeuge wurden mit dem hier entwickelten Animationswerkzeug „MaxControl“ automatisch animiert und anschließend in eine Live-Action-Szene eingebunden. Die virtuellen 3D-Objekte können auch Schatten und Licht auf die Live-Action-Szene werfen. Es wird also auch diese Richtung der Kombination von Live-Action und Computeranimationen unterstützt.



Abbildungen 10: Integration von Computeranimationen in Live-Action-Filme

Für abstrakte Darstellungen oder die Kombination mit Zeichentrick können die mit automatisch animierten Szenen auch mit *Cel-Shading* gerendert werden. Generell ist also nicht die Verwendung eines bestimmten Rendering-Systems festgelegt.

Da mit MaxControl für die automatische Bewegung von 3D-Objekten die physikalischen Gesetze der Mechanik verwendet werden können, sind bei so animierten 3D-Objekten die Bewegungsdaten wie Impulse und wirkende Kräfte zu jedem Zeitpunkt der Animation

bekannt. Auf der Basis dieser Daten könnten passende Bewegungen eines Motion-Ride-Systems berechnet werden, um die Bewegungen in der Animation für einen Zuschauer spürbar zu machen. Somit könnte die vorliegende Arbeit auch für Motion-Rides eingesetzt werden.

Dies sind nur einige Beispiele für mögliche Anwendungen. Durch die erweiterbare Architektur kann MaxControl für die Verwendung in verschiedensten Filmgenres angepasst werden.

1.2 3D-Animationsprogramme im Vergleich mit dieser Arbeit

Bei dreidimensionalen Computeranimationen sind grundsätzlich die folgenden zwei Darstellungsarten zu unterscheiden:

Eine Computeranimation kann in Echtzeit dargestellt werden. Echtzeit bedeutet generell, dass die Ergebnisse einer Benutzerinteraktion mit einem Programm ohne merklichen Zeitverlust sofort sichtbar sind. Bei einer computergestützten Simulation bedeutet Echtzeit im Idealfall, dass die Simulation in derselben Geschwindigkeit abläuft wie das reale Experiment. Für Computeranimationen wird in diesem Fall mindestens die dreidimensionale Darstellung der Szene in Echtzeit berechnet. Für eine als flüssig empfundene Wiedergabe sollten mindestens 30 oder besser 60 Einzelbilder pro Sekunde dargestellt werden. Da dieser Vorgang sehr rechenintensiv ist, wird er in der Praxis oft durch spezielle 3D-Hardware⁵ unterstützt. Weiterhin werden bei einer Echtzeitdarstellung meistens auch die zeitlichen Veränderungen der Szene direkt während des Ablaufes der Darstellung berechnet. Dies findet z.B. Verwendung bei einer Benutzerinteraktion, die in den Ablauf der Animation unvorhersehbar eingreift. Video- und Computerspiele verwenden daher meistens diese Art der Darstellung. Bei einer Echtzeitdarstellung steht für einen Rechenschritt von einem Frame zum nächsten nur die Darstellungszeit zwischen beiden Frames zur Verfügung ([Jack06]).

Eine andere Art dreidimensionaler Computeranimationen besteht darin, zuerst die Einzelbilder der Animation zu berechnen und diese auf einem Speichermedium abzulegen. Bei der Darstellung der Animation als Film werden dann von einem Speichermedium aus nur die berechneten Einzelbilder der Reihe nach abgespielt. Dies muss wieder schnell genug geschehen, um einen flüssigen Eindruck zu erzeugen. Der entscheidende Vorteil dieser Methode liegt darin, dass die Berechnung der Einzelbilder sehr viel länger dauern darf als das sukzessive Abspielen der Animation. Dadurch entfallen die bei Echtzeit-Computeranimationen gegebenen Limitierungen hinsichtlich der grafischen Qualität der Darstellung und der Komplexität der dreidimensionalen Szene sowie des Verhaltens ihrer 3D-Objekte.

Diese zweite Art von Computeranimationen wird in dieser Arbeit behandelt, wobei gerade die Einbindung einiger bisher oft vernachlässigter Vorteile der Echtzeitanimation besondere Beachtung findet.

Bei klassischer Computeranimation für Filme, die **nicht** in Echtzeit erstellt oder dargestellt wird, bestand die Zielsetzung meistens darin, eine überschaubare Anzahl hochwertiger 3D-Objekte möglichst kunstvoll zu animieren. Diese Aufgabe richtete sich daher auch eher an Künstler als an Spezialisten für 3D-Technologie. Dies galt sowohl für die Erstellung der 3D-Modelle als auch für deren Animation. Das Benutzerinterface professioneller

⁵ Eine spezielle Hardware, die ausschließlich für die schnelle und hochwertige Darstellung dreidimensionaler Computergrafiken in Echtzeit konzipiert ist. Eine solche Hardware kann den Hauptprozessor eines bestehenden Computersystems bei dieser Art von Aufgaben stark entlasten.

3D-Animationsprogramme erforderte daher auch nur wenige Fachkenntnisse über mathematische oder programmspezifische Grundlagen. Diese Form der Erstellung von 3D-Animationen bildet auch heute noch die Grundlage der meisten 3D-Animationsprogramme. Objekte können dabei relativ intuitiv erstellt und mit einfachen anschaulichen Benutzerinteraktionen einzelbildweise animiert werden. Durch Interpolationstechniken kann der Aufwand einer solchen Erstellung einer Animation erheblich verringert werden. So können einfache 3D-Animationen effizient und benutzerfreundlich hergestellt werden.

Diese Methode stößt an Grenzen, wenn es darum geht, komplexe Objekte in Bezug auf physikalisch korrekte Bewegungen und realistisches Verhalten glaubwürdig zu animieren. Dies führte insbesondere zu Problemen bei der Animation von Menschen und Tieren.

Um hier lebensechte Bewegungen erzeugen zu können, ohne das physikalische Verhalten und die kognitiven Fähigkeiten der zu animierenden Objekte simulieren zu müssen, wurde das „Motion Capturing“-Verfahren entwickelt (siehe dazu [Jack06]). Dabei werden die Bewegungen von Schauspielern oder auch von Tieren auf ein 3D-Modell übertragen, indem die Bewegungen von speziellen Markierungen am Schauspieler bzw. am Tier im zeitlichen Verlauf durch Kameras verfolgt und im Computer gespeichert werden. Dadurch entstehen Bewegungen für das Computermodell, die das Verhalten des Schauspielers bzw. des Tieres originalgetreu wiedergeben, wie z.B. die Art zu gehen oder zu fallen, wobei zwangsläufig gleichzeitig physikalisch korrekte Bewegungen entstehen. Einen Überblick über die historische Entwicklung der Motion-Capture Technologie gibt [Stur94]. Eine Anzahl von Arbeiten beschäftigt sich mit der Verbesserung der Motion-Capture Technologie, entweder bezüglich der Bewegungserfassung selbst oder bezüglich der Weiterverarbeitung der erfassten Bewegungsdaten. [Shin01] löst das Problem, Motion-Capture-Daten von einem Schauspieler auf eine Computeranimations-Figur zu übertragen, die eine andere Größe und andere Proportionen hat als der Schauspieler. [Zord03] überträgt Motion-Capture-Rohdaten, die aus einer Folge von Punktpositionen bestehen, auf eine 3D-Figur auf der Basis einer mechanischen Simulation von Federn und Gelenken. [Kov02] korrigiert automatisch Fehler bei der Bewegung von Füßen am Boden, die den Eindruck erwecken, als würden die Füße über den Boden gleiten statt bei Bodenberührung eine relativ feste Position dort zu behalten. Solche Fehler können durch Messfehler beim Motion-Capturing oder durch nachträgliches Editieren der Daten entstehen. Normalerweise werden optische Motion-Capturing-Verfahren vor neutralem Hintergrund angewendet. Nach [Rob06] wurde für den Film „*Fluch der Karibik 2*“ (2006) jedoch das neuartige Motion-Capturing-System „Imocap“ eingesetzt, das direkt am Set Motion-Capture-Daten aufnehmen kann, so dass vor dem Hintergrund des Sets gefilmte Schauspieler durch Computerfiguren ersetzt werden können, die genau die Bewegungen der Schauspieler reproduzieren.

Auch das Motion-Capturing stößt an seine Grenzen, sobald Objekte animiert werden sollen, die in der realen Welt nicht zu finden sind, wie futuristische Raumschiffe, Roboter oder Fabelwesen ohne vergleichbare Vorbilder in der Natur. Die Grenzen lassen sich erweitern, wenn man Puppen baut, deren Gelenke ihre aktuellen Drehwinkel messen können, so dass ihre Bewegungen auf ein Computermodell übertragen werden können. Ein Puppenspieler kann diese Puppe nun bewegen und somit ihre Bewegungen auf ein Computermodell übertragen, ohne dass sie Puppe dabei einem natürlichen Vorbild gleichen muss. Diese Technik wird auch als Waldo bezeichnet (siehe [Jack06] (eher in Bezug auf Puppensteuerung) und [Stur94] (in Bezug auf Computeranimationen)).

Die ständig wachsende Computerleistung ermöglicht inzwischen sehr realistische und komplexe Computeranimationsfilme. Die Komplexität liegt dabei nicht nur in der

geometrischen Komplexität und in der Anzahl der Objekte einer Szene, sondern vor allem auch in der theoretisch möglichen Komplexität des zeitlichen Verhaltens dieser Objekte. In vielen Kinofilmen wie z.B. „*Herr der Ringe – Die Zwei Türme*“ (2002) werden große Menschenmassen oder ganze Armeen mit Computergrafik dargestellt. Das Verhalten individueller Objekte aus einer entsprechend großen Anzahl solcher auch funktionell komplexer Objekte ist mit herkömmlichen Animationstechniken nur noch mit kaum vertretbarem Aufwand möglich, da die Bewegungen jedes einzelnen Objektes dabei manuell definiert werden müssten. Auch mit Motion Capturing wäre eine solche Aufgabe ohne weitere Hilfsmittel nur schwer zu bewerkstelligen, da auch die modernsten Motion Capturing Techniken nur eine sehr begrenzte Anzahl verschiedener Objekte (z.B. Schauspieler) gleichzeitig erfassen können. Bei der Titelsequenz zum Videospiel „*Onimusha*“ hat der Hersteller Capcom allerdings ein Motion Capture System verwendet, mit dem bis zu sechs Personen gleichzeitig erfasst werden konnten⁶. Wird jedoch eine noch höhere Anzahl an Objekten benötigt, würde auch ein Kopieren mehrerer einzeln mit Motion Capturing animierter Objekte ohne weitere Hilfsmittel keine realistische Interaktion der Objekte untereinander ermöglichen. Daher findet in solchen Fällen immer häufiger eine verstärkte Automatisierung des Animationsvorganges durch computergestützte Simulationen Verwendung (siehe dazu [Jack06]). Dabei wird das zeitliche Verhalten von Objekten in Computergrafik-Szenen algorithmisch gesteuert. Entweder werden alle Objekte zentral von einem Algorithmus gesteuert oder es werden den Objekten in der Szene allgemein definierte Verhaltensweisen zugeordnet, so dass die Objekte ihr Verhalten weitgehend selbst bestimmen. So können zum Beispiel ganze virtuelle Armeen selbständig gegeneinander kämpfen, ohne dass jede einzelne Bewegung manuell definiert werden muss. Jedoch sind auch diese Systeme durch verschiedene Faktoren in ihren Möglichkeiten beschränkt. Sie verwenden z.B. nur spezielle Konzepte zur Spezifikation des Verhaltens der Objekte wie an endliche Automaten angelehnte Übergangsdigramme oder sie werden nur für spezielle Probleme wie das Animieren menschlicher Figuren entwickelt. Auch die Vorteile der Objektorientierung bleiben oft ungenutzt.

Im Rahmen dieser Arbeit wurde nun ein System entwickelt, das eine weitgehend automatisierte Animation einer großen Anzahl von Objekten in einer 3D-Szene eines professionellen 3D-Animationsprogrammes ermöglicht. Die automatisch animierten Objekte sind dabei prinzipiell weder in ihrer geometrischen noch in ihrer strukturellen Komplexität (z.B. in der Anzahl von Unterobjekten) beschränkt. Ebenso unterliegt die Komplexität des Verhaltens jedes einzelnen Objektes keinen prinzipiellen Beschränkungen und die Verhaltensweisen sind nicht auf spezielle Lösungskonzepte festgelegt. Damit realistische Animationen entstehen, basieren Bewegungen in der Regel auf der Simulation von Mechanik, die durch das automatische Setzen von Kräften gemäß den Verhaltensweisen der Objekte beeinflusst wird. Damit jedoch auch hierdurch keine Einschränkungen entstehen, ist die Verwendung der Simulation von Mechanik nicht zwingend vorgeschrieben. Neben den Bewegungen der Objekte können auch die meisten anderen Eigenschaften der Objekte durch das System automatisch animiert werden, wie z.B. die Helligkeit einer Lichtquelle oder die Lautstärke einer Tonquelle. Das System wurde außerdem nicht für eine eingeschränkte Menge von Animationsproblemen entwickelt und es nutzt auf der Basis eines programmiersprachlichen Ansatzes die Vorteile der Objektorientierung.

⁶ <http://www.siggraph.org/s2000/media/news/rel10.html>,
<http://shop.capcom.com/store/capcomus/ContentTheme/pbPage.ProductInformation/ThemeID.166200>,
12.02.2007

1.3 Thesen

Das hier entwickelte Animationssystem basiert auf den folgenden Thesen:

- T-1: Für ein Simulationssystem zur automatischen Animation von 3D-Szenen ist eine Arbeitsteilung zwischen Künstler und Programmierer möglich, bei der der Künstler das Szenendesign sowie die manuelle Steuerung des Simulationssystems übernimmt und der Programmierer die Verhaltensweisen zur automatischen Steuerung der 3D-Objekte implementiert.
- T-2: Techniken der Objektorientierung sind in Verbindung mit der Simulation von Physik gut geeignet, um automatisiert realistische Animationen zu erzeugen.
- T-3: Ein entsprechendes Automatisierungssystem kann so konzipiert werden, dass das Handeln der 3D-Objekte gemäß der gewünschten Handlung des Films durch algorithmisch definierte Verhaltensweisen weitgehend festgelegt und auch manuell beeinflusst werden kann.
- T-4: Die Steuerung automatisch kontrollierter Objekte kann so gestaltet werden, dass sie mit Anweisungen an einzelne Schauspieler oder viele Statisten vergleichbar ist. Es ergeben sich vergleichbare Vor- und Nachteile.
- T-5: In Verbindung mit einer automatisierten Animation können auch Tonspuren mittels Samples mit angepasster Lautstärke, Tonhöhe und Position im 3D-Raum auf Basis der Simulation sinnvoll automatisch erzeugt werden, da sich hierbei erhebliche Vorteile gegenüber der bisher üblichen manuellen Nachvertonung eines Films ergeben können.
- T-6: Durch eine offene objektorientierte Architektur zur Definition von Verhaltensweisen für 3D-Objekte kann ein Simulationssystem flexibler sein als die meisten bisher existierenden Lösungen für die automatische Animation von 3D-Szenen.

2 Aufgabenstellung und Zusammenfassung dieser Arbeit

2.1 Automatisierung der Erstellung von Animationen: Vorstellung des entwickelten Lösungskonzeptes

Zur Produktion photorealistischer Computeranimationsfilme werden professionelle 3D-Animationsprogramme verwendet. Im Allgemeinen stellen diese Programme Hilfsmittel zur Durchführung von drei bereits in Kapitel 1.1 aufgeführten Hauptarbeitsschritten zur Verfügung:

Im ersten Schritt wird eine 3D-Szene entwickelt. Sie besteht unter anderem aus dreidimensionalen Objekten, deren Form, Oberflächeneigenschaften und Lage im 3D-Raum mit dem Animationsprogramm festgelegt werden. Die Beleuchtung der Szene kann durch verschiedene Arten von Lichtquellen erfolgen. Die Szenen können durch weitere Attribute wie Atmosphäreffekte und Hintergrundbilder ergänzt werden. Eine solche Szene ist zunächst statisch.

Der zweite Arbeitsschritt ist im Allgemeinen die Animation der Szene. Die Aufgabe in diesem Arbeitsschritt ist die Festlegung einer Folge von Zuständen für die Szene, welche die zeitlichen Veränderungen der Szene festhält. In einem Zustand sind unter anderem die momentanen Werte für die relevanten Eigenschaften aller Objekte, etwa Position und Rotation, festgehalten, bei sichtbaren Objekten deren momentane Farbe und bei Lichtquellen z.B. deren momentane Helligkeit. Die Zustände, technisch auch Frames genannt, entsprechen den Einzelbildern des als Endprodukt entstehenden Films.

Ist eine Animation festgelegt, wird jeder Frame der Animation aus der Sicht einer virtuellen Kamera in ein hochwertiges computergeneriertes Bild umgesetzt. Dieser dritte Arbeitsschritt wird als Rendern bezeichnet. Wird die so erzeugte Bildfolge nun in der korrekten Geschwindigkeit abgespielt, kann die vorher festgelegte Animation als Film betrachtet werden.

3D-Computeranimationsfilme werden meistens von Künstlern erstellt und nicht von Programmierern. Eine Programmiersprache wird daher in diesem Bereich von den Anwendern nur als letztes Mittel eingesetzt, um schwierige Probleme bei der Erstellung einer Animation zu lösen. Da dies teilweise jedoch unumgänglich ist, bieten viele 3D-Animationswerkzeuge Skriptsprachen an, diese sind in den meisten Fällen jedoch in Sprachumfang und Laufsicherheit nicht mit ausgereiften Programmiersprachen wie Java vergleichbar. Ebenfalls bedingt durch die Ausrichtung auf eine Benutzung durch Künstler soll das Arbeiten mit einem Werkzeug möglichst einfach und anschaulich sein, so dass sich die meisten dabei verfügbaren Techniken auf eine grafische Benutzeroberfläche stützen, welche die Objekte dreidimensional darstellt und eine Manipulation der Objekte im 3D-Raum durch verschiedene Eingabegeräte wie Maus, 3D-Maus oder über eine Tastatur ermöglicht.

Am weitesten verbreitet sind zur Erstellung von Animationen manuelle Keyframe-Techniken. Dabei werden Eigenschaften von Objekten nur in einigen Frames der Animation manuell festgelegt. Solche Frames werden Keyframes genannt. Die Werte dieser Eigenschaften zwischen den Keyframes werden dann durch Interpolationstechniken automatisch bestimmt. Um wenige 3D-Objekte zu erstellen und zu animieren ist die manuelle Keyframe-Technik eine sehr geeignete Vorgehensweise.

Die Weiterentwicklung der Technik für Computeranimationen erlaubte jedoch die Erstellung zunehmend komplexerer Szenen mit einer hohen Anzahl animierter 3D-Objekte.

Beispielsweise ist es inzwischen durchaus möglich, Regen in einer 3D-Animation so darzustellen, dass jeder einzelne Regentropfen durch ein eigenes Objekt repräsentiert wird. Die entsprechend hohe Anzahl solcher Regentropfen mit den herkömmlichen Techniken manuell zu animieren wäre nur mit nicht vertretbarem Aufwand realisierbar. Aus der Notwendigkeit heraus entstanden deshalb neue Techniken, um solche und ähnliche Probleme zu lösen, die bei einer großen Anzahl zu animierender Objekte entstehen.

Partikelsysteme stellen eine solche Technik dar, mit der viele gleichartige Objekte auf einfache Weise animiert werden können. Partikelsysteme sind hauptsächlich für die Animation natürlicher Partikelphänomene wie Regen, Schnee oder Sandstürme konzipiert. Durch die Angabe von Parametern kann bei einigen Partikelsystemen z.B. festgelegt werden, wie viele Objekte in einem Zeitintervall neu entstehen sollen, wie schnell sie sich in welche Richtung bewegen sollen, wie lange sie in der Szene bleiben sollen bevor sie wieder verschwinden und ob ihre Bewegung leicht tänzelnd sein soll. So können mit demselben System leicht Phänomene wie Regentropfen, tänzelnd zu Boden fallende Schneeflocken oder unregelmäßig aufsteigende Luftblasen erzeugt werden.

Auch für komplexere 3D-Objekte wie Menschen werden zunehmend Werkzeuge entwickelt, um viele solcher Objekte bequem und automatisiert zu animieren. Das Konzept automatisch handelnder 3D-Figuren wird auch in [Jack06] und [Giesen00e] behandelt.

Auch solche Werkzeuge sind meistens für die Benutzung durch Künstler konzipiert. Um deshalb eine anschauliche Spezifikation von Verhaltensweisen für Objekte zu ermöglichen, stellen diese Werkzeuge im Allgemeinen grafische Methoden oder auch Skriptsprachen zur Verfügung. Dabei werden häufig bestimmte Konzepte gewählt wie Zustandsübergangsdigramme, die mit Diagrammen endlicher Automaten vergleichbar sind, oder regelbasierte Systeme mit einer begrenzten Zahl wählbarer Arten von Regeln. Eine Festlegung auf solche Konzepte kann einerseits die Spezifikation von Verhaltensweisen erleichtern, andererseits werden die Spezifikationsmöglichkeiten durch eine derartige Festlegung häufig auch eingeschränkt. Auch fehlen solchen Werkzeugen dann fast immer die Vorteile objektorientierter Programmiersprachen.

Die meisten Systeme zur automatischen Erstellung von Animationen sind schon dadurch eingeschränkt, dass sie im Hinblick auf eine bestimmte Gruppe von Animationsproblemen entwickelt werden. Versucht man mit ihnen Animationen zu erstellen, für die sie nicht konzipiert wurden, stößt man häufig recht schnell an ihre Grenzen. Da z.B. viele Partikelsysteme auf die Animation von Phänomenen wie Regen, Schnee oder Sandstürme ausgerichtet sind, stoßen sie an Grenzen, wenn man versucht, mit einem solchen Partikelsystem andersartige Animationen zu erstellen, die innerhalb des begrenzten Definitionskonzeptes des Partikelsystems nur bedingt beschreibbar sind. Ein Beispiel wäre die Aufgabe, mit einem Partikelsystem Autos zu animieren, die mit mechanisch korrekt simulierter Radfederung auf einer Straße fahren und dem Streckenverlauf automatisch folgen sollen.

Unabhängig von der Ausdrucksstärke eines Konzeptes sollte auch berücksichtigt werden, mit welchem Aufwand bestimmte Probleme mit diesem Konzept lösbar sind. Es ist nicht unerheblich, wie bequem ein bestimmter Algorithmus in einem solchen System beschreibbar ist und ob das System für die Lösung eines Problems z.B. auf Dateien oder das Internet zugreifen kann.

Das hier entwickelte Animationssystem „MaxControl“ verfolgt nun einen offeneren Ansatz, um 3D-Computeranimationsfilme mit einem professionellen 3D-Animationsprogramm weitgehend automatisch zu animieren.

Grundlage einer Animation ist dabei bis auf wenige Ausnahmen das „Simulieren“ des Verhaltens von 3D-Objekten. Das Simulieren beruht auf Iterationsverfahren, die in Schritten („Simulationsschritten“) für jeden Frame der Animation und eventuellen Zwischenschritten („Simulationszwischenritten“) ablaufen. Dabei werden in jedem Iterationsschritt die für die zu simulierenden Objekte spezifizierten Verhaltensweisen in einer festen Reihenfolge jeweils für ein Zeitintervall Δt aufgerufen. Ausgehend von einem Anfangszustand und gewissen „Randbedingungen“ für den Verlauf der Animation entsteht so eine Folge von Szenenzuständen, die als Folge von Frames eine Animation der Szene darstellt.

Verhaltensweisen sind syntaktisch Methoden, die bei ihrem Aufruf Handlungen, beschränkt auf das Zeitintervall Δt , durchführen. Sowohl die Verhaltensweisen als auch die daraus iterativ zusammengesetzten Tätigkeiten (Verhalten) sind dem jeweiligen 3D-Objekt zugeordnet.

Die Simulation gliedert sich bei MaxControl in drei Hauptkomponenten. Zum einen werden in Java implementierte Verhaltensweisen verwendet, um das Verhalten der Objekte zu simulieren. Um realistische Animationen zu erzeugen, soll dabei die Einhaltung der Gesetze der Mechanik gewährleistet sein. Die zweite Komponente ist daher die Simulation von Mechanik für Objekte, die als feste Körper definiert werden. Eine dritte Komponente sind manuell festgelegte Animationen, die für bestimmte Zeitintervalle und ausgewählte Eigenschaften in die Simulation einfließen können und so eine Beeinflussung der Simulation im Sinne eines Drehbuchs ermöglichen.

Das System MaxControl ist insbesondere für die Animation von Szenen mit sehr vielen komplexen Objekten konzipiert. Solche Szenen enthalten meistens eine relativ geringe Anzahl von unterschiedlichen Arten von Objekten. So sind Ampeln, die in wechselseitiger Kommunikation den Straßenverkehr regeln, aufgrund ihres Verhaltens einem anderen Typ von Objekt zuzuordnen als Autos, die entlang einer Strecke fahren, dabei die Signale von Ampeln beachten und anderen Autos ausweichen. Autos, die sich nur in Form, Motorleistung und einigen Verhaltensparametern unterscheiden, können hingegen demselben Objekttyp zugeordnet werden und so mit lediglich unterschiedlichen Parametern demselben spezifizierten Verhalten folgen. Statt also jedes einzelne Objekt manuell zu animieren werden bei MaxControl nur die verschiedenen Objekttypen mit zugehörigen Verhaltensweisen spezifiziert, die dann für die zugehörigen 3D-Objekte schrittweise simuliert werden um eine Animation der Objekte zu erzeugen.

Auch das Zusammenwirken vieler verschiedener Arten von Objekten in einem komplexen Gesamtsystem ist simulierbar. Beispiele hierfür sind ein Flughafen mit startenden und landenden Flugzeugen, Gepäcktransport und weiteren Details „bis zur letzten Positionslampe“ sowie ein Flugzeugträger mit automatischer Navigation und mit Flugzeugen die starten, landen, nachtanken und jeweils ein eigenes komplexes Verhalten haben.

Zur Spezifikation der Verhaltensweisen wird ein programmiersprachlicher Ansatz auf der Basis von Java verwendet, wodurch sich MaxControl von den meisten anderen Werkzeugen, die zur Lösung vergleichbarer Problemstellungen für 3D-Animationsprogramme entwickelt wurden, unterscheidet. MaxControl ist in dieser Hinsicht eher mit Werkzeugen wie ASAS [Reyn82, Foley93] vergleichbar, die eine Mehrzweck-Programmiersprache als Basis eines automatischen Animationssystems und auch für die Definition der Objektverhaltensweisen

verwenden. Im Fall von ASAS wurde LISP als Basisprogrammiersprache gewählt, wodurch ASAS nach [Reyn82] „auf den Schultern von Giganten“ steht, weil LISP für sich schon eine weit entwickelte und ausdrucksstarke Programmiersprache ist und ASAS auf dieser Basis die gleiche Ausdrucksstärke besitzt. Diesem Prinzip folgt auch MaxControl, wobei hier anstelle von LISP als Basissprache Java gewählt wurde. Durch den Einsatz von Java können unter anderem die Vorteile der Objektorientierung ausgenutzt werden, welche die meisten in 3D-Animationsprogrammen verwendeten Skriptsprachen nicht zur Verfügung stellen. Ein objektorientierter Ansatz eignet sich sehr gut für die in dieser Arbeit gestellte Aufgabe, denn es soll das zeitliche Verhalten von vielen **Objekten** definiert werden, die sich **Klassen** zuordnen lassen. Diese Ansicht wird auch in [Foley93] vertreten.

Im Sinne der Objektorientierung wird in MaxControl jedem Objekt in einer 3D-Szene ein entsprechendes Java-Objekt zugeordnet. Der Typ eines 3D-Objektes setzt sich dann zusammen aus der Klasse des zugeordneten Java-Objekts, die das gewünschte Verhalten des Objektes spezifiziert, und dem Basistyp, den das 3D-Objekt bereits vor der Zuordnung einer Java-Klasse im 3D-Animationsprogramm hat. So kann einem 3D-Objekt, das geometrisch die Form eines Quaders hat, eine Java-Klasse „Auto“ mit entsprechenden Verhaltensweisen zugewiesen werden. Das 3D-Objekt sieht dann aus wie ein Quader und verhält sich wie ein Auto. Ebenso kann einem 3D-Objekt, das geometrisch ein Zylinder ist, eine Java-Klasse „Rad“ zugewiesen werden, so dass sich dieser Zylinder dann verhält wie ein Rad eines Autos. Jedem 3D-Objekt in der Szene können so in Java spezifizierte Verhaltensweisen zugewiesen werden. Die Java-Klasse, die einem 3D-Objekt zugeordnet wird, bestimmt dann, welche Verhaltensweisen zu diesem 3D-Objekt gehören. Auch die Verhaltensweisen werden als Java-Klassen implementiert. Durch die Zuweisung einer Java-Klasse können einem 3D-Objekt besondere Eigenschaften hinzugefügt werden, die in der Klasse definiert sind und sich auf das Verhalten des Objektes beziehen. Eine solche Eigenschaft könnte z.B. bestimmen, welche Geschwindigkeit ein Fahrzeug erreichen soll oder angeben, mit welcher Geschwindigkeit ein Fahrzeug gerade fährt. Über diese Eigenschaften können sich die Objekte gegenseitig steuern oder Informationen voneinander erhalten.

Ein 3D-Objekt kann mehrere Verhaltensweisen besitzen. Jede Verhaltensweise eines Objektes kann auf den Zustand dieses Objektes und auch auf die Zustände anderer Objekte zugreifen, Eigenschaften daraus lesen und diese auch verändern. So kann ein Fahrzeug den Zustand einer Ampel lesen, um entsprechend zu reagieren. Ein Düsentriebwerk kann Kräfte auf andere mechanisch simulierte Gegenstände ausüben, um so den Luftstrom des Triebwerks zu simulieren, der leichte Objekte in seiner Strömungsrichtung in Bewegung versetzen kann.

Es sind auch Objekte ohne Verhaltensweisen möglich, die dann z.B. den Verlauf einer Rennstrecke markieren oder Begrenzungen definieren. Diese Objekte ändern ihre Zustände während der Simulation nicht selbst. Andere Objekte können die Eigenschaften solcher Objekte jedoch lesen und diese auch verändern.

Zudem beschränken sich Animationen, die mit MaxControl automatisch erstellt werden können, nicht nur auf die Bewegungen von Körpern. Die spezifizierten Verhaltensweisen können beispielsweise ebenso andere Objekteigenschaften steuern wie den Radius einer Kugel, die Helligkeit von Lichtquellen oder auch den Zustand virtueller Tonquellen und Kameras in der 3D-Szene ändern. Nur wenige vergleichbare Systeme wie [Funge99] schließen ebenfalls die Animation von Lichtern und Kameras mit ein.

Ein Vorteil, den objektorientierte Sprachen gegenüber anderen Programmiersprachen haben, ist die Vererbung. Diese kann hier sowohl für Objekteigenschaften als auch für Verhaltensweisen sinnvoll genutzt werden. Ein Objekttyp „LKW“ könnte vom Objekttyp

„Fahrzeug“ alle Eigenschaften wie Gewicht und Farbe erben, und dabei dann einen neuen Parameter „maximale Beladung“ erhalten. Das Navigationsverhalten eines Flugzeugs könnte vom Navigationsverhalten eines Schiffes erben, wobei die Flugzeugnavigation dann zusätzlich auch Flughöhen berücksichtigen könnte.

In jedem Iterationsschritt der Simulation werden die Verhaltensweisen aller Objekte in einer festen Reihenfolge für ein Zeitintervall Δt aufgerufen. Aus einem Szenenzustand werden dabei zunächst die Eigenschaften der Objekte übernommen. Durch den Iterationsschritt werden diese Eigenschaften dann verändert, wodurch der zugehörige Folgezustand der Szene erzeugt wird. Durch die Verhaltensweisen aller Objekte entsteht eine Übergangsvorschrift für die Zustände einer 3D-Szene, wobei der erste Frame der Animation den Anfangszustand der Simulation trägt. So entsteht automatisch eine Animation, die dann in dem verwendeten 3D-Animationsprogramm verfügbar ist.

In jedem Simulationsschritt werden durch das Ausführen der Verhaltensweisen generelle Handlungsmuster befolgt. Die so durchgeführten Handlungen laufen für jeden Simulationsschritt nur in dem Zeitintervall ab, das in dem Schritt simuliert werden soll. Dieses Zeitintervall wird im Allgemeinen weit unter einer Sekunde liegen. Die Verhaltensweisen können bei jeder Ausführung also in der Regel nur einen kleinen Teil eventuell zeitaufwändigerer Aufgaben erfüllen. Eine Verhaltensweise, welche durch die Kontrolle eines Roboterarms ein Objekt greifen und an einen anderen Ort bringen soll, kann dieses Ziel im Allgemeinen nicht in einem Simulationsschritt durchführen, sondern benötigt dazu mehrere, eventuell sehr viele, Simulationsschritte, in denen diese Verhaltensweise immer wieder aufgerufen wird. Eine Verhaltensweise muss also so implementiert werden, dass sie bei jeder Ausführung im Rahmen eines Simulationsschrittes bestimmen kann, welcher kleine Teil einer zeitaufwändigeren Aufgabe in diesem Schritt durchgeführt werden muss. Ein Fahrzeug könnte in diesem Sinne Verhaltensweisen wie „Navigation“, „Fahrzeugsteuerung“, „Steuerung der automatischen Gangschaltung“ und „Steuerung der Lichtanlage“ haben. Die Verhaltensweise „Navigation“ würde nun in jedem Simulationsschritt bestimmen, in welche Richtung das Fahrzeug im aktuellen Simulationsschritt fahren soll. Die Verhaltensweise „Fahrzeugsteuerung“ würde dabei die genauere Steuerung des Fahrzeugs basierend auf dieser Navigationsrichtung übernehmen. Sie würde in jedem Simulationsschritt bestimmen, in welchen Winkel die Steuerung gebracht wird und wie stark gebremst oder beschleunigt wird. Die Verhaltensweisen „Steuerung der automatischen Gangschaltung“ und „Steuerung der Lichtanlage“ führen in jedem Simulationsschritt entsprechende Aufgaben durch. Verhaltensweisen sind hier also **nicht** als ausführbare Einzelaktionen zu sehen, die nach **einer** Ausführung der Verhaltensweise vollständig zu Ende geführt sind, sondern sie definieren Teilschritte eines iterativ durchzuführenden Verhaltens, die in jedem Schritt der Simulation befolgt werden. Soll das Verhalten eines Fahrzeugs festgelegt werden, wird es also **nicht** Verhaltensweisen haben wie „Einen Meter vorwärts fahren“, „Einen Meter rückwärts fahren“ oder „Rechts abbiegen“, die eine bestimmte zeitaufwändige Handlung mit einem einzigen Aufruf der Verhaltensweise vollständig durchführen.

MaxControl ist also ein zustandsbasiertes Simulationssystem und nicht ereignisbasiert. Dieser Unterschied ist wichtig, weil viele vergleichbare Simulationssysteme ereignisbasiert sind, z.B. [Mirt00, Luck97/Müller98⁷, Zele91, Döll97]. Der Unterschied zwischen diesen Ansätzen wird auch in [Jack06] für die Simulation von Mechanik behandelt. Ereignisbasierte Systeme teilen Objekten Ereignisse zumeist über Nachrichten mit, auf welche diese Objekte dann beim

⁷ Beide Texte erläutern dasselbe System.

Empfang dieser Nachrichten reagieren. So arbeitet z.B. das oben erwähnte System ASAS. Jedoch handeln Objekte bei ASAS jeweils nur einmal in jedem Frame der Animation. Daher reagieren die Objekte dort nicht wie in den meisten ereignisgesteuerten Systemen sofort beim Empfang einer Nachricht auf diese, sondern erst, wenn sie das nächste Mal innerhalb eines Simulationsschritts zwischen zwei Frames zum „Handeln“ aufgerufen werden. Insbesondere der Datenaustausch über Nachrichten ist auch typisch für objektorientierte Systeme (siehe [Foley93] zu „Actors“). In MaxControl können die Objekte hingegen relativ direkt auf die Eigenschaften anderer Objekte zugreifen, ohne dass diese Eigenschaften in Form von Nachrichten freigegeben werden müssen.

Ein ereignisgesteuertes nachrichtenbasiertes System ist sicher gut für interaktive Benutzeroberflächen geeignet, mit einer nicht-kontinuierlichen Benutzerinteraktion. Wenn in einer grafischen Benutzeroberfläche z.B. eine Schaltfläche mit der Maus aktiviert wird, kann eine dadurch ausgelöste Nachricht das zuständige Objekt dazu veranlassen, sofort auf diese Interaktion zu reagieren. Für ein solches System ist im Allgemeinen auch leicht festlegbar, welche Objekte über welche Benutzerinteraktionen informiert werden müssen. Ein Vorteil dabei ist, dass ein solches System keine Anweisung ausführen muss, solange keine Interaktion des Benutzers vorliegt, was bei Anwendungsprogrammen über relativ lange Zeiträume der Fall sein kann.

Eine andere Situation liegt bei kontinuierlicher Benutzerinteraktion vor, wenn z.B. bei einer Autorennsimulation ein Lenkrad als Eingabesystem vom Benutzer ständig in einem veränderlichen Winkel gehalten wird. Hier kann schwer festgelegt werden, wann eine Änderung der Lenkradstellung eine Nachricht auslösen soll. Außerdem muss auch bei keiner Änderung die Rennsimulation ständig weiter laufen, so dass das System nicht abwarten kann, bis der Benutzer wieder in die Simulation eingreift. Eine ähnliche Situation liegt auch bei einem Animationssystem wie MaxControl vor, dass über Simulationen eine Animation erzeugt. Wenn beispielsweise ein Würfel auf einen Tisch fällt, so ist der Moment des Kontaktes zwischen Würfel und Tisch sicher ein Ereignis, auf das reagiert werden muss. Doch auch während des Fallens ändert sich der Zustand des Würfels ständig, so dass die Simulation nicht auf das Kontakt ereignis warten kann, sondern die Objektzustände in einzelnen Simulationsschritten laufend ändern muss. In einem solchen Szenario kann ein auf Nachrichten beschränkter Datenaustausch zwischen den Objekten störend wirken.

Zum einen ist schwer zu definieren, welche Daten an welche Objekte übermittelt werden sollten, was oft zu Lösungen führt, die einfach jedem Objekt alle Änderungen aller anderen Objekte mitteilen (siehe [Ewald06]). Wenn z.B. Fahrzeuge anderen Fahrzeugen ausweichen sollen, so wäre die bestmögliche Einschränkung der Nachrichtenmenge bei einem nachrichtenbasierten System, in jedem Simulationsschritt nur von allen Fahrzeugen an alle jeweils anderen Fahrzeuge ihre aktuelle Position und Geschwindigkeit als Nachrichten zu übermitteln, ohne anderen Objekten wie z.B. Straßenlaternen diese Nachrichten zukommen zu lassen. Die Geschwindigkeit eines Fahrzeugs ist für ein anderes Fahrzeug aber unter Umständen nur dann interessant, wenn sich beide Fahrzeuge recht nahe sind. Außerdem werden nun im Nachrichteneingang jedes Fahrzeugs jeweils Kopien dieser Informationen über jedes andere Fahrzeug gespeichert, oder wenigstens Referenzen darauf, was zu einer unnötigen Datenmenge führt. Bei einem zustandsbasierten System wie MaxControl greifen alle Fahrzeuge einfach nur direkt auf die Positionen der jeweils anderen Fahrzeuge zu und lesen nur dann zusätzlich deren aktuelle Geschwindigkeiten, wenn diese Fahrzeuge ihnen gefährlich nahe sind. Die Anzahl übermittelter Daten kann so situationsbedingt sinken und es wird auch direkt auf diese Daten zugegriffen, ohne Kopien dieser Daten zu erzeugen. Noch schwieriger ist bei einem nachrichtenbasierten System, eine allgemeingültige Festlegung der Empfängermenge und der jeweils als Nachrichten zu übermittelnden Daten vorzunehmen.

Wenn im obigen Szenario zusätzlich Vögel implementiert werden, die davonfliegen sollen, wenn ihnen Fahrzeuge zu nahe kommen, müssten auch die Fahrzeuge angepasst werden, so dass sie ihre Positionen nun auch an alle Vögel übermitteln. Eine Alternative wäre zwar die Möglichkeit, dass Objekte ihr „Interesse“ an Eigenschaften anderer Objekte „anmelden“, so dass diese anderen Objekte ihnen dann diese Eigenschaften ständig oder bei jeder Änderung als Nachrichten mitteilen. Ein solcher Mechanismus würde die Implementierung jedoch weiter verkomplizieren und auch weiterhin die Übermittlung und Speicherung der Nachrichten nicht verhindern können. Dagegen könnten neu implementierte Vögel bei MaxControl direkt auf die Positionen der Fahrzeuge zugreifen, ohne die Fahrzeuge dafür neu implementieren zu müssen. Bei Systemen mit Nachrichtenaustausch ohne das oben genannte „Anmeldesystem“ muss also für jedes Objekt festgelegt werden, welchen anderen Objekten welche Informationen über dieses Objekt mitgeteilt werden müssen, wohingegen bei MaxControl für jedes Objekt „X“ festgelegt werden muss, welche Informationen welcher anderen Objekte für dieses Objekt „X“ relevant sind und von ihm gelesen werden müssen. Diese zweite Richtung ist in der Regel leichter und natürlicher festlegbar, weil sie vom Verhalten des jeweiligen Objektes „X“ selbst abhängt und dieses Verhalten bei der Implementierung des Objektes „X“ bereits als Zielsetzung bekannt ist. Die umgekehrte Richtung ist hier schwieriger festlegbar. Es muss dann z.B. bestimmt werden, für welche Objekte die Position eines Autos wichtig sein könnte, um sie diesen Objekten als Nachrichten zu übermitteln. Dies lässt sich schwieriger festlegen, weil es vom Verhalten eines Objektes abhängt, ob es Daten über Fahrzeuge in der Szene benötigt, und die in Frage kommenden Objekte und ihr jeweiliges Verhalten zum Zeitpunkt der Implementierung der Fahrzeuge unter Umständen nicht bekannt sind.

Bei nachrichtenbasierten Systemen könnte außerdem eine nichtterminierende Nachrichtenschleife entstehen, wenn die Objekte sofort bei Erhalt einer Nachricht darauf reagieren. Ein Objekt A könnte einem Objekt B eine Nachricht schicken, die Objekt B veranlasst, Objekte A eine Nachricht zu schicken. Wenn dies wiederum Objekt A zum erneuten Senden der ersten Nachricht an Objekt B veranlasst, wäre eine Schleife entstanden und der aktuelle Simulationsschritt könnte nicht zu Ende geführt werden.

Der Benutzer kann bei MaxControl in für jede Eigenschaft frei wählbaren Zeitintervallen die Werte dieser Eigenschaft manuell bestimmen. Dadurch erhält der Benutzer eine Möglichkeit, das Verhalten von Objekten auf einer im Vergleich zu klassischer Keyframe-Animation hohen Abstraktionsebene zu steuern, indem er Eigenschaften manuell kontrolliert, die das Verhalten der Objekte beeinflussen, wie z.B. die Geschwindigkeit, die ein Fahrzeug erreichen soll oder die Zielfahrtrichtung eines Schiffes. So kann durch das manuelle Festlegen weniger Eigenschaften ein komplexes Verhalten gesteuert werden, das als direkte Keyframe-Animation von Eigenschaften wie Position und Rotation des Objektes nur mit ungleich höherem Aufwand realisierbar wäre.

Manuell kontrollierte Werte von Eigenschaften setzen durch die Simulation entstehende Änderungen an diesen Eigenschaften außer Kraft. Eine Verhaltensweise, die z.B. eine Glühlampe blinken lassen soll, kann in jedem Simulationsschritt die Helligkeit der Lampe entsprechend ändern. Diese Lampe kann durch manuelle Kontrolle ihrer Helligkeit so eingestellt werden, dass sie dennoch immer dunkel bleibt, auch wenn sie von ihrer eigenen Verhaltensweise oder einem anderen Objekt den Befehl erhält, ihre Helligkeit zu ändern. Denn versucht die Verhaltensweise eines Objektes, eine so manuell kontrollierte Eigenschaft zu verändern, so wird diese Änderung nicht angenommen und die Eigenschaft behält ihren manuell eingestellten Wert..

MaxControl unterstützt eine kompositionelle Wiederverwendung auf zwei Ebenen. Die Java-Klassen, die 3D-Objekten zugeordnet werden, können wieder verwendet werden, indem sie mehreren 3D-Objekten einer Szene zugeordnet werden. Jedem Rad kann als Komponente eines Autos der Klasse „Auto“ die Klasse „Rad“ zugeordnet werden, so dass diese Klasse mit ihren Verhaltensweisen für jedes Rad wieder verwendet wird. Auch die Verhaltensweisen eines 3D-Objektes können in anderen 3D-Objekten wieder verwendet werden, dabei kann das Verhalten eines 3D-Objektes aus Verhaltensweisen anderer 3D-Objekte zusammengesetzt werden. So könnte ein Amphibienfahrzeug die Verhaltensweisen des Typs „Auto“ und die Verhaltensweisen des Typs „Schiff“ zusammen als Komponenten verwenden und je nach Situation zwischen diesen Verhaltensweisen wählen.

MaxControl wurde in ein professionelles 3D-Animationsprogramm eingebunden. Dadurch wird es möglich, eine 3D-Szene mit diesem 3D-Animationsprogramm zu erstellen und dann mit MaxControl durch Simulationstechniken automatisch zu animieren. Die Kommunikation von MaxControl mit dem 3D-Animationsprogramm während des Simulationsvorganges ist so angelegt, dass die Szene nicht erst vom 3D-Animationsprogramm nach MaxControl exportiert, dort animiert und dann wieder in das 3D-Animationsprogramm importiert werden muss, sondern es wird während der Simulation mit den Daten gearbeitet, die das 3D-Animationsprogramm speichert. So bleibt die Szene im 3D-Animationsprogramm und wird dort um die durch die Simulation erzeugte Zustandsfolge erweitert, welche die entstehende Animation darstellt. Die Zustandsfolge wird dabei mit effizient genutzten Keyframe-Techniken erzeugt, was sowohl die zu speichernde Datenmenge als auch die Anzahl der nötigen Kommunikationsschritte stark verringert. Diese Technik wird mit den gleichen Vorteilen auch für das Lesen der manuell kontrollierten Eigenschaften genutzt. Aus der so automatisch erstellten Animation kann anschließend mit dem 3D-Animationsprogramm durch Rendering ein Film hergestellt werden, der dann unabhängig von den zu seiner Herstellung verwendeten Werkzeugen abspielbar ist.

Viele vergleichbare Werkzeuge wie [Reeves83, Reeves85, Kalra92, Wei03, Terz88, Ell04, McKen90, Lau02] erzeugen direkt nach jedem Simulationsschritt durch Rendering ein Einzelbild der Animation und speichern den zugrundeliegenden Zustand der Szene nicht weiter, sondern erzeugen daraus direkt den nächsten Folgezustand. Insbesondere echtzeitfähige Systeme gehen in der Regel so vor. Dies hat den Nachteil, dass zusätzlich zur Simulationszeit immer auch die oft erhebliche Zeit für das Rendering der Einzelbilder aufgewendet werden muss, bevor die erzeugte Animation untersucht werden kann. Werden jetzt unerwünschte Ergebnisse der Simulation oder auch unerwünschte optische Eigenschaften an den durch Rendering entstandenen Einzelbildern entdeckt, müssen sowohl der Simulations- als auch der Rendering-Vorgang mit neuen Einstellungen wiederholt werden. Außerdem können so direkt vorgehende Systeme nur schwer Objekte in die Einzelbilder einfließen lassen, die ohne das jeweilige System animiert wurden, wie z.B. rein manuell animierte 3D-Objekte, die sich zusätzlich in der Szene befinden sollen. Auch eine Nachbearbeitung der Animation wie das Hinzufügen von Effekten, die auf anderen Simulationssystemen basieren, ist dann schwierig. Z.B. könnte es erwünscht sein, nachdem Fahrzeuge mit einem System automatisch animiert wurden, zusätzlich Regen auf diese Fahrzeuge fallen zu lassen, der durch ein unabhängiges Partikelsystem erzeugt wird. Auch die dazu möglichen Compositing-Techniken unterliegen Einschränkungen, die in Kapitel 4.3 erläutert werden. MaxControl kann dagegen bereits vor der Simulation in der Szene existierende animierte 3D-Objekte entweder ignorieren oder auch ihre vorgegebene Animation in die Simulation einfließen lassen. So können mit MaxControl animierte 3D-Objekte beispielsweise vorher manuell animierten 3D-Objekten ausweichen oder von ihnen umgeworfen werden. Ist eine Animation mit MaxControl erstellt worden, unterscheidet diese sich technisch nicht von einer manuell erstellten Animation, so dass sie wie gewohnt

weiterbearbeitet werden kann, z.B. wie oben gefordert durch Regen, der nachträglich durch ein Partikelsystem erzeugt wird und von den Fahrzeugen abprallt. Auch während der Simulation kann MaxControl Fähigkeiten des 3D-Animationsprogramms nutzen. So kann sich MaxControl darauf beschränken, nur die Füße eines Roboters zu bewegen und die Beine des Roboters können durch die inverse Kinematik des 3D-Animationsprogramms entsprechend bewegt werden. MaxControl kann auch Partikelsysteme des 3D-Animationsprogramms steuern und so seine eigenen Animationsfähigkeiten erweitern. Erst wenn die so insgesamt erstellte Animation den Vorgaben entspricht, muss sie durch Rendering in eine abspielbare Bildfolge umgewandelt werden. Werden hier unerwünschte optische Eigenschaften an den Einzelbildern entdeckt, die z.B. von den Materialeinstellungen der Objekte oder von Rendering-Einstellungen herrühren, können diese Einstellungen vor einem erneuten Rendering geändert werden, ohne dass die Simulation selbst durch MaxControl wiederholt werden muss. Diese Vorteile entsprechen auch einigen Forderungen in [Witt99]. Das dort vorgestellte System ist daher ebenfalls in eine bestehende Produktionsumgebung integriert, so dass unter anderem vorher vorhandene Szenenelemente für die Simulation genutzt werden können und Simulation und Rendering voneinander getrennt durchgeführt werden können, wie auch in [Fear00].

Da für MaxControl kein spezielles Konzept zur Spezifikation von Verhaltensweisen z.B. aus dem Bereich der künstlichen Intelligenz gewählt wurde, können hier beliebige Konzepte verwendet werden, auch die Wahl von hybriden Lösungen, die mehrere Konzepte parallel verwenden, ist möglich. Der Aufwand der Erstellung solcher Verhaltensweisen kann sich dadurch aber auch erhöhen, da die Unterstützung eines speziellen Konzeptes zunächst implementiert werden müsste. Für Java gibt es jedoch frei verfügbare Klassenbibliotheken und Werkzeuge, die beispielsweise sowohl die Verwendung von Fuzzy Methoden als auch von Neuronalen Netzen unterstützen⁸. Zu Fuzzy Methoden und Neuronalen Netzen siehe [Chen01]. Eine Nutzung dieser Konzepte ist also für die Definition von Verhaltensweisen auch auf der Basis von Java-Code möglich und könnte so in MaxControl verwendet werden.

Vergleichbare Werkzeuge für professionelle 3D-Animationsprogramme sind wie oben erwähnt meistens durch die Verwendung spezieller Konzepte für die Spezifikation des Verhaltens von Objekten in ihren Möglichkeiten eingeschränkt.

Insgesamt ist MaxControl nicht für eine bestimmte Gruppe von Animationen konzipiert, wie es z.B. häufig bei Systemen zur Animation von Figuren oder bei Partikelsystemen der Fall ist, sondern es ist als offenes erweiterbares System für eine Vielzahl verschiedener Arten von Animationen verwendbar und kann auch durch den Benutzer entsprechend ausgebaut werden.

Verschiedene für bestimmte Probleme entwickelte Lösungen wie Animationssysteme für Feuer, Rauch, Wasser, viele Figuren (=Crowds) oder Stoff sind aufgrund ihrer hohen Spezialisierung schwer miteinander in wechselseitige Interaktion zu bringen. Nur wenige Arbeiten beschäftigen sich mit der Integration mehrerer solcher Probleme in ein Animationssystem. Beispiele für solche Systeme sind [Clav05, Losa06, Melek05, Guend05, Losa06a, Sell05, Kacic03] (siehe Kap. 4.1.2), wobei auch hier die miteinander integrierten

⁸ Fuzzy Methoden: "Open source fuzzy inference engine for Java":
<http://people.clarkson.edu/~esazonov/FuzzyEngine.htm>,
http://people.clarkson.edu/~esazonov/neural_fuzzy/loadsway/LoadSway.htm, 15.03.2007
"FuzzyJ Toolkit": http://www.iit.nrc.ca/IR_public/fuzzy/fuzzyJToolkit2.html, 15.03.2007
Neuronale Netze: "NNDef Toolkit": <http://www.makhfi.com/nndef.htm>, 15.03.2007
"Java Neural Network Toolkit": <http://hei.unige.ch/projects/janet/>, 15.03.2007
„Weighscore Neural Network Toolkit“: <http://www.weighscore.com/features.php>, 15.03.2007

Phänomene jeweils auf eine bestimmte endliche Menge festgelegt sind. In MaxControl können alle Objekte in einer Szene miteinander interagieren, da sie wechselseitig auf die Eigenschaften der jeweils anderen Objekte zugreifen können, unabhängig davon, nach welchem Konzept sich diese Objekte verhalten. Ein durch Fuzzy Logic kontrolliertes Fahrzeug könnte also Figuren ausweichen, die durch hierarchische endliche Automaten gesteuert werden und die selbst auch dem Fahrzeug ausweichen. In Kapitel 10.4.1.6 wird eine MaxControl-Animation gezeigt, in der ein ohne die Berücksichtigung von Mechanik algorithmisch simulierter Roboter mechanisch simulierte Kartons umwirft und vorübergehend ein Fahrzeug festhält, das unter Berücksichtigung von Mechanik ebenfalls algorithmisch simuliert wird, indem es seine Position nicht direkt ändert sondern die dazu nötigen Kräfte erzeugt. Da sowohl dem Fahrzeug als auch dem Roboter die Position des jeweils anderen Objektes zugänglich ist, könnten sich beide bei entsprechender Implementation ihrer Verhaltensweisen auch gegenseitig ausweichen. So besteht für jede Verhaltensweise eines Objektes zumindest die Möglichkeit, auf beliebige andere Objekte zu reagieren oder diese zu beeinflussen, was in einem Zusammenspiel isolierter Systeme, die unter Umständen wie oben erläutert ihre Simulationsergebnisse nicht als Zustandsfolge ausgeben, sondern jeweils direkt eine Folge gerenderter Einzelbilder als Animation erzeugen, ungleich schwieriger zu realisieren ist.

Als Basisprinzip für die Animation starrer Körper wird bei MaxControl die Simulation von Mechanik verwendet. So entstehen automatisch komplexe und realistische Bewegungen, die sehr natürlich wirken. Sie sind in dieser Hinsicht Animationen überlegen, die wie in vielen vergleichbaren Tools aus mehreren vorgefertigten Animationsabläufen zusammengesetzt werden, da mechanisch simulierte Körper flexibler und realistischer auf die physikalische Umgebung und äußere Einflüsse wie externe Kräfte reagieren [Zord05, Reitsma03]. In Bezug auf Bewegungen stellt diese Herangehensweise für die Implementation von Verhaltensweisen eine höhere Herausforderung dar, weil z.B. die Bewegung eines Objektes an einen anderen Ort dann nicht einfach durch mehrere aufeinanderfolgende Positionsänderungen erfolgt, sondern stattdessen die für eine entsprechende Bewegung nötigen Kräfte berechnet und diese in der Simulation erzeugt werden.

Mittel, um diese Kräfte zu erzeugen, können bei einem Flugzeug Schubkräfte aus Düsentriebwerken oder bei einem Auto Drehmomente an den Rädern sein. Korrekte Verhaltensweisen sind so zwar schwieriger zu erstellen, dafür erhält man aber automatisch physikalisch realistische Bewegungen. Eine solche Herangehensweise wurde beispielsweise in [Tu99] mit Hilfe einer Simulation von Wasserdynamik für die Darstellung des Verhaltens von Fischen eingesetzt (siehe dazu auch [Tu94]).

Besonders bei der Animation von Maschinen wie Autos oder Robotern mit dieser Methode entstehen für die Implementation der Verhaltensweisen im Grunde ähnliche Probleme wie bei Steuerungsprozessen in der Robotik. Die Qualität und Flexibilität der so möglichen Animationen wird diesen Aufwand allerdings rechtfertigen.

Das Grundprinzip der Simulation der Mechanik starrer Körper schließt eine direktere Spezifikation von Bewegungen nicht aus. Objekttypen und Verhaltensweisen können auch so implementiert werden, dass die zugehörigen Objekte ihre Position und Rotation direkt verändern, ohne dabei die Simulation von Mechanik zu berücksichtigen.

Es ist auch die automatische Erzeugung von Tonspuren möglich, die zu der 3D-Animation passen. Da durch die Simulation des Verhaltens der Objekte genaue Daten über Ereignisse für diese Objekte bekannt sind, können auch Toneffekte zu passenden Zeitpunkten automatisch generiert werden. Beispiele hierfür sind Motorengeräusche, Triebwerksgeräusche, Toneffekte

bei Kollisionen unter Berücksichtigung der kollidierenden Materialien oder Toneffekte bei Explosionen. Da auch die dreidimensionale Position sowie die Geschwindigkeit der Tonquellen bekannt sind, können korrekte Surround-Sound⁹-Tonspuren und Effekte wie der Dopplereffekt ebenfalls automatisch erzeugt werden. Die Erzeugung von Surround-Sound für virtuelle Welten ist Thema der unten genannten Arbeiten [Funk98], [Funk99], [Naef02] und [Tsing01] und wird auch von MaxControl unterstützt. Bei herkömmlicher 3D-Animationstechnik müssen solche Toneffekte zumeist in der Nachbearbeitung mit hohem Aufwand manuell hinzugefügt werden. Es gibt zwar Arbeiten zur automatischen Erzeugung von Tönen zu Computeranimationen (z.B. [Dob03, OBrien02, Cardle03, Pai01, Doel01, Funk98, Funk99, Tsing01, Naef02]), diese beziehen sich jedoch größtenteils auf Echtzeitanwendungen wie z. B. Computer- und Videospiele, bei denen eine automatische Tongenerierung synchron zu den erzeugten Echtzeitanimationen nicht neu ist. Vergleichbar mit [OBrien02], [Doel01] und [Pai01] kann MaxControl auch die Kollisionsinformationen aus der Mechaniksimulation nutzen, um auf deren Basis passende Töne bei Rollbewegungen, bei Reibung oder bei Stossereignissen zu erzeugen.

Es seien nochmals die in der Einleitung erwähnten 3D-Animationen in Echtzeit angesprochen. Entsprechende Systeme, z.B. wissenschaftliche Simulationen oder Computerspiele, erzeugen automatisch eine Animation und können diese während ihrer Entstehung sofort anzeigen. Insbesondere Spielsimulationen müssen in Echtzeit interaktiv auf das Verhalten des Benutzers reagieren können. Dadurch sind vordefinierte Animationsabläufe nur bedingt einsetzbar und es müssen Programmteile implementiert werden, die auf das Benutzerverhalten reagierende Animationen in Echtzeit erzeugen können. Solche Systeme werden meistens in einer Programmiersprache wie C oder C++ implementiert, um unter anderem eine effiziente 3D-Darstellung der Animationen in Echtzeit zu erreichen. So liegt es nahe, auch die Verhaltensweisen der 3D-Objekte, die auf den Benutzer reagieren sollen, in dieser Programmiersprache zu implementieren. Verwendet man dabei eine objektorientierte Sprache wie C++, ist es eine natürliche Herangehensweise, auch bei der Implementation des Verhaltens der 3D-Objekte die objektorientierten Techniken der Sprache zu nutzen. Ähnlich geht z.B. das oben erwähnte Animationssystem ASAS vor, weitere vergleichbare objektorientierte Systeme sind [Gerv94, Zele91, Döll97, Getto90, Fiume87, Luck97/Müller98¹⁰]. Bei solchen meist echtzeitorientierten Systemen ist eine Verwendung objektorientierter Programmier Techniken für die Spezifikation des Verhaltens von 3D-Objekten, wie sie bei MaxControl zum Einsatz kommt, also durchaus üblich.

Jede Lösung zur Erstellung von Animationen in Echtzeit unterliegt jedoch Einschränkungen, die aus dem Echtzeitananspruch entstehen. Wie in Kapitel 1.2 erwähnt, sollten dabei möglichst 60 Einzelbilder pro Sekunde erzeugt werden, um einen flüssigen Eindruck zu vermitteln. Das bedeutet, dass die Anwendung 60-mal in der Sekunde für alle Objekte einen neuen Zustand berechnen und diesen dann als 3D-Darstellung anzeigen muss. Die Simulation des Verhaltens der Objekte kann nur eine begrenzte Zeit der für jedes Einzelbild zur Verfügung stehenden Rechenzeit beanspruchen. Dies ist nicht mit einer beliebigen Anzahl von Objekten möglich. Meistens ist die dreidimensionale Darstellung des neu erzeugten Zustandes der Szene der aufwändigste Teil dieses Prozesses. Auch hier entstehen Grenzen in der möglichen

⁹ Surround-Sound wird in der Fachliteratur auch als *Spatialized Sound* bezeichnet. Statt den 2 Tonkanälen für herkömmlichen Stereo-Ton werden 6 oder mehr Tonspuren erzeugt, die beim Abspielen auf ebenso viele im Raum angeordnete Lautsprecher verteilt werden. Dadurch kann der Zuhörer die Position einer virtuellen Tonquelle im Raum differenzierter erkennen.

¹⁰ Beide Texte erläutern dasselbe System.

Bildauflösung, der Anzahl gleichzeitig darstellbarer Objekte und weiterer Eigenschaften der 3D-Darstellung, welche die Qualität der Darstellung maßgeblich beeinflussen.

MaxControl ist im Kern einem solchen Echtzeitanimationssystem sehr ähnlich, ohne den oben genannten Einschränkungen von Echtzeitsystemen zu unterliegen. Jedoch ist es schon konzeptionell auf die Integration in ein professionelles 3D-Animationsprogramm ausgerichtet, um die Vorteile nutzen zu können, die solche Animationsprogramme gegenüber Echtzeitanimationssystemen haben. Somit verbindet MaxControl die Vorteile beider Arten von Animationssystemen. Es basiert auf den Ideen zur automatischen Animation von 3D-Objekten aus den Echtzeitanimationssystemen und nutzt in einem professionellen 3D-Animationsprogrammen dessen intuitive Benutzeroberfläche, seine erprobten Möglichkeiten zur Erstellung von 3D-Szenen und vor allem die überlegene Bildqualität der Software-Renderingsysteme.

Die Kernidee von MaxControl sei hier noch einmal zusammengefasst: MaxControl wurde als System zur automatischen Animation von nicht-interaktiven 3D-Animationsfilmen in ein professionelles 3D-Animationsprogramm integriert.

Neu ist an MaxControl die Vereinigung der folgenden Eigenschaften in einem System:

- Es werden die oben genannten Vorteile von Echtzeitsystemen für die Erstellung nicht-interaktiver 3D-Animationsfilme genutzt.
- Im Gegensatz zu vergleichbaren Systemen wie ASAS ist MaxControl enger in ein bestehendes 3D-Animationsprogramm integriert, so können vor, während und nach der Simulation Fähigkeiten dieses 3D-Animationsprogramms genutzt und anders erstellte Animationen integriert werden (siehe S. 28).
- Das Verhalten der 3D-Objekte wird über eine offene erweiterbare Architektur spezifiziert, basierend auf Java und der Simulation von Mechanik.
- Die Fähigkeiten von Java zur Objektorientierung werden bei der Definition von Objekttypen, die 3D-Objekten zugewiesen werden können, und den zugehörigen Verhaltensweisen systematisch genutzt.
- Objekttypen und ihre Verhaltensweisen sind kompositionell wieder verwendbar.
- Im Gegensatz zu den meisten objektorientierten Systemen tauschen die Objekte hier Informationen in der Regel nicht über Nachrichten sondern über direktere Datenzugriffe aus, was die oben (ab S. 25) erwähnten Vorteile hat.
- MaxControl enthält ein flexibles System zur gezielten manuellen Beeinflussung der Simulation.
- Die Verwendung von Keyframe-Techniken erlaubt ein effizientes Lesen der manuell kontrollierten Werte und eine effiziente Speicherung der erzeugten Animation im 3D-Animationsprogramm.
- Die mit MaxControl erstellbaren Animationen beschränken sich nicht nur auf die Bewegung von Objekten im Raum, sondern es können verschiedenste Eigenschaften animiert werden wie die Größe von 3D-Objekten, die Helligkeit von Lichtquellen, Tonhöhe und Lautstärke von Tonquellen oder Aktionen einer virtuellen Kamera.

- Das System ist weder auf bestimmte Animationsprobleme noch auf ein spezielles Lösungskonzept festgelegt, sondern verschiedene Lösungsansätze für unterschiedliche Animationsprobleme können in einem System **interagierend** miteinander arbeiten.

Insgesamt wird hier die Ausdruckstärke von Java genutzt, um in Kombination mit der Simulation von Mechanik „lebendige“ komplexe Welten mit verschiedenartigen autonomen interagierenden Objekten zu simulieren. In diesen Welten soll eine Handlung durch möglichst maßvolle gezielte Anleitung von außen fast von selbst ablaufen können.

Wichtig ist hierbei auch, dass keine Simulation in Echtzeit angestrebt wird, sondern dass Animationen innerhalb eines professionellen 3D-Animationsprogrammes durch Simulation automatisch erstellt werden sollen. Durch den fehlenden Echtzeitanspruch wird eine hohe Komplexität der Szenen möglich und durch die Anwendung eines professionellen 3D-Animationsprogrammes können die resultierenden Filme in einer grafisch hochwertigen Qualität erzeugt werden, wie es in Echtzeit auch mit spezieller 3D-Hardware noch nicht möglich wäre.

So entstand ein flexibles Animationssystem, das sich nicht auf bestimmte Lösungskonzepte und Problemstellungen aus dem Gebiet der Computeranimation beschränkt, wie z.B. die Animation von Figuren, sondern als offenes erweiterbares System die automatische Erstellung von Animationen ermöglicht, erleichtert und durch ein klares Simulationskonzept unterstützt. Durch die Simulation von Mechanik können dabei realistische Bewegungen erzeugt werden.

2.2 Architektur von MaxControl

Um den Entwicklungsaufwand möglichst gering zu halten und die Eigenentwicklungen auf die wirklich neuen Konzepte zu beschränken, wurden verschiedene bereits existierende Softwarekomponenten verwendet, die Aufgaben übernehmen, für die bereits gut entwickelte Lösungen existieren.

Als professionelles Basiswerkzeug für die Erstellung der 3D-Animationen wird 3D-Studio-MAX mit den folgenden Plugins verwendet: Final Render wurde als Renderingsystem gewählt, Foley Studio MAX übernimmt das Erzeugen von Tondateien auf der Basis von 3D-Tonquellen. Für das Rendern von speziellen Atmosphäreneffekten ist Afterburn zuständig.

Außerhalb von 3D-Studio-MAX wird für die Simulation von Mechanik das Produkt Shockwave von Macromedia verwendet.

Als Programmiersprache für die Erstellung des Hauptprogramms MaxControl und zur Spezifikation der Objekttypen und ihrer Verhaltensweisen wird Java verwendet, in Form des Java-Plugins der Firma Sun für den Internet-Explorer. Dabei findet auch die Java-Spracherweiterung „Java-3D“ Verwendung. Als Werkzeug zur Integration der Tools wurde der „Microsoft Internet Explorer“ gewählt. Abbildung 11 gibt einen Überblick über die Gesamtarchitektur von MaxControl.

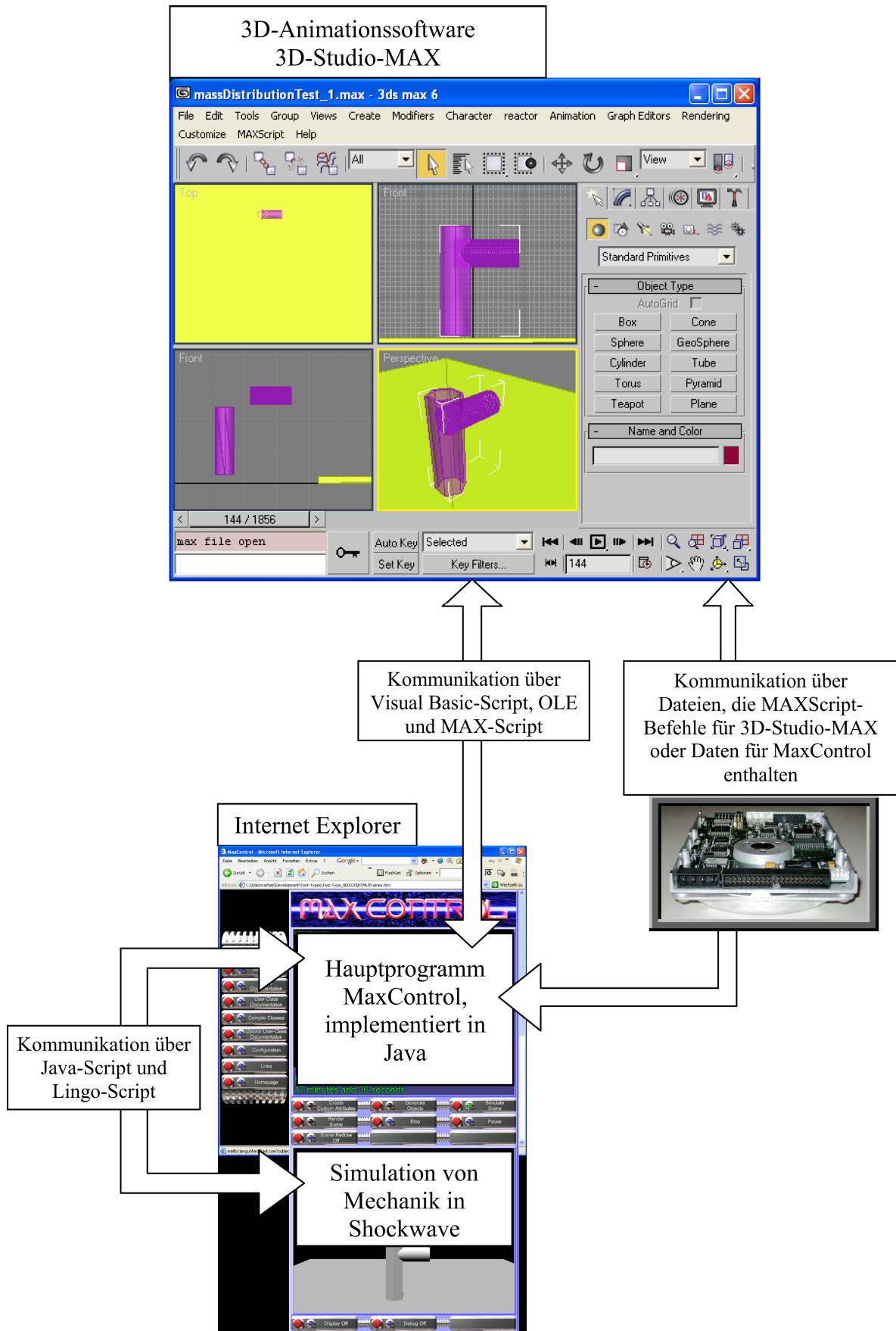


Abbildung 11: Architektur von MaxControl

Die verschiedenen Werkzeuge werden zentral von MaxControl gesteuert, und zwar übernimmt MaxControl die grundsätzliche Steuerung der Simulation, die Übertragung von 3D-Szenendaten in simulierbare Java-Datenstrukturen und die Kommunikation mit den beteiligten Werkzeugen. Ebenso wird die automatisch erstellte Animation von MaxControl effizient in die Szene geschrieben, die vom 3D-Animationsprogramm gespeichert wird. Dieses Schreiben einer Animation in Form von Animations-Keys wurde so konzipiert, dass automatisch möglichst keine redundanten Keys gesetzt werden, da dies zu einer längeren Kommunikationszeit, unnötiger Speicherbelastung und zu einer verringerten Leistung des 3D-Animationsprogrammes führen würde.

Die Kommunikation der Werkzeuge untereinander wird nicht über implementationsnahe Schnittstellen, die oft sehr versionsabhängig sind, sondern über die jeweils verfügbaren Skriptsprachen realisiert. Es hat sich gezeigt, dass die Befehle der Skriptsprachen bei neuen Programmversionen syntaktisch und semantisch meistens unverändert bleiben, während implementationsnahe Schnittstellen bei jedem Versionswechsel des befehlsempfangenden Programms oft erheblichen Änderungen unterliegen. Dies gilt insbesondere für 3D-Studio-MAX, dessen in C++ spezifizierte Schnittstellen für Plugins bei den meisten Versionen signifikant verändert werden, im Gegensatz zur zugehörigen Skriptsprache MAXScript.

Um den Simulationsablauf möglichst effizient zu halten, wurde besonderer Wert gelegt auf die Implementation einer schnellen aber auch laufsicheren Kommunikation zwischen den verwendeten Werkzeugen. Während einer Simulation müssen automatisch korrekte Programme in den Skriptsprachen erzeugt werden, damit sie als Befehle gesendet werden können. Diese Programme müssen effizient sein und sollten keine redundanten Befehle enthalten. Es mussten auch Unzulänglichkeiten der verwendeten Tools berücksichtigt werden, wie fehlende dynamische Speicherreservierung bei MAX-Script oder Instabilitäten bei der Übermittlung zu langer Programme an 3D-Studio-MAX und Macromedia Shockwave.

Für MaxControl wurde ein klares Simulationskonzept entworfen, in dem unter anderem festgelegt wird, wie das Verhalten von Objekten simuliert wird, wie Verhaltensweisen die Eigenschaften von Objekten ändern können und wie sich die Simulation von Mechanik in die Verhaltensweisen einfügt. Für dieses Simulationskonzept wurden Klassenstrukturen entwickelt, die bei der Spezifikation von Verhaltensweisen verwendet werden. Jedem 3D-Objekt in der Szene wird ein Objekttyp zugeordnet, der im Allgemeinen eine hierarchische Struktur von Verhaltensweisen enthält. Sowohl Objekttypen als auch Verhaltensweisen werden als Java-Klassen spezifiziert, die auf einem festen Bestand von Basisklassen mit sinnvollen Datenstrukturen und Methoden aufbauen.

MaxControl enthält in der aktuellen Version bereits eine Reihe von Objekttypen und Verhaltensweisen, die auf diesen Basisklassen aufbauen und mit denen beispielsweise Lichtquellen oder Tonquellen in einer Szene gesteuert werden können. Solche Klassen können durch die objektorientierte Struktur von Java bequem erweitert oder wieder verwendet werden, um neue oder erweiterte Objekttypen und Verhaltensweisen zu spezifizieren.

Für die Simulation von Mechanik wurde zwar Macromedia Shockwave als Komponente verwendet, jedoch waren umfangreiche Eigenentwicklungen nötig, um spezielle Probleme in diesem Bereich zu lösen. Shockwave ist ein Werkzeug zur Entwicklung von webbasierten Anwendungen und unterstützt dabei auch die Simulation des mechanischen Verhaltens von starren Körpern. Die in Shockwave entwickelte Anwendung zur Simulation von Mechanik enthält eine spezielle Schnittstelle zur Kommunikation mit dem Hauptprogramm MaxControl. Da in Shockwave eine direkte Unterstützung von Dreh- und Schiebegelenken fehlt, werden

Drehgelenke durch automatisch von MaxControl erzeugte Federn nachgebildet, die in Shockwave simuliert werden können. Ein analoges Vorgehen ist für Schiebegelenke möglich.

Ein großes Problem stellt bei der Simulation von Mechanik auch die Stabilität der Simulation dar, insbesondere bei der Simulation von harten Federn, die für die hier gewählte Methode zur Simulation von Dreh- und Schiebegelenken in Shockwave jedoch erforderlich sind. Bei einer instabilen Simulation können z.B. zu hohe Federkräfte auftreten und zu sichtbar fehlerhaften Simulationsverläufen führen, in denen sich die simulierten Objekte explosionsartig voneinander weg bewegen. Die Stabilität kann oft nur durch zusätzliche Simulationszwischenritte erhalten werden. Diese erhöhen jedoch auch die Simulationszeit. Shockwave arbeitet eigentlich mit einer festen Anzahl an Simulationszwischenritten, obwohl sich die Anzahl nötiger Zwischenritte im Verlauf einer Simulation erheblich ändern kann. Deshalb wurde ein System entwickelt, das die Anzahl der Zwischenritte automatisch regelt, um eine Balance zwischen Simulationsstabilität und Ausführungszeit herzustellen. Dabei wurden verschiedene Techniken entwickelt, die vom Benutzer nun gewählt und durch die Veränderung von Parametern den jeweiligen Erfordernissen angepasst werden können.

Weiterhin mussten Methoden zur effizienten Kommunikation mit Shockwave entwickelt werden, um keine redundanten Daten zu übermitteln. Für diese Kommunikation war zusätzlich die Entwicklung spezieller Mechanismen erforderlich, um Instabilitäten und Fehler von Shockwave zu umgehen oder auszugleichen.

Das Konzept von MaxControl sieht eine klare Trennung der Aufgabenstellung für mögliche Anwender vor. Die Spezifikation von Objekttypen und Verhaltensweisen durch Java-Klassen sollte durch Programmierer durchgeführt werden, während deren praktische Anwendung auch für Künstler möglich ist, die nicht über den Bereich der manuellen Erstellung von 3D-Computeranimationen hinaus vorgebildet sind. Dies wird durch die Unterstützung der grafischen Benutzeroberfläche des hier als Basiswerkzeug verwendeten Programms „3D-Studio-MAX“ möglich. In dieser Benutzeroberfläche kann ein Künstler wie gewohnt mit den 3D-Objekten arbeiten und ihnen die spezifizierten Verhaltensweisen zuweisen. Ferner kann er innerhalb der Benutzeroberfläche durch Veränderungen von Parametern die Verhaltensweisen bequem seinen Erfordernissen anpassen und die Möglichkeiten zur manuellen Kontrolle der Simulation nutzen.

Auch für den Programmierer wurde ein einfaches übersichtliches Spezifizieren der Verhaltensweisen ermöglicht. Alle Datenschnittstellen und Simulationskomponenten sind so in eine abstrahierende Interfacestruktur gekapselt, dass der Programmierer die technische Umsetzung der Kommunikation zwischen den verwendeten Werkzeugen und die technische Umsetzung der Simulation selbst, insbesondere im Bereich Mechanik, weder genau kennen noch näher beachten muss.

Durch diese Kapselung ist auch ein Austausch von Softwaretools möglich, ohne gravierende Änderungen an den Benutzerschnittstellen vornehmen zu müssen. So wurde neben der mit Shockwave entwickelten Anwendung zur Simulation von Mechanik experimentell eine rudimentäre Verbindung des Systems mit ODE¹¹ hergestellt, einer anderen freien Software zur Simulation von Mechanik. Sollte eines der bisher verwendeten Werkzeuge also nicht mehr weiterentwickelt werden oder aus anderen Gründen nicht mehr verfügbar sein, so kann MaxControl grundsätzlich für die Nutzung anderer vergleichbarer Komponenten angepasst werden, ohne grundlegende Änderungen an den Benutzerschnittstellen vornehmen zu müssen.

¹¹ <http://ode.org/>, <https://odejava.dev.java.net/>, 26.03.2007

Dies erhöht die Zukunftssicherheit des Systems. Diese Möglichkeit könnte auch zur Erhöhung der Flexibilität genutzt werden, indem z.B. mehrere Werkzeuge für jeweils die gleiche Aufgabe in MaxControl integriert und dann passend zu den Anforderungen gewählt werden. So wäre auch eine Unterstützung anderer professioneller 3D-Animationsprogramme neben 3D-Studio-MAX denkbar, wie z.B. Maya oder Softimage.

In den praktischen Anwendungen hat sich gezeigt, dass mit MaxControl sehr lange Animationen von komplexen Szenen erzeugt werden können, die dabei frei von unerwünschten Wiederholungen sind. Die so herstellbaren Filme wären mit manuellen Keyframe-Techniken nur mit kaum vertretbarem Aufwand machbar gewesen. Insbesondere die Animation mit mechanisch realistisch simulierten Fahrzeugen wäre auch mit den meisten anderen Automatisierungstools wie Partikelsystemen oder Systemen zur Animation von Figuren nicht herstellbar gewesen, unter anderem weil viele dieser Werkzeuge zu sehr auf spezielle Probleme ausgerichtet sind. Die offene Architektur von MaxControl unterstützt dagegen auch solche Animationen und kann für weitere Themen durch die Implementierung neuer Objekttypen und Verhaltensweisen ausgebaut werden.

3 Begriffsdefinitionen

3.1 3D-Objekt, Objekt

Der Begriff Objekt hat in diesem Text zwei mögliche Bedeutungen. Da ein zentrales Thema dieser Arbeit 3D-Animationen sind, ist mit einem Objekt meistens ein 3D-Objekt in einer 3D-Szene gemeint. Im Zusammenhang mit den hier verwendeten Programmiersprachen, insbesondere Java, werden die Instanzen einer Klasse auch als Objekte bezeichnet. Zur Verdeutlichung werden im Zweifelsfall Objekte in einer 3D-Szene als 3D-Objekte bezeichnet, während Objekte als Instanzen von Klassen in Java dann Java-Objekte genannt werden.

Es wird nun der Begriff 3D-Objekt genauer erläutert. Ein dreidimensionales Objekt ist die abstrahierte mathematische Repräsentation eines Körpers, der oft (aber nicht zwingend) ein reales Objekt nachbilden soll. Klassisch werden 3D-Objekte in Computerprogrammen aus Punkten im dreidimensionalen Raum zusammengesetzt, die dann zu Kanten verbunden werden. So entstehen Flächen, welche die Oberfläche des dreidimensionalen Objektes bilden. Diese Flächen werden auch als Polygone bezeichnet (siehe [Watt01]). Zur Vereinfachung werden die Flächen oft auf Dreiecksflächen beschränkt.

Es gibt auch andere Methoden zur Repräsentation von Körpern, z.B. Image-Based-Rendering, Volumetric-Rendering (siehe Kap. 5.5) oder Nurbs. Bei der Verwendung von Nurbs werden die Oberflächen eines Körpers durch mathematisch definierte Kurven erzeugt statt durch ebene Flächen wie bei klassischer Dreiecksflächenrepräsentation. Für genauere Erläuterungen zu Nurbs sei auf [Piegl91] verwiesen. Arbeiten, die auf Nurbs basieren, sind z.B. [Terz94, Qin96]. Die Repräsentation durch Dreiecksflächen, die neueren Techniken wie Nurbs in einigen Bereichen unterlegen ist, ist allerdings noch sehr weit verbreitet.

Darstellung einer Pyramide durch Dreiecksflächen:

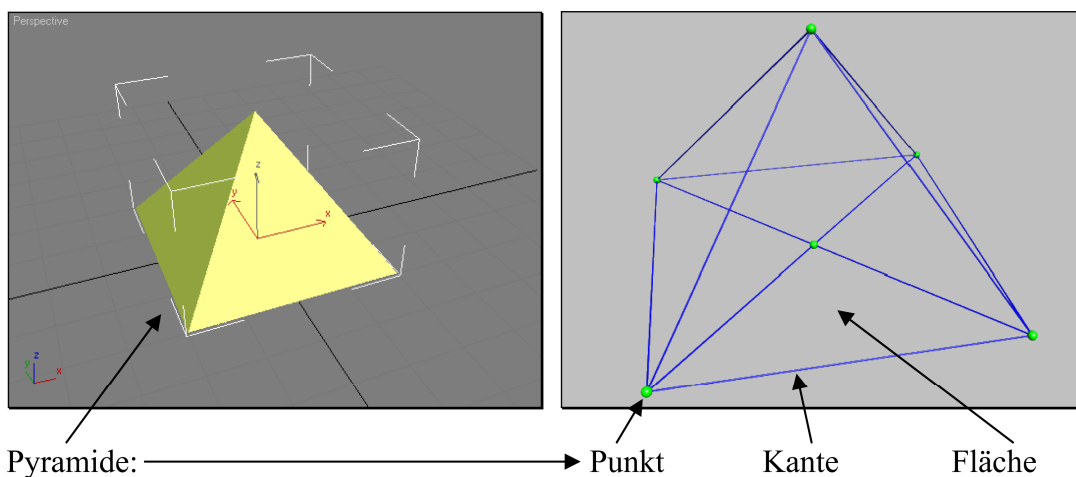


Abbildung 12: Darstellung einer Pyramide durch Dreiecksflächen

3.2 Transformation

Die Position, Rotation und Skalierung jedes Objektes im Raum wird durch seine zugehörige Transformation angegeben. Transformationen und ihre Matrix-Repräsentationen werden ausführlich in [Foley97] erläutert. Jede Transformation erzeugt ein eigenes Koordinatensystem, dem weitere Transformationen untergeordnet werden können. Dies wird für die Eltern-Kind-Beziehungen in der Szenenhierarchie verwendet (siehe folgendes Kapitel 3.3). Jedes Objekt besitzt durch seine Transformation also ein eigenes Koordinatensystem, in dem beispielsweise lokale Punkte innerhalb des Objektes angegeben werden können, die

zusammen mit dem Objekt verschoben, rotiert oder skaliert werden, sobald sich die Transformation des Objektes ändert. Solche Punkte behalten dabei also ihre Position relativ zum Objekt.

Die Transformation eines Objektes besteht aus einer Translation, welche die Position eines Objektes im übergeordneten Koordinatensystem angibt, einer Rotation, welche die Ausrichtung des Objektes angibt, sowie einer Skalierung. Die Rotation rotiert das zugehörige Objekt um seinen Mittelpunkt. Die Rotationsachsen werden dabei relativ zum übergeordneten Koordinatensystem angegeben. Auch die Skalierung skaliert das Objekt von seinem Mittelpunkt aus, wobei die Skalierung verschiedene Werte für jeweils die X-, Y- und Z-Achse des übergeordneten Koordinatensystems enthalten kann. Die Wirkungsweise der drei Eigenschaften der Transformation ist so festgelegt, dass zuerst die Skalierung durchgeführt wird, danach die Rotation und dann die Translation. Das Ergebnis der Transformation ist von dieser Reihenfolge abhängig.

Um festlegen zu können, wie ein Objekt durch die Transformation verändert wird, muss auch ein Mittelpunkt des Objektes bekannt sein. Dieser Mittelpunkt verfügt über eine eigene Position und Ausrichtung innerhalb des Objektes und wird als „Pivot-Punkt“ bezeichnet. Die gesamte Transformation bezieht sich auf diesen Pivot-Punkt. Die Wirkung einer Transformation ist im Hinblick auf den Pivot-Punkt so zu sehen, dass das Objekt zusammen mit seinem Pivot-Punkt zunächst so transformiert wird, dass der Pivot-Punkt im Ursprung des übergeordneten Koordinatensystems liegt und nicht rotiert ist. Dann wird das Objekt entsprechend der Transformation skaliert, dann rotiert und danach verschoben, wobei der Pivot-Punkt immer dieselbe Position und Ausrichtung innerhalb des Objektes behält.

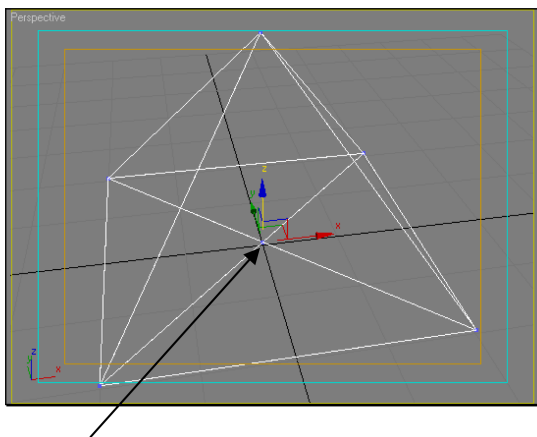


Abbildung 13: Pivot-Punkt der Pyramide aus Kapitel 3.1

3.3 3D-Szene, Szene, Szenenzustand

In einer 3D-Szene (=Szene) wird eine Gruppe dreidimensionaler Objekte zusammengefasst, mit ihren Eigenschaften und äußeren Erscheinungsformen, mit ihren jeweiligen Bezeichnungen, ihrer räumlichen Anordnung zueinander in einem Koordinatensystem und ihrer hierarchischen Ordnung in Ober- und Unterobjekte. Eine Szene kann weitere globale Eigenschaften enthalten sowie Objekte, die nicht zur dreidimensionalen Darstellung konzipiert sind. Darauf wird weiter unten näher Bezug genommen.

Eigenschaften können entweder zu Objekten gehören oder auch global für die Szene gelten. Als Eigenschaften werden hier Attribute bezeichnet, die ihrem Datentyp entsprechende Werte tragen können. Gehört zu einem 3D-Objekt, das die Form einer Kugel hat, ein Attribut „Radius“, so kann dieses Attribut mit einem Wert belegt werden, der bei entsprechender Semantik den Radius der Kugel bestimmt.

Eine 3D-Szene wird oft als Baum gesehen, in dem die 3D-Objekte die Knoten bilden und die Kanten eine Ordnung der Objekte in Ober- und Unterobjekte definieren. Dieser Baum wird auch als Szenengraph oder Szenenhierarchie bezeichnet. Sein Wurzelknoten repräsentiert die gesamte 3D-Szenenstruktur, denn alle 3D-Objekte in der Szene sind direkte oder indirekte Nachfolger dieses Knotens. Der Wurzelknoten ist ein abstraktes Objekt und gehört nicht zum sichtbaren Teil der Szene. Eine solche hierarchische Objektordnung für den Aufbau einer Szene wird auch in [Plenge88] vorgeschlagen und ist auch Teil des VRML-97-Standards (siehe [Kloss98]). Eine ähnliche Sicht einer Hierarchie zwischen 3D-Objekten wird in [Foley97] angegeben, während eine Szenenhierarchie nach [Bung02] eher 3D-Objekte in grafische Grundobjekte wie Kreise und Linien aufteilt, statt vollständige 3D-Objekte wie hier in eine Hierarchie zu bringen. Die Szenenhierarchie ist eine binäre Relation über der Menge aller 3D-Objekte.

Durch die hierarchische Ordnung können komplexe 3D-Objekte definiert werden. Die Finger einer Hand können z.B. dem Handrücken als Unterobjekte zugeordnet werden. Im Szenengraphen sind die Finger dann direkte Nachfolger des Handrückens. In den meisten 3D-Animationsprogrammen hängen solche Objekte automatisch zusammen, eine Bewegung des Handrückens bewegt dann die Finger entsprechend mit, während die Finger jedoch relativ zur Position und Ausrichtung des Handrückens unabhängig voneinander bewegt werden können (siehe Abbildung 14).

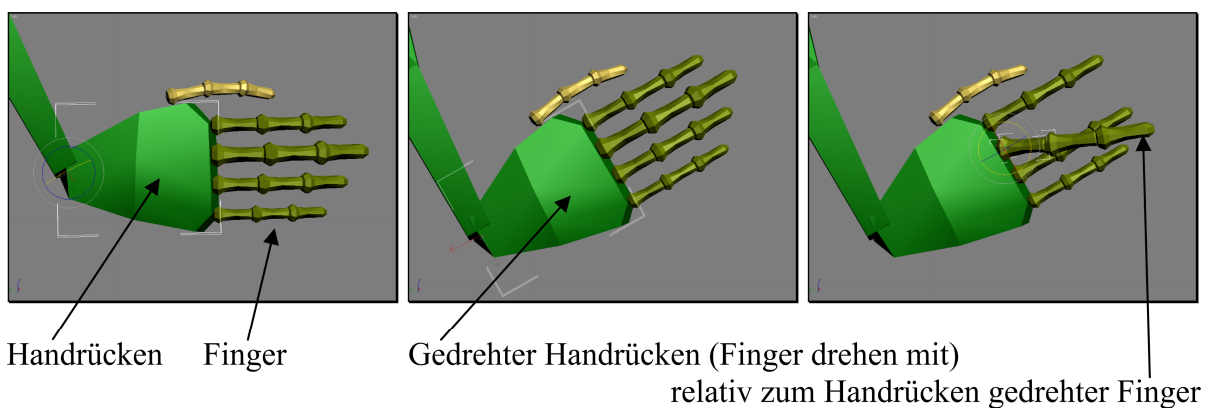


Abbildung 14: Hand aus hierarchisch verknüpften Einzelobjekten

Es folgt eine schematische Darstellung der hierarchischen Ordnung der oben dargestellten Hand und ihrer Finger. Die Pfeile (↓) zeigen von den Oberobjekten zu den Unterobjekten.

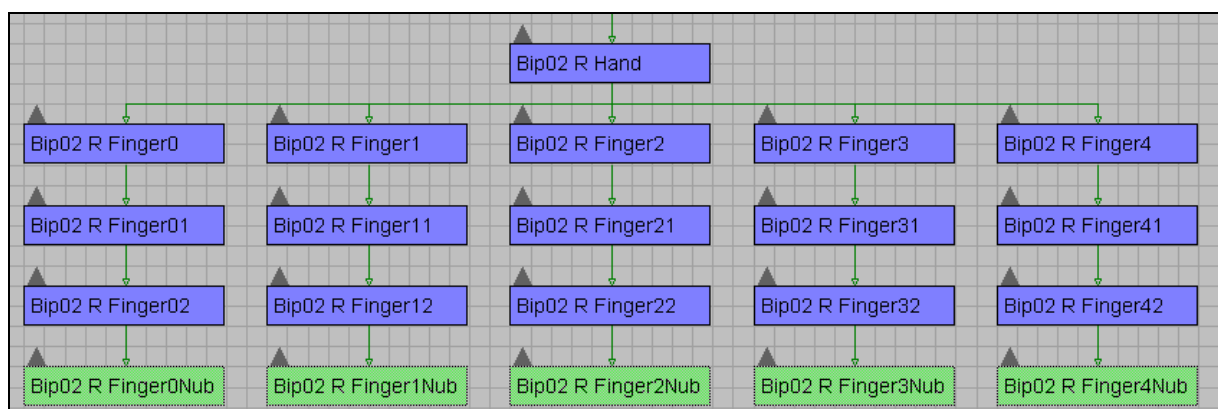


Abbildung 15: Baumdarstellung von hierarchisch verknüpften Einzelobjekten

Direkte Nachfolger eines Knotens werden hier als dessen Kinder, Kindobjekte oder Unterobjekte bezeichnet und direkte Vorgänger als Eltern, Elternobjekte oder Oberobjekte.

Die räumliche Anordnung der Objekte im Ursprungskoordinatensystem, das dem Wurzelknoten zugeordnet ist, wird durch die Transformationen der Objekte festgelegt. Das Ursprungskoordinatensystem wird auch Weltkoordinatensystem genannt. Translationen relativ zu diesem Koordinatensystem werden als Welt-Koordinaten bezeichnet.

Dabei wird die Transformation eines Kind-Knotens üblicherweise im Koordinatensystem des entsprechenden Eltern-Knotens interpretiert. Die tatsächliche Transformation eines Objektes in Welt-Koordinaten erhält man dadurch, dass man zunächst im Ursprungskoordinatensystem die Transformation dieses Objektes durchführt und danach die Transformationen aller übergeordneten Eltern-Knoten bis zum Wurzelknoten auf dieses Objekt anwendet, wobei die entsprechende Reihenfolge einzuhalten ist. Dies entspricht einer Multiplikation der Matrix-Repräsentationen dieser Transformationen in umgekehrter Reihenfolge vom Wurzelknoten bis zum Objekt (zur Multiplikationsreihenfolge siehe [Jack06], zu genaueren Formeln für einen Wechsel des Koordinatensystems durch Multiplikation der Transformations-Matrizen siehe [Foley97]). Da der Wurzelknoten untransformiert im Ursprung des Weltkoordinatensystems liegt, ist seine Transformation dabei nicht anzuwenden, meistens ist ihm auch gar keine eigene Transformation zugeordnet, da er nur aus strukturellen Gründen existiert und kein sichtbares 3D-Objekt repräsentiert.






Es ist zu beachten, dass schon die erste direkt zum Objekt gehörende Transformation den Pivot-Punkt des Objektes mit dem Objekt zusammen verschieben und drehen kann. Eine nachfolgend angewendete Transformation eines direkten oder indirekten Eltern-Knotens kann das Objekt nicht nur skalieren, sondern es zusammen mit seinem Pivot-Punkt auch weiter verschieben, da die Skalierung nun vom Ursprung des dem Objekt **übergeordneten** Koordinatensystems des Eltern-Objektes ausgeht und somit auch die Position des Pivot-Punktes des Objektes skaliert wird. Ebenso kann eine nachfolgend angewendete Rotation eines Eltern-Objektes das Objekt nicht nur rotieren, sondern es zusammen mit seinem Pivot-Punkt ebenfalls weiter verschieben, da diese Rotation das Objekt nun um den Ursprung des dem Objekt **übergeordneten** Koordinatensystems des Eltern-Objektes rotiert und somit auch die Position des Pivot-Punktes rotiert wird.

Diese Form der Transformationsvererbung führt dazu, dass bei einer Verschiebung, Skalierung oder Rotation eines Eltern-Objektes auch alle direkten und indirekten Kinder mit transformiert werden, so als würden das Eltern-Objekt und seine Kinder ein fest zusammenhängendes Objekt bilden (siehe obige Abbildung 14). Dieser feste Zusammenhang gilt allerdings nur bei einer Änderung der Transformation des Eltern-Objektes, die Kind-Objekte können sich relativ zu ihren Oberobjekten bewegen, wobei diese Bewegungen jedoch immer im Koordinatensystem des übergeordneten Knotens im Szenengraphen stattfinden. In [Jack06] werden auf diese Weise kinematische Ketten definiert, die z.B. mit Gelenken verbundene Teile einer Figur darstellen, beispielsweise Arme und Beine.

Zur Szene gehören in vielen 3D-Animationsprogrammen auch Objekte, die nicht dreidimensional dargestellt werden sollen, die aber Daten über 3D-Objekte oder über die gesamte Szene enthalten. So bilden Materialien eine Menge von Objekten, welche die Erscheinungsform von Oberflächen festlegen. Wird einem 3D-Objekt nun ein solches Material-Objekt zugeordnet, erhält dieses 3D-Objekt bei seiner Darstellung die entsprechende Oberfläche. Ebenso kann es eine Menge von Effektoobjekten geben, die z.B. in Verbindung mit Lichtquellen sichtbare Scheinwerferlichtkegel oder Rauchwolken darstellen können, ohne dabei ein 3D-Objekt im eigentlichen Sinn darzustellen.

Analog dazu gibt es in Szenen auch globale Eigenschaften, die keinem speziellen Objekt zugeordnet sind. So kann eine so genannte „World Scale“ angegeben sein, die festlegt, wie die Szeneneinheiten mit echten Maßeinheiten wie „Meter“ zusammenhängen sollen. Oder es wird eine Farbe bestimmt, die als „Ambient Light“ eine Grundbeleuchtung angibt, die auch ohne weitere Lichtquelle auf jedes Objekt aus jeder Richtung wirkt.

Auch diese zusätzlichen Objekte und Eigenschaften können als Knoten in einer baumförmigen Struktur angeordnet sein und in die Gesamtstruktur des Szenengraphen integriert werden, wobei dann einige Knoten nur zur Gruppierung von Unterknoten dienen. Einem Knoten untergeordnete Eigenschaften können daher auch als Untereigenschaften bezeichnet werden. Die Baumstruktur des so erweiterten Szenengraphen umfasst die Zusammenhangsrelation der 3D-Objekte. Sie wird weiterhin Szenenhierarchie genannt und ihr ursprünglicher Wurzelknoten bleibt als Wurzelknoten erhalten. Baumstrukturen von Eigenschaften sind in Abbildung 16 enthalten.

Zur Verdeutlichung des Szenengraphen wird in Abbildung 16 und Abbildung 17 jeweils ein Ausschnitt aus der Baumdarstellung einer Szene in 3D-Studio-MAX angegeben. Es werden nicht alle Unterobjekte und -eigenschaften angezeigt. Ein -Symbol oder ein -Symbol vor einem Knoten bedeutet, dass die zugehörigen **Untereigenschaften** bzw. die zugehörigen **Unterobjekte** in dieser Darstellung **nicht** angezeigt werden. Bei den Symbolen  und  werden die jeweiligen Unterstrukturen angezeigt. Symbole der Form  zeigen an, dass es sich bei dem jeweiligen Objekt um ein 3D-Objekt handelt.

Bisher haben wir uns mit der Struktur einer Szene beschäftigt, die Werte von Eigenschaften wurden weitgehend außer Acht gelassen. Belegen wir die Eigenschaften einer Szene mit Werten aus den entsprechenden Wertebereichen, entsteht ein Szenenzustand. Analog können wir vom Zustand eines Objektes sprechen. Szenenzustände werden im Folgenden meistens total definiert sein. In ihnen ist jeweils auch die Szenenstruktur festgehalten.

Für viele der oben beschriebenen Eigenschaften können zeitliche Veränderungen definiert werden, die dann eine Animation der Szene erzeugen. Zu diesen dynamisch veränderbaren Eigenschaften gehören die Transformationen der 3D-Objekte und weitere Eigenschaften wie der Radius einer Kugel oder globale Eigenschaften wie das oben erwähnte „Ambient Light“.

Ziel einer Animation ist ein abspielbarer Film. Deshalb benötigen wir eine Folge von Szenenzuständen, deren zeitliche Abstände dem zeitlichen Raster der Einzelbilder des Films entsprechen. Diese Folge von Szenenzuständen wird ebenfalls Animation genannt. Die Animation wird meistens auch als zur Szene gehörend angesehen und in fast jedem 3D-Animationsprogramm zusammen mit der Szene in einer Datei gespeichert. Eine 3D-Szene kann man daher aufteilen in eine **statische** Struktur, die hier als Szenenstruktur bezeichnet wird und festlegt, welche Objekte und Eigenschaften sich in welcher hierarchischen Struktur grundsätzlich in der Szene befinden, und zugehörige Daten über die **dynamischen** Veränderungen der Objekte und der globalen Eigenschaften im zeitlichen Ablauf des herzustellenden Films, die daher auch als Animation bezeichnet werden.

Aufgrund der Iterationsverfahren zur Herstellung der Animation benötigen wir im Allgemeinen Verfeinerungen des durch die Einzelbilder des Films bestimmten Rasters durch Simulationszwischenstufen. Die zugehörigen Zwischenzustände werden nicht im 3D-Animationsprogramm gespeichert.

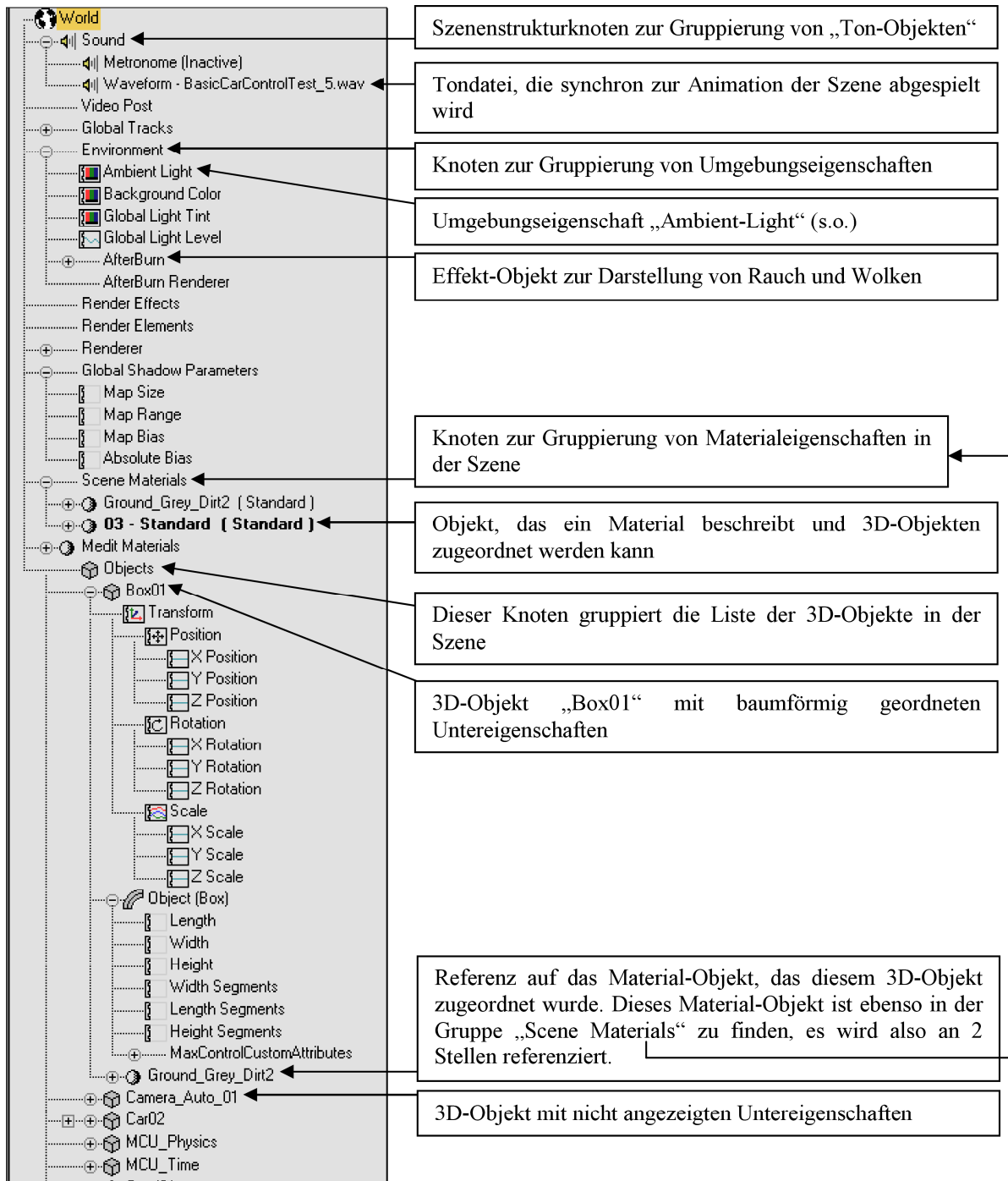


Abbildung 16: Szenenstruktur in 3D-Studio-MAX

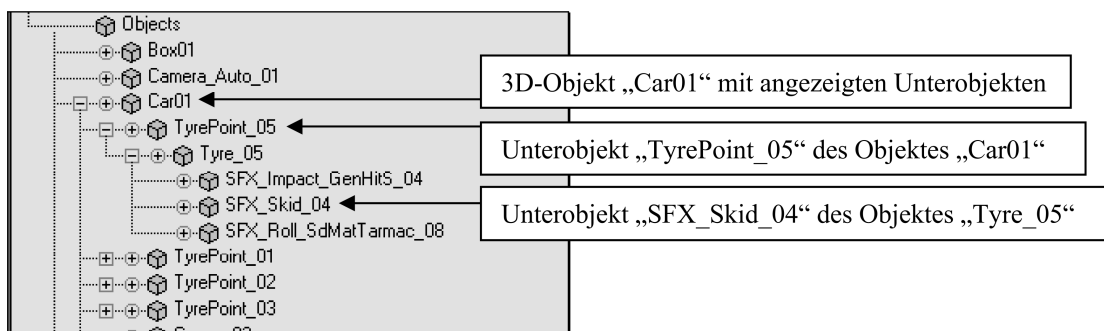


Abbildung 17: Hierarchie von 3D-Objekten in 3D-Studio-MAX

Bei den meisten 3D-Animationsprogrammen bleibt der Szenengraph statisch. Die zeitlichen Veränderungen betreffen nur die Werte von Eigenschaften. Dies bedeutet insbesondere, dass die Menge der Objekte einer Szene durch die Animation nicht verändert werden kann. Im zeitlichen Ablauf der Animation können also keine neuen Objekte hinzukommen oder bisher existierende Objekte gelöscht werden. Eine scheinbare Ausnahme bilden hier Partikelsysteme, bei denen dynamisch neue Objekte (=Partikel) erzeugt werden können. Jedoch wird ein solches Partikelsystem in der Szene im Allgemeinen als eigenständiges Objekt gesehen, das die von ihm dynamisch erzeugten Objekte selbst verwaltet. Die Objekte, die ein Partikelsystem erzeugt, sind damit nie direkt Objekte der Szene. Sie können aber zusammen mit der 3D-Szene dargestellt werden.

Wird hier von einer 3D-Szene gesprochen, ist meistens eine 3D-Szene gemeint, die im 3D-Animationsprogramm 3D-Studio-MAX gespeichert ist. Dieses Animationsprogramm wird in der konkreten technischen Umsetzung (ab Kapitel 10) als wichtige Programmkomponente verwendet, welche die Szene unter anderem speichert und dem Benutzer die Erstellung und Manipulation der Szene ermöglicht.

3.4 3D-Animation, Animation, animierbar, Frame

Als Animation bezeichnet man die Festlegung zeitlicher Veränderungen einer ganzen Szene, einzelner Objekte oder einzelner Eigenschaften von Objekten in der Szene, siehe dazu [Foley93]. Indem man für ein Zeitintervall einen Verlauf der Werte von Objekteigenschaften wie Position, Rotation oder Farbe festlegt, können zeitliche Veränderungen für die Objekte einer Szene definiert werden.

Dabei gibt es für jede Animation jeweils eine dem Zeitraster des geplanten Films entsprechende fest definierte Einteilung des Zeitintervalls durch äquidistante Zeitpunkte. Jedem Zeitpunkt ist ein Szenenzustand zugeordnet. Werden im Zeitintervall aufeinander folgende Szenenzustände der Szene als Bilder dargestellt und so gespeichert, entsteht ein Computeranimationsfilm. Das Zeitintervall der Animation wird hier auch als Animationsintervall bezeichnet.

Die aus den Szenenzuständen erzeugbaren Bilder, die zusammengesetzt den Film bilden, werden auch als Einzelbilder oder Frames der Animation bezeichnet. Die Frames sind durchnummeriert, üblicherweise beginnend bei 0. Die zu den Einzelbildern gehörenden Zeitpunkte im Animationsintervall werden ebenfalls als Frames bezeichnet. Mit „Frame 0“ kann also das erste Einzelbild eines Computeranimationsfilms bezeichnet werden, ebenso aber auch der Zeitpunkt oder Zustand, der in der zugrunde liegenden 3D-Szene diesem Einzelbild zugeordnet ist.

Bei einer Animation können nicht unbedingt für alle Eigenschaften der Objekte zeitliche Veränderungen festgelegt werden. Für welche Eigenschaften dies möglich ist, hängt vom verwendeten 3D-Animationsprogramm ab. Im Allgemeinen stellt es eine nicht erwünschte Einschränkung dar, wenn für einige Eigenschaften solche zeitlichen Veränderungen nicht definiert werden können. Solche Eigenschaften werden hier auch als nicht-animierbare Eigenschaften bezeichnet. Die Werte solcher Eigenschaften können zwar verändert werden, sie bleiben dann über das gesamte Animationsintervall konstant. Eigenschaften, für die beliebige zeitliche Veränderungen festgelegt werden können, heißen hier animierbar.

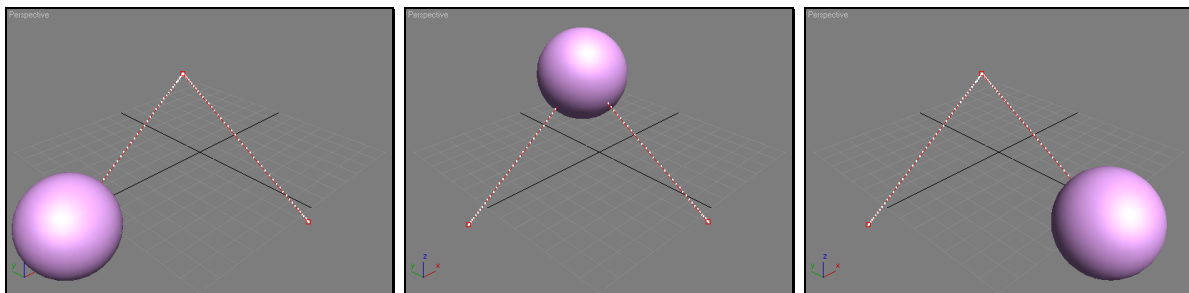
3.5 Key, Animations-Key, Keyframe-Animation

Wird eine Animation wie bisher meistens üblich manuell erzeugt, so werden die Werteverläufe von Objekteigenschaften oft über Schlüsselwerte, so genannte Keys, festgelegt.

Dadurch ist es nicht erforderlich, für jedes Einzelbild der Animation einen Wert für die animierte Eigenschaft festzulegen und diesen in der Szene zu speichern, sondern es reicht, nur für einige Frames der Animation Werte in Keys festzulegen. Die Werte für die anderen Frames werden dann durch Interpolation zwischen den Keys bestimmt.

Frames, die für eine Eigenschaft einen Animations-Key enthalten, werden auch als Keyframes bezeichnet. Die Methode zur Festlegung von Animationen mit Hilfe von Keys wird auch Keyframe-Animation genannt (siehe dazu [Foley93]).

Es folgen die Darstellungen einer Beispielanimation und zugehöriger Werteverläufe der Positions-Komponenten in 3D-Studio-MAX:



Abbildungen 18: Drei Phasen einer Beispielanimation

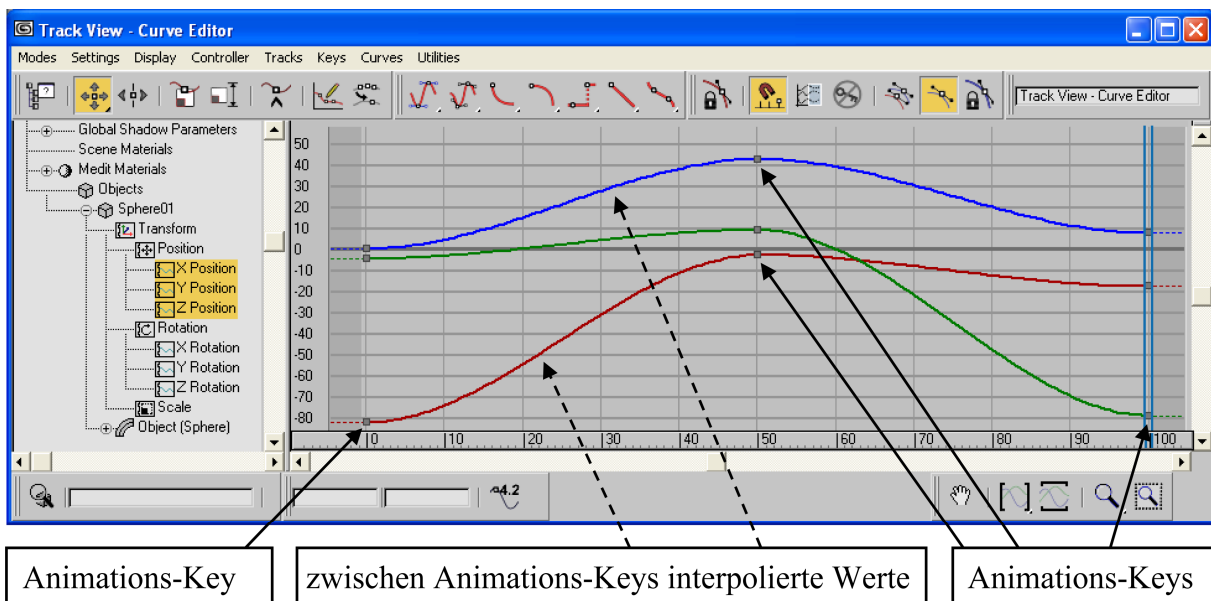


Abbildung 19: Darstellung von Keys und Werteverläufen in 3D-Studio-MAX

Die Positionen der Kugel werden durch die Animations-Keys und die durch Interpolation entstehenden Werteverläufe für die X-, Y- und Z-Komponenten der Positionen festgelegt. Diese Werteverläufe werden auch Animations-Tracks genannt. Ein Animations-Track einer animierbaren Eigenschaft kann als Funktion über der Zeitachse gesehen werden, die in Abhängigkeit von der Nummer des zu betrachtenden Frames dieser Eigenschaft Werte zuweist. Ein Track kann auch als Funktion über den Nummern der Frames interpretiert werden.

Die Keys dieser Tracks müssen nicht jeweils in denselben Frames liegen, wie das folgende Beispiel zeigt:

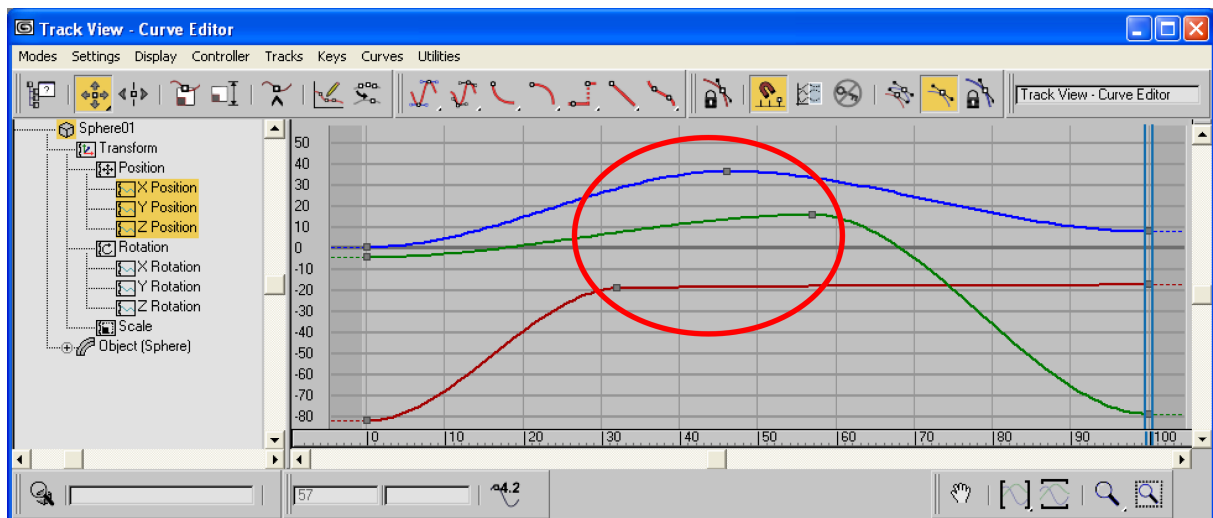


Abbildung 20: Darstellung von Animations-Tracks in 3D-Studio-MAX

Zwischen den Animations-Keys werden die Werte durch Interpolation bestimmt. Es können in 3D-Studio-MAX verschiedene Methoden eingestellt werden, mit denen diese Interpolation durchgeführt wird. Diese Interpolationsarten können für jeden Animations-Track einzeln festgelegt werden. Die Auswahlmöglichkeiten hängen dabei vom Typ der animierten Werte ab. Für die Datentypen „Float“ und „Boolean“ stehen z.B. die folgenden Interpolatoren zur Verfügung:

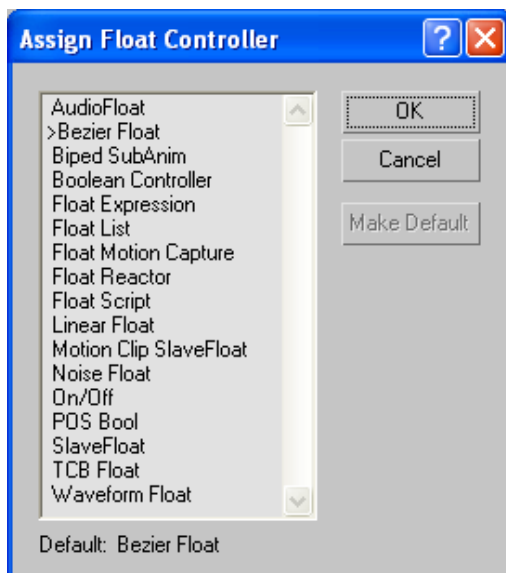


Abbildung 21: Interpolatoren (Controller) in 3D-Studio-MAX

Diese Interpolatoren werden bei 3D-Studio-MAX als „Controller“ bezeichnet, da sich die Auswahlmöglichkeiten nicht nur auf verschiedene Methoden der Interpolation zwischen Animations-Keys beschränken, sondern auch weitere Möglichkeiten zur Bestimmung der Werte eines Tracks zur Verfügung stehen. So bietet der Controllertyp „AudioFloat“ die Möglichkeit, einen zeitlichen Verlauf von Float-Werten aus einer Audiodatei abzuleiten.

Es kann in den meisten 3D-Animationsprogrammen ebenfalls festgelegt werden, welche Werte außerhalb des durch die Folge von Keys definierten Intervalls gültig sein sollen. Abbildung 22 zeigt die Auswahlmöglichkeiten in 3D-Studio-MAX:

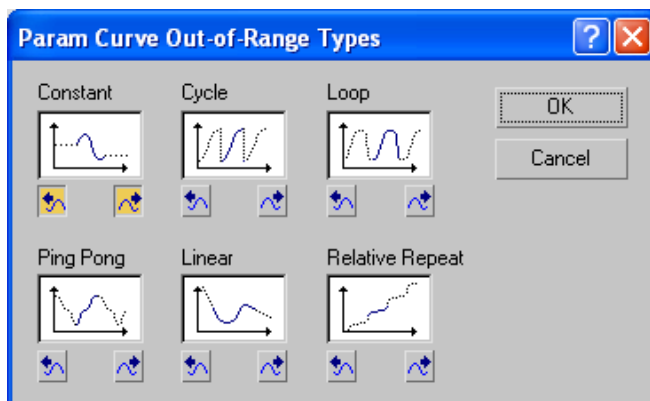


Abbildung 22: Mögliche Fortsetzungsarten für Animations-Tracks in 3D-Studio-MAX

Die durch Keys festgelegte Kurve kann außerhalb der Key-Folge auf verschiedene Arten systematisch fortgesetzt werden. Beispielsweise werden der Wert des ersten Keys der Folge für den links davon liegenden Bereich und der Wert des letzten Keys für den rechts davon liegenden Bereich konstant gehalten. Dies entspricht im obigen Bild der Fortsetzungsart „Constant“.

Auch wenn für eine Eigenschaft kein Key festgelegt wird, so ist sie in den meisten 3D-Animationsprogrammen wie 3D-Studio-MAX dennoch stets mit einem Wert belegt. Der Wert einer Eigenschaft kann also nicht undefiniert sein, sondern es wird mindestens ein vordefinierter Standardwert verwendet. Jede Eigenschaft hat in jedem Frame der Animation einen Wert, wurden für die Eigenschaft keine Keys festgelegt, so gilt im gesamten Animationsintervall für diese Eigenschaft konstant der gleiche Wert. Dieser konstante Wert kann auch verändert werden, ohne dabei einen Key zu erzeugen. Der neue Wert gilt dann wieder konstant für das gesamte Animationsintervall. Dies gilt auch für die im vorangehenden Kapitel 3.4 erwähnten nicht-animierbaren Eigenschaften. Da für sie keine Keys festgelegt werden können, bleiben sie im Verlauf des Animationsintervalls zwangsläufig konstant, wobei dieser konstante Wert durch den Benutzer jedoch geändert werden kann, so dass dann der neue Wert konstant für das gesamte Animationsintervall gilt.

3.6 Rendering

Als *Rendering* bezeichnet man die Erzeugung einer computergenerierten Darstellung eines Zustandes einer Szene in Form eines zumeist zweidimensionalen Bildes. Diese Darstellung erfolgt aus einer bestimmten Blickrichtung und -position. Dieser Begriff wird auch in [Foley93] näher erläutert.

In Echtzeitanwendungen geschieht dies zumeist hardwareunterstützt, für die Erzeugung eines Filmes werden hingegen meistens softwarebasiert photorealistische Bilder erzeugt, was im Allgemeinen mehr Rechenzeit benötigt als die hardwarebasierte Darstellung, aber zu höherwertigen Ergebnissen führt.

Liegt für die Szene eine Animation vor, also eine Folge von Zuständen der Szene, so kann man durch Rendering dieser Zustände einen Film erzeugen. Dabei entsteht aus jedem Zustand der Szene ein Frame des resultierenden Films.

Ein solcher Computeranimationsfilm besteht häufig aus einer großen Datenmenge, da er alle Einzelbilder der Animation in grafischer Form enthält. Von einem ausreichend großen Speichermedium mit entsprechender Datentransfargeschwindigkeit kann sie abgespielt werden, vorausgesetzt das abspielende System hat die nötigen Fähigkeiten zur Darstellung von Grafik.

Vorher wird die Animation eventuell in ein spezielles effizienter speicherbares und leichter abspielbares Filmformat umgewandelt. Diese Verfahren verringern in der Regel die für die Einzelbildfolge benötigte Speicherkapazität, wobei jedoch durch die angewendeten Kompressionsverfahren oft ein Qualitätsverlust an den Einzelbildern entsteht.

4 Anforderungen an MaxControl auf der Basis des aktuellen Stands der Technik

MaxControl wird in diesem Kapitel mit anderen Werkzeugen, Techniken und Produktionsprozessen verglichen. Aus diesem Vergleich ergeben sich Anforderungen, die MaxControl erfüllen soll, um Schwächen bisher existierender Werkzeuge und Techniken auszugleichen, ihre Stärken zu nutzen oder sich in erprobte Produktionsprozesse einfügen zu können.

4.1 Existierende und mögliche Techniken zur automatischen Erstellung von Animationen

Um eine teilweise oder sogar vollständige Automatisierung des Animationsvorganges zu ermöglichen, existieren bereits verschiedene Technologien, von denen einige wichtige in den folgenden Kapiteln 4.1.1 bis 4.1.4 vorgestellt werden. Dabei wird dargelegt, welche Schwächen solcher Ansätze MaxControl vermeiden soll und welche ihrer Fähigkeiten MaxControl nutzen kann. Überblicke über solche Techniken sind in [Foley93] und [Jack06] zu finden.

Es werden in den Kapiteln 4.1.5 bis 4.1.8 auch alternative mögliche Lösungsansätze diskutiert, mit denen die an MaxControl gestellten Forderungen eventuell erfüllt werden könnten.

Kapitel 4.1.9 fasst weitere Schwächen bisher existierender Lösungen zusammen, die MaxControl vermeiden soll.

4.1.1 Systeme zur Animation natürlicher Phänomene

Die Darstellung natürlicher Phänomene wie Regen, Schnee, das Verhalten von Wasser z.B. in Meeren oder von sichtbaren Gasen wie Wolken, Rauch oder Feuer ist auch in Computeranimationsfilmen häufig wünschenswert. Das oft sehr komplexe Verhalten dieser Phänomene ist kaum manuell animierbar. Daher beschäftigt sich ein großer Teil der Automatisierungsverfahren mit der Realisierung solcher Animationen durch verschiedenste Simulationstechniken.

Weil die meisten dieser Phänomene auf der Interaktion kleinster Partikel basieren, z.B. von Molekülen in Gas oder Wasser, lösen viele Systeme diese Animationsaufgabe auch über die im folgenden Kapitel 4.1.2 behandelten Partikelsysteme. Da jedoch spätestens bei der Simulation von Molekülinteraktionen bei Partikelansätzen zu viele Partikel individuell simuliert werden müssten, werden oft gitterbasierte Finite-Elemente-Lösungen verwendet oder hybride Lösungen aus beiden Ansätzen, die ebenfalls in Kapitel 4.1.2 erläutert werden. Gerade ältere Lösungen verwenden oft weitere spezialisierte Ansätze wie Fraktale oder Wellenfunktionen.

In [Foley93] und [Jack06] wird jeweils ein guter Überblick über die verschiedenen Techniken gegeben, hier seien zusätzlich zu den in Kapitel 4.1.2 behandelten Partikel- und Hybridlösungen noch andere wichtige genannt, z.B. rein gitterbasierte Finite-Elemente-Lösungen ohne Partikel.

Mit dem System aus [Perlin85] können über die Komposition nichtlinearer Funktionen Wolken, Feuer und Wasser direkt animiert und dargestellt werden. Wasser wird dabei durch die Veränderung der Oberflächen-Normalen einer Wasseroberfläche über spezielle Wellenfunktionen realisiert, die Oberfläche wird also nicht im 3D-Raum verformt sondern

nur die Reaktion der Oberfläche auf Licht wird wie bei Bump-Mapping-Verfahren verändert. Feuer und Wolken werden ähnlich durch die Bestimmung von Farbwerten über spezielle Turbulenz-Funktionen erzeugt. In [Fourn86] und [Peach86] wird zur Animation von Meereswellen die Wasseroberfläche wirklich dreidimensional durch Wellenfunktionen verformt, Partikelsysteme simulieren dabei die Gischt. In [Kass90] wird lediglich über Gleichungen, die für flache Gewässer gültig sind, ein Höhenfeld für die Wasseroberfläche erzeugt und animiert, so dass damit keine vollen 3D-Formen wie sich überstürzende Wellen erzeugt werden können. Ein Überblick über aktuellere Methoden zur Simulation von Meereswellen wird in [Tess02] gegeben. In [Gard84, Gard85] werden Wolken durch prozedural texturierte Ellipsoide dargestellt. Durch Variation der Parameter der prozeduralen Texturen können diese Wolken auch animiert werden. Nach [Jack06] stellt [Ebert90] eine frühe relativ einfache gitterbasierte Methode zur Animation und Darstellung von Gasvolumina dar, wohingegen [Sakas93] in einem gitterbasierten Ansatz Turbulenzen in einem auf Fraktalen basierenden Gasvolumen simulieren kann und so realistischere Ergebnisse liefert. In [Dob00] werden Wolken über Zellautomaten animiert.

Alle Lösungen zur Animation solcher Naturphänomene sind spezialisiert und schon durch ihre Aufgabenstellung in ihren Möglichkeiten beschränkt. Ohne gravierende Änderungen kann ein System zur Animation von Meereswellen z.B. kein Autorennen simulieren oder die Lichtquellen einer virtuellen Leuchtreklame ansteuern. Schon die Kombination verschiedener Lösungen aus dem Bereich der Naturphänomene in einem System ist problematisch und wird, wie schon in Kapitel 2.1 ab S. 29 erläutert, nur von wenigen Systemen unterstützt. Daher soll MaxControl keiner so hohen Spezialisierung auf ein bestimmtes Aufgabengebiet unterliegen.

4.1.2 Partikelsysteme

Am weitesten verbreitet sind für die automatische Erstellung von Animationen Partikelsysteme. Sie wurden in der Arbeit [Reeves83] entwickelt. Sie sind inzwischen Bestandteil fast jeder 3D-Animationssoftware. Nähere Informationen zu Partikelsystemen sind auch in [Foley93] zu finden.

Grundsätzlich werden Partikelsysteme verwendet, um viele zumeist einfache Objekte, so genannte Partikel, dynamisch zu erzeugen, die oft nur kurz in der Animation sichtbar bleiben, wie z.B. Schneeflocken oder Regentropfen.

Eine seltener genutzte Anwendung von Partikelsystemen ist die automatische Erstellung komplexer Geometrie, die in [Reeves85] entwickelt wurde. Dabei wurde ein Wald generiert, der durch Partikelsysteme erzeugte Pflanzen enthält, deren Einzelteile Partikel sind, die sich auch bewegen können (z.B. Grashalme im Wind).

Zu bewegten Partikeln gibt es meistens einen Partikelemitter, der die Partikel im Verlauf der Animation erzeugt. So kann ein Partikelemitter beispielsweise in jedem Einzelbild der Animation eine bestimmte Anzahl neuer Partikel an seiner aktuellen Position erzeugen, die sich dann in einer bestimmten Richtung vom Partikelemitter wegbewegen und nach einer festgelegten Zeit oder nach festgelegten Ereignissen wieder verschwinden.

Die Gestalt der Partikel ist zwar meistens einfacher Natur, wie Dreiecke oder Kugeln, sie können bei neueren Partikelsystemen allerdings auch fast beliebige Formen annehmen. Das zeitliche Verhalten der Partikel wird meistens durch die Einstellung von Parametern des Partikelsystems bestimmt. So werden Wetterphänomene wie Schnee, Regen oder Sandstürme oder auch andere visuelle Effekte wie aufgewirbelter Staub, Funken oder Flammen realisiert. Durch Parametereinstellungen können oft Eigenschaften wie die Startrichtung der Partikel,

ihre Startgeschwindigkeit oder die Amplitude und die Frequenz einer Tänzelsbewegung (z.B. für Schneeflocken oder Luftblasen) festgelegt werden. Zusätzlich können externe Kräfte wie Gravitation, Wind oder Turbulenzen definiert werden, um die Bewegung der Partikel zu beeinflussen.

Wie bereits in Kapitel 4.1.1 erwähnt, wurden in [Fourn86] und [Peach86] erstmals Partikelsysteme zur Simulation von Gischt bei Meereswellen eingesetzt. Viskose Flüssigkeiten werden in [Mill89] durch miteinander interagierende Partikel simuliert. In [Breen92] wird sogar das Verhalten von Stoff durch ein entsprechendes Partikelsystem realisiert, wie z.B. auch in [Bara98, Brid02, Choi02, Bara03, Cord02, Vill05, Wang02]. Ein Überblick über diese historische Entwicklung der Partikelsysteme gibt [Jack06].

Verschiedene Anwendungen von Partikelsystemen führen zu speziellen Problemen und entsprechenden Lösungsansätzen. So entwickelte [Bell05] eine partikelbasierte Methode zur Simulation granularer Materialien wie Sand, die aufgrund ihres speziellen Verhaltens z.B. bei Reibung nicht leicht zu simulieren sind. In [Feld03] werden Explosionen in einer Kombination aus Partikeln und Flüssigkeitsdynamik simuliert. Obwohl das Verhalten von Flüssigkeiten zurzeit eher mit gitterbasierten Finite-Elemente-Methoden simuliert wird, verwendet [Müller03] Partikel, um das Verhalten von Wasser in Echtzeit simulieren zu können. [Des99] verwendet bereits einen ähnlichen Ansatz, der über Partikel die Simulation stark deformierbarer Körper ermöglicht, von Flüssigkeiten bis hin zu Festkörpern. Die Simulation ist dabei zeitlich und räumlich adaptiv, um die Effizienz zu erhöhen. Wasser wird auch in [OBrien95] simuliert, wobei Partikel hier nur für Wassertropfen verwendet werden, die sich von der Wasseroberfläche gelöst haben. Das Schmelzen und Erhärten von Objekten wird in [Carls02] simuliert, wobei Partikel in dem Simulationssystem nur verwendet werden, um eine genauere Form der Oberfläche erhalten zu können, als es mit der zugrundeliegenden gitterbasierten Methode allein möglich wäre. Auch in [Clav05] werden viskoelastische Flüssigkeiten wie Schlamm oder Farbe und plastisch verformbare Körper über Partikel simuliert, wobei auch eine wechselseitige Interaktion mit Festkörpern möglich ist. [Prem03] simuliert Flüssigkeiten ebenfalls als Partikel und ermöglicht damit die Simulation des Mischens verschiedener Flüssigkeiten mit unterschiedlichen physikalischen Eigenschaften. Auch die unten genannte Arbeit [Losa06] ermöglicht das Mischen verschiedener Flüssigkeiten. Abgeleitet aus der in [Müller03] erläuterten Methode, Wasserdynamik über Partikel zu simulieren, werden in [Hadap01] auch Haare als fluides Kontinuum simuliert, wobei die einzelnen Haare als individuelle Einheiten erhalten bleiben. Auch [Vol04, Vol06] simuliert das Verhalten von Haaren mit Hilfe von Partikeln. Hier spannen die Partikel ein dreidimensionales Gitter auf und verformen es z.B. als Reaktion auf Kollisionen mit dem Kopf oder den Schultern. Das Gitter verformt wiederum entsprechend das Haarvolumen. [Angel05] verwendet ebenfalls Partikel, um das Verhalten von Rauch möglichst effizient zu simulieren, wobei auch eine manuelle Kontrolle der Simulation ermöglicht wird. Auch [Schpok03] ermöglicht die manuelle Kontrolle von Wolkensimulationen über Partikel, an deren manuell festgelegter oder simulierter Bewegung sich die Wolkensimulation orientiert. Ähnlich geht auch [Pigh04] vor. Hier kann die Simulation von Gasen, die dort grundsätzlich einen gitterbasierten Ansatz verwendet, durch die Manipulation der Bewegungspfade zusätzlich berechneter Partikel manuell kontrolliert werden. Die Simulation von Gasen mit Hilfe von Partikeln war nach [Jack06] bereits Thema der Arbeiten [Stam93, Stam95, Wejch91] und [Stam97]. [Stam93] simuliert das Verhalten von Gasen in Turbulenzen, während sich [Stam95] mehr mit der effizienten Darstellung der simulierten Gase beschäftigt. [Wejch91] simuliert das Verhalten von Objekten in bewegten Gasen oder Flüssigkeiten, wie z.B. von Blättern, die im Wind umherwehen. Diese Objekte werden aus Massepunkten zusammengesetzt, die durch Federn miteinander verbunden sind und bei diesem Simulationssystem die Rolle der Partikel übernehmen. Ähnlich wie die unten erwähnte Arbeit

[Sell05] verwendet auch [Stam97] zur Simulation von Gasen einen Partikelansatz zusammen mit einer gitterbasierten Finite-Elemente-Methode, in diesem Fall um eine Animation in zwei Phasen mit steigender Genauigkeit festlegen zu können. In [Ras03] wird der traditionelle gitterbasierte Ansatz vermieden, da er bei großen zu simulierenden Gasvolumina ineffizient wird. Stattdessen werden hauptsächlich Partikel simuliert, zusammen mit einigen feineren 2D-Gittern und einem groben 3D-Gitter, so dass ein vollständig dreidimensionales feines 3D-Gitter vermieden werden kann. Einige Arbeiten wie [Blinn82, Hab02, Clav05] beschäftigen sich mit dem Problem, aus mehreren Partikeln eine kontinuierliche Oberfläche abzuleiten, um so ein Objekt darstellen zu können, das sich aus mehreren Partikeln zusammensetzt, wie z.B. miteinander verbundene Tropfen eines Wasserstrahls oder Atome in einem Molekül, die sich Elektronen teilen. Während sich die meisten Partikelsysteme auf ein bestimmtes Problem festlegen, beherrschen einige partikelgestützte Systeme wie [Losa06, Melek05, Guend05, Losa06a, Sell05] auch die gleichzeitige Simulation verschiedener Phänomene, wobei [Melek05] Partikel nur am Rande einsetzt. [Losa06] simuliert neben dem Mischen verschiedener Flüssigkeiten und Gase auch chemische Vorgänge wie das Verbrennen von Flüssigkeiten, wobei auch die Flammen als Gase simuliert werden können. In [Melek05] wird eine Systematik entwickelt, möglichst viele verschiedene Phänomene wie Verbrennen, Zerschlagen, Wasser, frierendes Wasser und verformbare Körper zusammen simulieren zu können, indem Objekte je nach Problemstellung anders repräsentiert werden können. In [Guend05] wird die Interaktion zwischen Wasser, Stoff und Festkörpern in einem System simuliert. In [Losa06a] wird das Schmelzen und Verbrennen von Festkörpern zu Flüssigkeiten und Gasen simuliert. [Sell05] simuliert gleichzeitig Explosionen, Rauch und Wasser. Hier wird einen Partikelansatz zusammen mit einer gitterbasierten Finite-Elemente-Methode genutzt, um die jeweiligen Schwächen der beiden Methoden umgehen zu können. Dies hat auch [Foster01] zum Ziel, wobei sich diese Arbeit auf die Simulation von Wasser und anderen viskosen Flüssigkeiten wie Schlamm konzentriert. Eine Kombination aus gitterbasierter Methode und Partikelsystem nutzt auch [Green04], um in einem gitterbasiert simulierten Wasservolumen das Entstehen von Luftblasen zu berücksichtigen, die dann als Partikel simuliert werden. Flüssigkeiten wie Wasser und teilweise auch Gase werden in einer kombinierten gitter- und partikelbasierten Methode auch in [Enr02, Wang05, Guend05, Losa06, Losa06a, Foster96] simuliert. Dabei beschränkt sich [Wang05] auf die Bewegung von Wassertropfen auf festen Oberflächen. Ebenso verwendet [Yngve00] grundsätzlich einen gitterbasierten Ansatz, um Explosionen zu simulieren, und kombiniert dies mit Partikeln. Ein Teil dieser Partikel folgt der gitterbasierten Simulation, um den Feuerball der Explosion anhand der jeweiligen Position und Temperatur dieser Partikel darstellen zu können. Andere Partikel werden verwendet, um durch die Explosion aufgewirbelten Staub zu simulieren und darzustellen. Einige partikelgestützte Systeme sind auf die Animation von Flammen spezialisiert, wie z.B. [Lam02]. In [Kim04] wird das Wachstum von Eiskristallen simuliert, ebenfalls mit einer Kombination aus gitterbasierter Methode und simulierten Partikeln, die sich an einem Eiskristall anlagern können. [Fear00] simuliert das Anhäufen und Abrutschen von Schneemassen mit Hilfe von Partikeln. In [Chen05] wird der Weg spezieller Partikel, so genannter γ -tons, durch eine Szene verfolgt, um so verschiedene Verwitterungserscheinungen an 3D-Objekten auf der Basis dieser Partikel zu simulieren. Auch in [Cutler02] werden Partikel mit speziellen Verhaltensweisen simuliert, die bei Berührung 3D-Modelle in ihrem Aussehen verändern können, um ebenfalls das Verwittern oder Verschmutzen von Objekten zu simulieren. In einem ähnlichen Ansatz, der von Lichtquellen ausgehende „Lichtpartikel“ verfolgt, wird in [Haev04] das Wachstum von Pflanzen in Abhängigkeit des globalen Lichteinfalls auf die Pflanzen simuliert. [Kaji84] beschäftigt sich unter anderem mit dem Rendering vieler Partikel, was bei der oft hohen Anzahl an Partikeln ein beträchtliches Problem darstellen kann. [Lengy01] simuliert Partikel, um virtuelles Fell zu erzeugen. Das in [Kacic03] beschriebene System kann viele mechanische Phänomene simulieren, neben der

Mechanik von festen und verformbaren Körpern sowie von Stoff und Haaren auch das mechanische Verhalten von Partikeln, auf deren Simulation die Umsetzung der anderen Phänomene basiert, indem z.B. in Festkörpern die Partikel durch Federn verbundene Massepunkte repräsentieren. In [Yaeg86] werden Partikel dazu verwendet, eine Textur zweidimensional nach berechneten Strömungen zu verzerren, damit diese Textur als durch Wirbel bewegte Atmosphäre eines Planeten verwendet werden kann. So wurde die Atmosphäre des Planeten Jupiter für den Film "2010" (1984) animiert.

Der Komplexität und Anzahl der Parameter und der simulierbaren externen Kräfte sind durch die Architektur eines Partikelsystems Grenzen gesetzt, jedoch werden diese Grenzen wie oben erläutert ständig erweitert. Natürlich können Partikelsysteme in ihrem Verhalten beliebig komplex implementiert werden, so dass mit ihnen sogar die Animation vieler Figuren möglich ist, dies ist jedoch selten der Fall und es ist auch fraglich, ob solche Systeme dann noch als Partikelsysteme oder schon als Crowd-Systeme bezeichnet werden sollten. Beispiele für so komplexe Partikelsysteme sind [Kamp04], das dort als Partikelsystem bezeichnet wird, [Braun05], welches das Verhalten vieler Menschen in Notfallsituationen simuliert und die in [Hery04] erläuterte Methode, ein Partikelsystem durch Plugins und Skripte so zu erweitern, dass es als Crowd-Animationssystem verwendet werden kann. Diese Methode wurde für den Film "Star Wars Episode 1: The Phantom Menace" (1999) eingesetzt. Das System zur automatischen Animation von Vögeln aus [Reyn87] wird als eine Weiterentwicklung eines Partikelsystems bezeichnet, in dem die simulierten Vögel die Partikel sind. Das Problem, Partikelsysteme klar von komplexeren Systemen wie dem dort vorgestellten System oder beispielsweise Crowd-Systemen zur Animation vieler Menschen oder Tiere abzugrenzen, wird dort auch thematisiert. Das Verhalten der in [Reyn87] simulierten Objekte sei demnach um ein oder zwei Größenordnungen komplexer als die typischen Verhaltensweisen von Partikeln. Dadurch unterschieden sich die Systeme jedoch nur in gewissen Größen voneinander, nicht jedoch in ihrer Art. Zur klareren Unterscheidung zwischen Partikelsystemen, Schwarmsystemen und Crowd-Systemen mit autonomen Akteuren gibt [Jack06] eine Tabelle nach R. Parent an. Dort liegen die Hauptunterscheidungskriterien in der Anzahl der animierten Elemente und in deren individueller „Intelligenz“. Demzufolge animieren Partikelsysteme typischerweise zwischen 10^4 und 10^9 Objekte ohne Intelligenz, Schwarmsysteme 10^2 bis 10^5 Objekte mit begrenzter Intelligenz und Systeme mit autonomen Akteuren animieren demnach unter 10^2 Objekte mit relativ hoher Eigenintelligenz. Insbesondere die Anzahlen animierter Objekte sind von der ständig steigenden Leistungsfähigkeit der eingesetzten Rechner abhängig, so dass schon die Angabe zu Crowd-Systemen mit unter 100 Objekten als veraltet anzusehen ist. Denn entsprechende aktuelle Systeme, wie beispielsweise das für die „Herr der Ringe“-Trilogie (2001-2003) eingesetzte „Massive“, animieren eher mehrere Hundert Tausend Objekte (zu diesen Zahlen bezüglich Massive siehe [Aitken04]). Die Größenverhältnisse der angegebenen Zahlen untereinander sind jedoch weiterhin als Unterscheidungsmerkmal angemessen.

Von Ausnahmen wie den oben angegebenen abgesehen sind die meisten Partikelsysteme in ihrem Funktionsumfang auf physikalische Phänomene ausgerichtet. Ein komplexes autonomes Verhalten komplexer Objekte ist so nicht realisierbar. Partikelsysteme steuern ihre Partikel typischerweise zentral mit global auf alle Partikel wirkenden Algorithmen, die Partikel steuern sich nicht mit „eigenen Denkprozessen“ selbst, wie es in Crowd-Systemen wie Massive üblich ist (siehe dazu [Jack06]). Der überwiegende Teil der Standard-Partikelsysteme stößt daher an Grenzen, wenn die Partikel eigene Entscheidungen treffen sollen, wie z.B. das Auffinden effizienter Pfade durch eine komplexe Umgebung.

Solche Probleme würden sich gut durch programmiersprachliche Ansätze zur Definition des Verhaltens einzelner Objekte lösen lassen. Da die 3D-Animationsprogramme aber wie oben

erwähnt eher auf Künstler zugeschnitten sind und 3D-Animationskünstler nach der allgemeinen Einschätzung der Industrie die Erstellung eigener Programme vermeiden, wird grundsätzlich versucht, solche und vergleichbare Probleme möglichst ohne den Einsatz von Programmiersprachen allein in der grafischen Benutzeroberfläche des 3D-Animationsprogramms lösbar zu machen. Um die Flexibilität und Ausdrucksstärke der Partikelsysteme auch ohne den Einsatz von Programmiersprachen zu erhöhen, wird bei einigen neuen Partikelsystemen ein interessanter Lösungsansatz verwendet. Hier können die Verhaltensweisen verschiedener Partikel durch Operatoren festgelegt werden, die durch Symbole repräsentiert werden und auf einer graphischen Benutzeroberfläche miteinander verknüpft werden können. Beispiele für solche Partikelsysteme sind „Thinking Particles“¹² und das System „Particle Flow“ in 3D-Studio-MAX. So wird jedoch auch bei diesen Systemen kein „programmiersprachlicher“ Ansatz im klassischen Sinn verfolgt, sondern es wird z.B. für „Thinking Particles“ damit geworben, dass weiterhin keine Programmierkenntnisse erforderlich sind.

Partikelsysteme sind in der Regel auf das dynamische Erzeugen neuer Partikel ausgerichtet, so dass mit ihnen nur Partikel animiert werden können, die auch von dem Partikelsystem erzeugt wurden. Es nicht möglich, mit einem solchen Partikelsystem Objekte zu animieren, die vor der Aktivierung des Partikelsystems bereits in der Szene vorhanden waren und manuell an gewünschte Ausgangspositionen bewegt wurden. Eine genaue Kontrolle von Ausgangspositionen und Anzahl der Partikel ist so nicht ohne weiteres möglich.

Insgesamt reicht die Ausdrucksstärke der meisten Partikelsysteme also nicht aus, um in einer Szene komplexe Objekte mit beliebiger Struktur und mit komplexem Verhalten mechanisch korrekt automatisch zu animieren. Auch flexiblere Systeme wie „Thinking Particles“ nutzen weiterhin nicht die Vorteile objektorientierter Programmiersprachen. Dies sind jedoch Ziele, die MaxControl verfolgt, weshalb ein rein partikelorientierter Ansatz hier nicht ausreichend ist.

4.1.3 Simulation von Mechanik

Eine weitere Methode zur Animationsautomatisierung, die zunehmend von 3D-Animationsprogrammen unterstützt wird, stellen Simulationstools für physikalische Vorgänge dar, die vornehmlich auf die Simulation der Mechanik fester und verformbarer Körper beschränkt sind. Der große Vorteil einer auf Mechanik-Simulation basierenden Animation ist der große Realismus der entstehenden Bewegungen im Vergleich zu manuell definierten Bewegungen z.B. durch Keyframe-Techniken [Jack06].

Die 3D-Animationssoftware Maya besitzt eine solche Komponente, und bei 3D-Studio-MAX ist das Physik-Simulations-Plugin „Reactor“¹³ inzwischen zu einer mitgelieferten Standardkomponente geworden (zu beiden Animationswerkzeugen siehe Kap. 5.2). Diese Komponenten ermöglichen es, das mechanische Verhalten einer zumeist festen Anzahl geometrisch durchaus komplexer Objekte zu simulieren. Dabei werden z.B. Kräfte, Kollisionen und Federkräfte im zeitlichen Verlauf berechnet, um realistische Bewegungen zu erzeugen. Mit diesen Werkzeugen können fast immer auch Dreh- und Schiebegelenke mit Bewegungsbeschränkungen definiert werden. Für aus solchen Gelenken aufgebaute Figuren wie z.B. Menschen können auf diese Weise passive Verhaltensweisen wie Umfallen nach einer Kollision simuliert werden. Diese spezielle Art der Simulation für Figuren wird auch als „Ragdoll“-Simulation (siehe [Zord05]) bezeichnet.

¹² <http://www.cebas.com/products/products.asp?UD=10-7888-33-788&PID=15>, 07.04.2007

¹³ <http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=8108755#dynamics>, 10.04.2007

Viele der in den Kapiteln 4.1.1 und 4.1.2 behandelten Simulationssysteme, z.B. zur Simulation von Wasser oder festen Partikeln, fallen ebenfalls in den Bereich der Simulation von Mechanik. In diesem Kapitel soll mehr auf die Simulation größerer Körper mit komplexer Geometrie und evtl. mechanischen Besonderheiten wie Dreh- und Schiebegelenken eingegangen werden. Darunter fallen z.B. Festkörper, verformbare Körper und Stoff.

Methoden zur Simulation von Mechanik werden in [Jack06] erläutert. Eines der wichtigsten Probleme bei der Simulation von Mechanik ist die Erkennung von Kollisionen zwischen Objekten, die eine komplexe, eventuell konkave Oberfläche haben. Die Effizienz dieser Verfahren lässt sich z.B. erhöhen, indem man die Objekte mit einfacheren Objekten wie Quadern oder Kugeln umschließt, für die der Fall, dass je zwei dieser einhüllenden Objekte und damit auch die jeweils zugehörigen eingeschlossenen Objekte nicht miteinander kollidieren, leichter zu erkennen ist. Einen Überblick über solche Verfahren gibt laut [Jack06] die Arbeit [Lin98]. Die oben genannten Gelenke schränken die relativen Bewegungen von Objekten zueinander ein und werden daher als „Constraints“ bezeichnet. Es lassen sich laut [Jack06] genau sechs verschiedene Gelenktypen angeben, wie z.B. Kugel-, Scharnier- und Schiebegelenke (=„prismatische Gelenke“). Die Simulation von Mechanik kann auf verschiedene Weisen beeinflusst werden, damit nicht nur passiv ablaufende Vorgänge wie das Umfallen eines Holzstapels simuliert werden können, sondern auch aktive Verhaltensweisen wie z.B. das Verhalten eines mechanisch simulierten Fahrzeugs. Nach [Jack06] kann diese Kontrolle auf verschiedenen Ebenen stattfinden. So sind Methoden der „Inversen Dynamik“ nutzbar, welche die Kräfte bestimmen können, die ein Objekt erzeugen muss, um die gewünschte Bewegung auszuführen oder eine gewünschte Position zu erreichen. Die oben genannten Constraints stellen ebenfalls eine Möglichkeit der Kontrolle dar. Auf der obersten Ebene stehen autonome Agenten, die eigene Entscheidungen treffen und auf dieser Basis Kräfte erzeugen. Diese Methode ist eine zentrale Animationstechnik in MaxControl, weil mit ihr autonomes Verhalten und realistische Bewegungen kombiniert werden können.

Die Simulation von Stoff stellt wegen seines komplexen mechanischen Verhaltens ein besonderes Problem dar. Nach [Jack06] gibt [Ng96] einen Überblick über Methoden zur Simulation von Stoff, wie auch [Jack06] selbst.

Meistens sind bei Werkzeugen zur Simulation von Mechanik, die in professionelle Animationswerkzeuge integriert sind, die Möglichkeiten zur Steuerung der Simulation, zum Beispiel durch das gezielte Setzen neuer Kräfte im Verlauf der Simulation, sehr eingeschränkt. Es ist häufig nur eine Nutzung der oben genannten Gelenk-Constraints möglich. Bei „Reactor“, dem entsprechenden Plugin für 3D-Studio-MAX, können neben diesen Constraints nur konstante Anziehungs-, Wind- und Motorkräfte sowie zu Beginn der Simulation gültige Impulse gesetzt werden. Die Impulse für ein Objekt werden dabei nur indirekt abgeleitet aus den Veränderungen der Position und der Rotation des Objektes zwischen den letzten beiden Frames vor Beginn der Simulation. Im weiteren Verlauf kann die Simulation dann fast nur noch durch im Voraus animierte Objekte beeinflusst werden, deren vordefinierte Bewegungen durch automatisch berechnete Kräfte erzwungen werden und die auf die anderen physikalisch simulierten Objekte einwirken können. Ein Eingreifen in die Simulation ist auch mit MAXScript¹⁴ nur bedingt möglich. Ein Grund dafür ist, dass z.B. die Simulation nicht fehlerfrei Schritt für Schritt durch ein Skript gesteuert ausgeführt werden kann. Wäre dies möglich, könnte ein schrittweises Verändern von bisher nicht animierbaren physikalischen Parametern genutzt werden, wie z.B. von Federruhelängen, Massen und

¹⁴ Die Skriptsprache von 3D-Studio-MAX, durch die fast alle Befehle, die durch die Benutzeroberfläche manuell gegeben werden können, auch in einem programmiersprachlichen Ansatz möglich sind

Impulsen. Außerdem können insbesondere keine an Körpern wirkenden Kräfte durch MAXScript gesetzt oder gelesen werden. Damit ist es nicht möglich, das physikalische Verhalten von Objekten mit komplexem, nach Möglichkeit sogar intelligentem Verhalten zu koppeln. Solche integrierten Komponenten allein sind also nicht für die Simulation autonomer Verhaltensweisen mit mechanisch korrekten Bewegungen ausreichend, sondern es sind Erweiterungen im Sinne der oben genannten Kontrollmöglichkeiten wie z.B. autonome Agenten erforderlich. Eine solche Herangehensweise wird von anderen Automatisierungswerkzeugen genutzt, die Bewegungen auf der Basis von gesteuerter Mechaniksimulation erzeugen. Dazu gehören einige der im folgenden Kapitel 4.1.4 erläuterten Methoden.

Dennoch wird die Simulation von Mechanik insbesondere bei der Simulation einer hohen Anzahl autonom handelnder komplexer Objekte wie Figuren oder Fahrzeugen nur sehr eingeschränkt eingesetzt. Dazu müssten autonom handelnde Objekte innerhalb der Mechaniksimulation aktiv Kräfte setzen können, um sich mechanisch korrekt bewegen zu können.

Ein solches Vorgehen wäre beispielsweise für die korrekte Animation von Düsentriebwerken hilfreich, die bislang sehr oft physikalisch fehlerhaft animiert wurden. So fliegen in Science-Fiction-Filmen Raumjäger im Weltraum häufig mit aktivierten Haupttriebwerken, die eine konstante Schubkraft erzeugen, halten dabei aber eine konstante Geschwindigkeit, obwohl sie ständig beschleunigen müssten. Am Ziel angekommen müsste ihr Bremsweg bei gleich hoher Bremskraft dann noch einmal genau dem Weg entsprechen, den sie mit aktivierten Triebwerken vorher zurückgelegt haben, was auch nicht so dargestellt wird. Der Jäger-Killer, der in einer der Anfangsszenen von „*Terminator 3*“ (2003) über dem Wasser schwebt, müsste eigentlich wie ein Stein ins Wasser fallen, sobald er seine Düsen in horizontale Richtung stellt, was in dem Film aber ebenfalls nicht geschieht.

Bei einer Animation, die nicht auf Simulation von Mechanik basiert, können solche Fehler eingebaut werden, da man sich dabei an keine physikalischen Gesetze halten muss. Da sich ähnliche Fehler leider häufig finden, sollten auch die negativen didaktischen Auswirkungen auf das Physikverständnis des gerade bei solchen Filmen oft auch relativ jungen Publikums nicht außer Acht gelassen werden. Dies ist ein weiteres Argument dafür, dass eine Animationstechnik, die auf der Simulation von Mechanik basiert, ein erstrebenswertes Ziel ist. Eine solche Animationstechnik könnte ohne weiteres gar keine unrealistischen Bewegungen erzeugen, die einige Gesetze der Physik verletzen. Die Wichtigkeit einer physikalisch korrekten Simulation des mechanischen Verhaltens von Objekten in Computeranimationen zeigen Arbeiten wie [Reitsma03, Sullivan03], welche die Fähigkeit des Menschen untersuchen, physikalisch fehlerhaftes Verhalten von 3D-Objekten in Computeranimationen zu erkennen. In [Sullivan01] wird auf der Basis solcher Betrachtungen ein System zur Simulation des mechanischen Verhaltens fester Körper entwickelt, das die Genauigkeit von Kollisionsberechnungen so herabsetzt, dass es von einem Betrachter möglichst wenig bemerkt wird, um die Effizienz der Simulation zu erhöhen, was dem hier geforderten Realismus jedoch wieder entgegenwirkt.

Eine Kontrolle der Mechaniksimulation durch manuell kontrollierte Werte, Controller oder autonome Agenten wurde daher in MaxControl ermöglicht, damit nicht nur eine Simulation von Mechanik zur Verfügung steht, sondern sich die simulierten Objekte auch durch das aktive Setzen von Kräften auf dieser Basis realistisch bewegen können. So wird erreicht, dass die oben genannten Einschränkungen der in 3D-Animationsprogramme integrierten Komponenten zur Simulation von Mechanik bezüglich der aktiven Bewegungskontrolle in MaxControl nicht vorliegen.

4.1.4 Tools zur automatischen Animation vieler 3D-Figuren

Es wurde bereits eine Reihe an Werkzeugen und Methoden für das automatisierte Animieren von 3D-Figuren entwickelt, insbesondere für zweibeinige menschenähnliche Figuren, so genannte „Bipeds“, aber auch für anders geformte 3D-Objekte (z.B. Vögel). Solche Systeme werden auch als Crowd-Animationssysteme bezeichnet, wenn sie sehr viele Figuren animieren sollen. 3D-Studio-MAX enthält dazu z.B. das Plugin „Character Studio“. Solche Techniken werden in [Jack06] erläutert.

Damit die Anwender von 3D-Animationsprogrammen nicht selbst programmieren müssen, können bei einigen in 3D-Animationsprogramme integrierten Lösungsansätzen wie „Character Studio“ bestimmte Verhaltensmuster auf der Basis von Grundverhaltenstypen wie „Ausweichen“, „Suchen“, „Reaktion auf Kollision“ etc. durch Einstellung entsprechender Parameter definiert und miteinander kombiniert werden. Hier wird das Verhaltensrepertoire aber durch die vorgegebenen Grundverhaltenstypen eingeschränkt. Die Verhaltensmuster können oft auch durch Skripte definiert werden, was die Ausdruckskraft dieser Tools stark erhöht. Jedoch unterliegen auch die Skriptsprachen der 3D-Animationsprogramme wieder Einschränkungen, die unten in Kap. 4.1.6 näher erläutert werden. Andere Werkzeuge wie „Behavior“¹⁵ von Softimage bieten durch Skripte erweiterbare hierarchische endliche Automaten zur Definition der Verhaltensweisen an, wobei die Ausdrucksmöglichkeiten auch hier wieder durch das erzwungene Automatenkonzept und die Nachteile der Skriptsprache eingeschränkt sind. Zu hierarchischen endlichen Automaten siehe [Alur99]. Diese integrierten Lösungen bieten mit Ausnahme von „Behavior“ zumeist auch keine Simulation von Mechanik, wie sie in MaxControl zur Verfügung steht.

Ein erstes Modell, um Vogel- und Fischeschwärme automatisch zu animieren, wurde in [Reyn87] entwickelt. Nach [Funge99] entwickelt sich die Komplexität des Verhaltens automatisch animierter Objekte über mechanische Simulation bis hin zu kognitiven Prozessen. Dies impliziert, dass kognitiv gesteuerte Objekte als Basis nach den Gesetzen der Mechanik simulierte Bewegungen behalten sollten, was, wie bereits erläutert, eines der Hauptkonzepte von MaxControl ist und auch in [Tu99] so gehandhabt wurde. Für die Erzeugung autonom handelnder Akteure finden auch auf Evolution basierende Techniken Verwendung, z.B. in [Sims94]. Auch die im vorangehenden Kapitel 4.1.3 genannte Inverse Dynamik kann verwendet werden, um die Kräfte zu berechnen, die eine Figur ausüben muss, um unter einer Simulation von Mechanik eine gewünschte Endkonfiguration zu erreichen bzw. einen gewünschten Bewegungsablauf nachzubilden oder diesem möglichst nahe zu kommen. Ein solcher Ansatz wurde zuerst von [Witkin88] verfolgt und findet sich auch in [Ausl95] und [Fang03]. Da ein solches Verfahren sehr rechenaufwändig ist (siehe [Foley93]), wird es selten für die Animation einer sehr großen Anzahl an Figuren eingesetzt.

Ein kommerziell erhältliches Standalone-Tool, das ebenfalls Bewegungen von Figuren auf der Basis einer Simulation von Mechanik erzeugt, ist das Produkt „Endorphin“¹⁶ der Firma „Natural Motion“. Dabei werden natürliche Reaktionen einer Figur durch künstliche Intelligenz und Simulation von Mechanik erzeugt, die sich hauptsächlich auf reflexartige Bewegungen z.B. als Reaktion auf das Fallen oder als Versuch, das Gleichgewicht zu halten,

¹⁵ http://www.softimage.com/products/xsi/pricing_and_packaging/advanced/behavior/, 11.04.2007

¹⁶ <http://www.3sat.de/3sat.php?http://www.3sat.de/nano/cstuecke/52524/index.html>,
<http://www.naturalmotion.com/endorphin.htm>, <http://www.naturalmotion.com/>, 11.04.2007

beschränken. Diese neue Technologie wurde bereits im dritten Teil von „*Herr der Ringe*“ und auch in dem Film „*Troja*“ verwendet¹⁷.

Die meisten Crowd-Systeme animieren eine große Anzahl an Figuren, indem sie vordefinierte Bewegungszyklen, die z.B. durch Motion Capturing hergestellt worden sein können, zu neuen längeren Bewegungsabläufen zusammensetzen. Dabei können Bewegungsabläufe wie „mit der Hand winken“ und „vorwärts laufen“ auch gemischt werden, um z.B. eine Figur im Lauf winken zu lassen. Diese Vorgehensweise nutzt beispielsweise das in Kapitel 4.1.2 ab S. 53 erläuterte Crowd-System, das in „*Star Wars Episode 1: The Phantom Menace*“ eingesetzt wurde [Hery04].

Eine Reihe solcher Systeme nutzt dabei wie oben erwähnt hierarchische endliche Automaten. Mit diesen kann gut die Auswahl passender Bewegungszyklen abgebildet werden. Ist eine Figur im Zustand „Laufen“, kann sie mit einer der zur Verfügung stehenden Laufanimationen animiert werden. Die Unterzustände des Zustandes „Laufen“ könnten „Schnell laufen“ oder „Langsam laufen“ sein, so dass je nach Unterzustand entsprechende Laufanimationen gewählt werden könnten. Analog können im Zustand „Gegner angreifen“ verschiedene Kampfanimationen gewählt werden. Auf solchen Automaten basieren neben dem oben genannten „Behavior“ auch die Systeme aus [Guti05] und [Rudo05].

Eine Simulation von Mechanik wird dabei im Allgemeinen nur selten genutzt, um die auf der Basis von Bewegungszyklen animierten Figuren z.B. bei Stürzen anhand einer „Ragdoll“-Simulation realistisch umfallen zu lassen. Dies ist beispielsweise beim Werkzeug „Massive“ der Fall, dass für Massenszenen in der „*Herr der Ringe*“-Trilogie eingesetzt wurde [Aitken04, Reg04, Rah03, Rosenb03]. Nachträglich können dort mit „sekundärer“ Mechaniksimulation auch Haare und Stoff für die Figuren animiert werden. Die Figurensteuerung basiert bei „Massive“ auf einem Fuzzy-Logic-System, dessen Regeln mit einer grafischen Benutzeroberfläche durch einen Nutzer manipuliert werden können.

Eine engere Mischung aus vordefinierten Bewegungsabläufen und nach den Gesetzen der Mechanik simulierten Bewegungen findet sich in [Zord05]. Dort kann eine Figur zunächst einer vorliegenden Animation folgen, die z.B. durch Motion-Capture entstanden sein kann. Wird die Figur dann durch eine andere Figur umgeworfen, wird zwar das Umfallen durch eine Simulation von Mechanik animiert, jedoch so, dass die Figur beim Umfallen aktiv durch Kräfte reagiert und damit ein fließender Übergang zu einer passenden Folgeanimation möglich ist, die wieder auf Motion-Capture basieren kann. Im Gegensatz zu vergleichbaren Systemen wie „Massive“ bleibt eine umgeworfene Figur hier also nicht liegen, sondern kann nach dem Sturz weiter durch vordefinierte Bewegungsabläufe animiert werden.

Jedes der hier erläuterten Systeme ist auf die Animation von Figuren spezialisiert. Die Form der Figuren ist dabei meist auf zweibeinige Figuren („Bipeds“) festgelegt. Einige Werkzeuge erlauben auch die Animation anders geformter Figuren, z.B. von Insekten oder Vögeln. Jedoch sind solche Systeme immer für eine solche Aufgabenstellung spezialisiert und bieten entsprechende Lösungsansätze zur Definition von Verhaltensweisen wie hierarchische endliche Automaten an, die für andere Animationsprobleme nicht unbedingt gleichermaßen geeignet sind. So wie ein optimiertes System zur Animation von Wasser nicht ohne gravierende Änderungen an seiner Architektur eine große Anzahl an Bipeds animieren kann, kann ein Crowd-System kein Wasser animieren. Damit ist schon die Interaktion solcher Systeme schwierig. Eine große Anzahl von Figuren, die zuvor mit einem Crowd-System animiert wurde, kann zwar nachträglich simuliertes Wasser verdrängen und so Wellen

¹⁷ <http://www.wired.com/wired/archive/12.01/stuntbots.html>, 21.04.2007

erzeugen, jedoch können diese Figuren dann nicht mehr vom Wasser verlangsamt oder von Wellen umgeworfen werden. Statt sich auf ein spezielles Animationsproblem zu konzentrieren, wurde MaxControl daher als flexibles System konzipiert, das verschiedenste Animationsprobleme in einem Werkzeug lösen kann, so dass diese verschiedenen Lösungen in einem System miteinander interagieren können. Nur so können z.B. mit Fuzzy-Logic kontrollierte mechanisch simulierte Fahrzeuge Figuren ausweichen, die über hierarchische endliche Automaten gesteuert werden und ihrerseits gleichzeitig den Fahrzeugen ausweichen, was mit zwei getrennten Systemen nicht möglich wäre.

4.1.5 Plugins als mögliche Technik

Eine zur gewählten Java-Lösung alternative Möglichkeit zur Implementation von MaxControl bestünde darin, für die automatische Steuerung des Verhaltens der Szenen-Objekte ein Plugin¹⁸ für die verwendete 3D-Animationssoftware zu entwickeln. Dies ist meistens nur in einer maschinennahen Programmiersprache wie C++ möglich, bei der viel Wert auf Ausführungsgeschwindigkeit gelegt wird.

Solche Programmiersprachen sind oft jedoch sehr fehleranfällig. Ein klassisches Beispiel dafür ist die fehlende Grenzenüberprüfung beim Zugriff auf Arrays, die ein Grund für schwerwiegende Programm- oder Systemabstürze sein kann. Gerade bei zeitaufwändigen Simulationen ist es für kostenintensive Produktionen mit engem Zeitplan nicht akzeptabel, die Möglichkeit eines Simulationsabbruchs oder von Datenverlusten durch Programmabstürze einkalkulieren zu müssen. Dies ist ein Argument für die Verwendung von Java, da diese Programmiersprache besonderen Wert auf Laufsicherheit legt. Dort wird eine Arraygrenzenüberprüfung immer durchgeführt.

Das Entwickeln eines Plugins ist relativ aufwändig, und durch seine enge Integration in die 3D-Animationssoftware ist ein solches Plugin auch sehr anfällig gegenüber Änderungen an der Implementation der 3D-Animationssoftware z.B. bei einer neuen Version. Bei „3D-Studio-MAX“ musste z.B. bei jedem ganzzahligen Wechsel von Version 2 bis zu Version 6 jedes Plugin durch eine neue angepasste Version ersetzt werden. Durch die große Nähe zur internen Implementation des 3D-Animationsprogramms werden umfassende Änderungen an den Plugins nötig, sobald das Hauptprogramm an wichtigen Schnittstellen verändert wurde.

Die für MaxControl gewählte Lösung basiert auf Java und der Kommunikation mit dem verwendeten 3D-Animationsprogramm über dessen Skriptsprache. Die Skriptsprache ist weniger anfällig gegenüber Versionsänderungen am 3D-Animationsprogramm (siehe folgendes Kap. 4.1.6) und Java bietet die oben erläuterte erhöhte Laufsicherheit.

4.1.6 Skriptsprachen

Damit ein programmiersprachlicher Ansatz zur Lösung spezieller Probleme möglich ist, werden in 3D-Animationsprogramme wie 3D-Studio-MAX oder Maya oft Skriptsprachen wie MAXScript bzw. Mel integriert.

Mit diesen Sprachen können programmgesteuerte Änderungen an der bearbeiteten Szene vorgenommen werden. So können Animationen erzeugt werden, aber es können z.B. auch

¹⁸ Eine zusätzliche Softwarekomponente, die sich beim Start meistens automatisch in ein vorhandenes Programm integriert, sobald sie in ein bestimmtes Verzeichnis (häufig „Plugins“ genannt) innerhalb des Programmordners kopiert wurde. Ein Beispiel dafür wäre ein spezieller Bildfilter für ein Grafikprogramm. Plugins werden oft auch von Drittanbietern entwickelt.

Objekte in der Szene systematisch erstellt werden. Einige der oben genannten Werkzeuge zur automatisierten Animation von Objekten, die in professionelle 3D-Animationssysteme integriert sind, bieten zusätzlich auch die Möglichkeit an, die im 3D-Animationssystem verfügbare Skriptsprache zur Definition von Verhaltensweisen für Szenenobjekte zu verwenden. Dies gilt beispielsweise für „Character Studio“ in 3D-Studio-MAX.

Nutzt man die Skriptsprache des verwendeten 3D-Animationsprogrammes, um Befehle an dieses Programm zu senden, müssen diese Skripte bei Versionswechseln fast nie angepasst werden, da Ihre Befehle auf einem hohen Abstraktionsniveau ausgedrückt werden, das weitgehend implementationsunabhängig ist. Um z.B. ein Szenenobjekt mit dem Namen „Kugel 01“ an eine neue Position zu bringen, kann in 3D-Studio-MAX der folgende MAXScript-Befehl gegeben werden:

```
$'Kugel 01'.pos = [47.6361, 22.1211, 0]
```

Die Position wird einfach als eine Folge von 3 Zahlen angegeben, deren Genauigkeit (wie float oder double) nicht näher spezifiziert wird. In der Programmiersprache C++ würde ein entsprechender Methodenaufruf evtl. nur Werte des Typs „double“ akzeptieren. Auf das Objekt wird hier anhand seines Namens in Form der entsprechenden Zeichenkette zugegriffen. Ein genauer Typ des Objektes wird nicht angegeben. Eine solche Anweisung wird voraussichtlich noch bei vielen Versionen von 3D-Studio-MAX gültig bleiben. In einer Skriptsprache implementierte Programme müssen bei einem Versionswechsel des Basiswerkzeugs also weitaus seltener angepasst werden als Programme, die in einer systemnäheren und strenger typisierten Programmiersprache wie C++ implementiert wurden.

Skriptsprachen sind jedoch meistens sehr eingeschränkt und nur in ihrer Datenzugriffstruktur objektorientiert. Sie sind nicht so ausgereift und ausdrucksstark wie klassische Programmiersprachen, z.B. Java. Eine tatsächliche Objektorientierung wie die Möglichkeit zur Definition eigener Klassen findet man bei den Skriptsprachen meistens nicht. Genau dies wäre aber wünschenswert, weil die in dieser Arbeit gestellte Aufgabe sich sehr gut für objektorientierte Ansätze eignet, denn es soll das zeitliche Verhalten von vielen **Objekten** definiert werden, die sich meistens verschiedenen **Klassen** zuordnen lassen.

Statt sich vollständig auf die Skriptsprache des verwendeten 3D-Animationswerkzeugs zu stützen, nutzt MaxControl diese Skriptsprache daher lediglich, um mit dem 3D-Animationswerkzeug zu kommunizieren, während das Verhalten der 3D-Objekte in Java spezifiziert und simuliert wird.

4.1.7 Proprietäre spezielle Lösungen

Bei vielen Filmproduktionen steht ein großes Budget zur Verfügung, so dass dort eine weitere möglicher Herangehensweise verfolgt wird, die jedoch nur mit großem Arbeitsaufwand und damit auch nur mit erheblichem finanziellen Aufwand realisierbar ist.

Es werden je nach Problemstellung Speziallösungen entwickelt, die genau für das vorliegende Problem konzipiert werden. So ist das oben erwähnte „Massive“ entstanden, um Armeen zu animieren, die gegeneinander kämpfen. Solche Lösungen zu entwickeln, ohne ein genereller einsetzbares Automatisierungssystem wie MaxControl als Basis zur Verfügung zu haben, ist mit hohem Aufwand und Kosten verbunden, was für Filmproduktionen mit großem Budget aber durchaus tragbar ist.

Durch die hohe Spezialisierung sind auch solche Lösungen nicht flexibel für gänzlich andere Animationsprobleme einsetzbar. Wahrscheinlich auch beeinflusst durch den positiven Eindruck, denn die mit „Massive“ animierten Szenen aus der „Herr der Ringe“-Trilogie (10.

Dezember 2001, 5. Dezember 2002, 1. Dezember 2003) hinterließen, entwickelten viele Firmen ähnliche Werkzeuge, wie z.B. das oben erwähnte „Behaviour“ (S. 57). Damit waren große Schlachten zwischen Armeen für viele Filmproduktionen möglich und es folgte eine Reihe von Filmen, die solche Szenen thematisierten („*Die Mumie kehrt zurück*“ (mit 29. April 2001 allerdings vor „*Herr der Ringe*“ erschienen), „*Troja*“ (2004), „*300*“ (2006)). Sogar die eher für Weltraumschlachten bekannte Filmserie „*Star Wars*“ konzentrierte sich in den neueren Episoden 1-3 (19. Mai 1999, 16. Mai 2002, 15. Mai 2005) eher auf Kämpfe zwischen Armeen auf Planetenoberflächen. Statt also aufwändige Neuentwicklungen zu leisten, greifen viele Filmproduktionen auf erprobte Techniken zurück (z.B. zur Animation von Armeen), auch wenn sie die Lösungen dazu wie im Fall von „*Star Wars*“ selbst entwickeln (S. 53/58), statt völlig neue Animationsprobleme anzugehen und Lösungen dafür zu implementieren, weil dies einen noch größeren Aufwand darstellen würde. In dieser Hinsicht scheint die zur Verfügung stehende Technik teilweise die Filminhalte zu beeinflussen.

Eine der seltenen Ausnahme bildet z.B. der Film „*Independence Day*“ (1996), für dessen Luftschlachten ein eigenes Automatisierungswerkzeug entwickelt wurde. Ein solches Animationsproblem scheint in dieser Form weder in neueren noch in älteren Filmen so gelöst worden zu sein.

Insgesamt zeigt sich also, dass große Filmproduktionen durchaus in der Lage sind, problemspezifische Animationslösungen für den aktuell zu produzierenden Film zu entwickeln, wegen des hohen Aufwandes aber dennoch manchmal eher den Filminhalt an die verfügbaren Effektsysteme angleichen als umgekehrt.

Der hohe Aufwand einer problemspezifischen Lösung liegt sicher auch in einer fehlenden Basis, auf die bei der Lösung neuer Problemen aufgebaut werden könnte, so dass solche Lösungen meistens vollständige Neuentwicklungen sind. Mit dem generell einsetzbaren flexiblen Animationssystem MaxControl als Basiswerkzeug kann das Verhalten von 3D-Objekten dagegen auf vielfältige Weise spezifiziert und simuliert werden, ohne bei der Implementierung technische Probleme lösen zu müssen wie die Integration des Animationssystems in den bestehenden Produktionsprozess. So können neue Animationsaufgaben effizient umgesetzt werden, weil in MaxControl viele rein technische Probleme bereits gelöst sind und sich die Neuentwicklung ganz auf das Verhalten der 3D-Objekte konzentrieren kann. Hinzu kommt, dass dabei auf bereits für MaxControl entwickelte Verhaltensweisen und Objekttypen zurückgegriffen werden kann und so unter anderem auch alte Animationslösungen (z.B. Animation von Fußgängern) mit neu entwickelten Lösungen (z.B. ein Autorennen mit mechanisch simulierten Fahrzeugen) in einem System zusammenarbeiten können (Fußgänger weichen Fahrzeugen aus und umgekehrt).

4.1.8 Java als mögliche Technik

Die in den Kapiteln 4.1.5 und 4.1.6 genannten Faktoren sprechen für einen Ansatz, der Java als Basisentwicklungsumgebung verwendet und das verwendete 3D-Animationsprogramm über dessen Skriptsprache steuert.

MaxControl basiert mit Java ähnlich wie das oben erwähnte ASAS auf einer allgemein nutzbaren Programmiersprache, die hier für die Animation von 3D-Szenen genutzt wird. Ein solcher Lösungsansatz wird in [Foley93] als Einsatz von „General-Purpose Languages“ bezeichnet.

Würde man versuchen, eine Lösung allein auf der Basis von Java zu entwickeln, wäre die gewünschte enge Integration in eine bestehende 3D-Animationssoftware nicht gegeben.

Stattdessen müsste MaxControl auch die Aufgaben der Objektmodellierung und des Rendering übernehmen. Mit Hilfe von MAXScript wurde daher eine Integration von MaxControl in 3D-Studio-MAX erreicht, so dass die 3D-Szenen mit diesem Werkzeug erstellt, nachbearbeitet und durch Rendering in einen Film umgewandelt werden können. Die Implementation von MaxControl kann somit neben der Kommunikation mit 3D-Studio-MAX auf die eigentliche Aufgabe der automatisierten Animation einer bestehenden Szene beschränkt bleiben.

Mit Java steht eine sehr laufsichere und ausdrucksstarke objektorientierte Sprache zur Verfügung. Die Steuerung des verwendeten 3D-Animationsprogrammes mit dessen Skriptsprache ist sehr versionsunabhängig und erfordert nur einen relativ geringen Entwicklungsaufwand.

Auch eine Erweiterung des Funktionsumfangs der auf MAXScript basierenden Steuerungskomponente ist leicht möglich, da ein entsprechend verändertes Steuerungssystem nur die zusätzlich erforderlichen Arten von Skript-Befehlen dynamisch erzeugen können muss und so den Funktionsumfang von MaxControl erhöht. Wenn eine neue Version der Steuerungskomponente z.B. neben der Position eines Objektes nun auch dessen Materialeigenschaften verändern können soll, müssen entsprechende Skriptbefehle für den korrekten Zugriff auf das Material eines Objektes erzeugt werden können. Durch die relative Versionsunabhängigkeit der Skriptbefehle wird dies im Allgemeinen leicht möglich sein.

4.1.9 Zusammenfassende Betrachtung weiterer Nachteile bisher existierender Techniken

Bisher wurden Techniken wie die in den vorangehenden Kapiteln behandelten Lösungen nur selten zur Animation von Objekten wie Lichtern, Flugzeugen, Autos, Schiffen, Raumschiffen oder komplexen Anlagen wie einem ganzen Flughafen mit Flugbetrieb verwendet. Dies wird durch die hohe Spezialisierung dieser Werkzeuge verhindert, die sich bei komplexen Objekten zumeist auf Figurenanimation (Kap. 4.1.4) und bei einfachen Objekten meistens auf typische Partikelverhaltensweisen (Kap. 4.1.2) beschränken und nicht leicht für andere Aufgaben erweiterbar sind.

Bedingt durch die Konzentration auf die Animation der Position von Objekten wie Bewegungen von Armen und Beinen, beschränken sich die Werkzeuge zumeist auch auf Translations- und Rotationsbewegungen sowie Formveränderungen der Objekte. Nur wenige Systeme unterstützen eine automatisierte Veränderung von Eigenschaften wie Farbe, Oberflächenattributen oder Lichthelligkeit.

Es findet meistens auch nur eine Interaktion vieler gleichartiger Objekte statt, wie von Kriegerern in mehreren Armeen oder von Vögeln in einem Schwarm. Es werden jedoch noch keine komplexen Gesamtsysteme mit vielen verschiedenen Objekttypen und zugehörigen Verhaltensweisen simuliert, wie ein Flughafen mit Flugbetrieb, beispielsweise mit startenden und landenden Flugzeugen, die vom Tower Befehle erhalten, mit Landesignalen und Gepäcktransport.

Die verschiedenen Verhaltensweisen der Objekte werden bei solchen Werkzeugen oft mit Hilfe von grafischen Benutzeroberflächen erstellt, wobei meistens eher eine Verknüpfung grafischer Symbole als klassische textbasierte Programmierung im Vordergrund steht. Bei „Character Studio“ (Kap. 4.1.4) z.B. können Grundverhaltensmuster miteinander kombiniert werden, während das „Softimage“-Tool „Behaviour“ hauptsächlich grafisch erstellte hierarchische endliche Automaten verwendet [das muss alles in erstem Abschnitt über CS etc.

genauer ausgeführt werden]. „Thinking Particles“ bietet ähnlich wie „Particle Flow“ wiederum ein Repertoire von mit Symbolen dargestellten Operatoren an, aus denen das Verhalten eines Objektes grafisch zusammengesetzt werden kann (Kap. 4.1.2). Auch bei dem in Kapitel 4.1.4 erwähnten Programm „Massive“ werden die Regeln des Fuzzy-Logic-Systems, welches die Figuren kontrolliert, ebenfalls mit Hilfe einer grafischen Benutzeroberfläche durch einen Nutzer manipuliert. Die Ausdrucksstärke sowie der Aufwand für die Erstellung einer bestimmten Verhaltensweise hängen bei solchen Vorgehensweisen von den Ausdrucksmöglichkeiten der jeweils gegebenen endlichen Menge von Grundbausteinen ab. Eine Vereinfachung der Bedienung, z.B. durch grafische Methoden, bedeutet daher meistens auch immer eine Einschränkung der Ausdrucksmöglichkeiten. Gleiches bewirkt die in der Regel damit einhergehende Festlegung auf ein bestimmtes Lösungskonzept, wie endliche Automaten, neuronale Netze oder Fuzzy Methoden.

In der Regel wird keines solcher Konzepte für jede Animationsaufgabe gut geeignet sein, sondern die Wahl des Lösungskonzeptes sollte problemabhängig möglich sein, was MaxControl durch seinen programmiersprachlichen Ansatz in Verbindung mit seiner offenen Architektur ermöglicht. Durch den Einsatz von Java ist es auch flexibler als Systeme, welche die Definition von Verhaltensweisen nur über grafische Benutzeroberflächen zulassen. Einige der angesprochenen Werkzeuge lassen zwar zusätzlich die Verwendung von Skript-Code zu, jedoch unterliegt auch diese Form der Verhaltensdefinition Einschränkungen, wovon einige in Kapitel 4.1.6 behandelt wurden.

Die flexible Erweiterbarkeit von MaxControl vermeidet auch die weiteren in diesem Kapitel angesprochenen Nachteile anderer Systeme. So ist MaxControl weder auf die Animation einer bestimmten Gruppe von Objekten wie z.B. Bipedes beschränkt, noch auf die Animation bestimmter Parameter wie Translation und Rotation, sondern es können sowohl beliebige andere Arten von Objekten wie Lichtquellen oder Fahrzeuge animiert werden als auch beliebige ihrer Eigenschaften wie der Radius einer Kugel, die Lautstärke einer Tonquelle oder die Helligkeit einer Lichtquelle. Dadurch können auch nicht nur viele Objekte einer bestimmten Art miteinander interagieren, sondern alle mit MaxControl animierbaren Objektarten können in einer Simulation miteinander interagieren, z.B. Ampeln mit Fahrzeugen, Fußgängern und Vögeln.

4.2 Vergleich von MaxControl mit Techniken aus dem Bereich der Computerspiele

Die angestrebten Simulationen sollen mit Hilfe der Programmiersprache Java realisiert werden. Dabei werden die Zustandsänderungen jedes 3D-Objektes iterativ durch Java-Code berechnet, der dem jeweiligen Objekttyp zugeordnet ist, wobei die Simulation von Mechanik dabei durch eine weitere Softwarekomponente durchgeführt wird, auf die unter anderem in Kapitel 5.6 näher eingegangen wird.

Diese Methode der Animation von 3D-Szenen gleicht dem am Anfang von Kapitel 1.2 erwähnten Echtzeitansatz, wie er häufig in Spielen Verwendung findet. Die Vorteile dieses Ansatzes gegenüber dem oft auf bekannte Animationsprobleme spezialisierten Funktionsumfang eines professionellen 3D-Animationsprogramms sind oft so gravierend, dass viele Spielehersteller inzwischen auch für die Darstellung von nicht beeinflussbaren Sequenzen¹⁹ ihrer Spiele die jeweilige Game-Engine²⁰ verwenden, um die Sequenz in

¹⁹ Dies sind bei einem Spiel Sequenzen, die einen nicht beeinflussbaren Teil des Spiels darstellen, der z.B. bei einem Abenteuerspiel (=“Adventure“) die zum Spiel gehörige Geschichte weitererzählt, sofern dieser Teil der Geschichte durch das Verhalten des Spielers nicht beeinflusst werden kann. Solche Sequenzen können relativ

Echtzeit darzustellen²¹, statt einen mit einem herkömmlichen 3D-Animationsprogramm erstellten Film abzuspielen.

Wird dieser Ansatz nicht verfolgt, erreichen die nicht-interaktiven Sequenzen eines Spieles in der Komplexität des Verhaltens der einzelnen Szenenobjekte oft nicht die entsprechende Komplexität der interaktiven Spielsequenzen und sind ihnen nur noch in der optischen Qualität der Darstellung überlegen²².

Da das dynamische Verhalten der Objekte für die normalen interaktiven Spielsequenzen bereits in Programmform implementiert wurde, ist es nun einfach, diese Objekte mit ihren Verhaltensweisen auch in den Zwischensequenzen wieder zu verwenden, wenn auch hier die Game-Engine verwendet wird. Zu solchen Verhaltensweisen können intelligente selbständige Bewegungen von Objekten oder auch programmierte Spezialeffekte wie Waffenmündungsfeuer, Blinklichter, Düsenstrahlen, Funkensprühen durch Reibung oder durch abprallende Geschosse sowie Explosionen mit davonfliegenden physikalisch simulierten Partikeln gehören. Diese Verhaltensweisen können ursprünglich für das Spiel entwickelt worden sein und eignen sich dann ebenso gut für die nicht interaktiven Zwischensequenzen. Würde man dieselben Verhaltensweisen mit herkömmlichen 3D-Animationstechniken wie Partikelsystemen nachahmen wollen, würden oft komplexe oder sogar kaum lösbare Probleme auftreten und damit würde ein unnötiger Aufwand entstehen.

Ebenso bieten viele Spiele heute die Möglichkeit, eine interaktiv gespielte Sequenz als filmartige Wiederholung in Echtzeit darstellen zu lassen²³. Ansprechende Kamerafahrten werden dabei ebenfalls in Echtzeit erstellt. Solche Wiederholungen sind in ihrem Realismus oft schon sehr beeindruckend, durch die Echtzeitdarstellung fehlt es ihnen lediglich an der grafischen Qualität und Komplexität professioneller Computeranimationsfilme, die nicht in Echtzeit berechnet werden. Da im Rahmen dieser Arbeit keine Echtzeitdarstellung angestrebt wird, entfallen diese Einschränkungen bei dem hier verfolgten Ansatz jedoch.

Ein weiterer Vorteil der Spieltechnologie ist die Erzeugung von passenden Toneffekten in Echtzeit. Durch die Kenntnis von Geschwindigkeit, Position und Verhalten der Objekte können Toneffekte automatisch zum richtigen Zeitpunkt abgespielt werden und auch korrekte

lang sein (im Minutenbereich) und sind oft aufwendig produziert, um die Atmosphäre des Spiels nachhaltig zu unterstützen. Sie werden auch als „Cut Scenes“ bezeichnet.

²⁰ Eine „Game-Engine“ ist der Teil eines Spielprogramms, der die Spielmechanik kontrolliert, die Benutzereingaben verarbeitet und die audiovisuelle Darstellung des Spiels übernimmt.

²¹ z.B. „Zone of the Enders – The 2nd Runner“: <http://ps2.ign.com/objects/481/481744.html>, 22.04.2007

(hier wurden zusätzlich einige Zeichentricksequenzen eingefügt)

oder „Xenosaga – Episode 1 – Der Wille zur Macht“: <http://ps2.ign.com/objects/016/016268.html>, 22.04.2007

(hier wurde allerdings für die Zwischensequenzen kein Echtzeitansatz beibehalten, die von der evtl. nicht mehr in Echtzeit operierenden Game-Engine erzeugten Einzelbilder wurden scheinbar gespeichert, evtl. nachbearbeitet und werden im Spiel als vorausberechnete Filme vom Speichermedium abgespielt)

oder „Metal Wolf Chaos“: <http://xbox.ign.com/objects/683/683311.html>, 22.04.2007

²² dieser Effekt ist z.B. zu beobachten in

„Homeworld 2“: <http://pc.ign.com/objects/016/016221.html>, 22/04/2007

oder in „SILPHEED – THE LOST PLANET“: <http://ps2.ign.com/objects/014/014374.html>, 22.04.2007

oder in „Omega Boost“:

<http://psx.ign.com/objects/011/011434.html>, 22.04.2007,

<http://www.gamespot.com/ps/action/omegaboost/index.html>, 22.04.2007

²³ z.B. „Gran Turismo 4“: <http://ps2.ign.com/objects/489/489327.html>, 22.04.2007

oder „Ace Combat Zero: The Belkan War“: <http://ps2.ign.com/objects/771/771978.html>, 22.04.2007

Surround-Sound-Signale oder Effekte wie der Dopplereffekt erzeugt werden. Dies ist auch mit MaxControl möglich.

Natürlich müssen solche algorithmisch definierten Verhaltensweisen zuerst implementiert worden sein, wenn man nicht von einem bereits entwickelten Spiel oder einer bereits implementierten Simulation ausgehen kann. Dies ist sicher ein Grund dafür, dass solche Ansätze bisher bei reinen Computeranimationsfilmen selten verfolgt wurden, da sie hauptsächlich von 3D-Animationskünstlern und nicht von Technikern erstellt werden. In [Giesen00e] wird auch berichtet, dass man in der Industrie stets bestrebt ist, 3D-Animationsprogramme so zu entwickeln, dass sie von Künstlern ohne jegliche Unterstützung durch Computerspezialisten verwendet werden können. Die Tatsache, dass jedoch fast alle 3D-Animationsprogramme über eine Skriptsprache verfügen, belegt die Einsicht, dass viele Probleme ganz ohne Programmierung kaum lösbar sind. Die Programmierarbeit sollte auch nicht zwingend von 3D-Künstlern durchgeführt werden, sondern von darauf spezialisierten Fachleuten. Diese Art von enger Zusammenarbeit zwischen Künstler und Programmierer wird schon seit der frühesten Geschichte der Computer- und Videospiele erfolgreich eingesetzt, weshalb dies auch bei reinen Computeranimationsfilmen verstärkt und vorbehaltlos genutzt werden sollte. Von einer solchen Arbeitsteilung, die auch für den Einsatz von MaxControl vorgeschlagen wird, berichtet auch [Giesen00u] in Bezug auf den Film „*Starship Troopers*“ (1997), bei dessen Produktion Programmierer auf das Filmprojekt zugeschnittene Software entwickelten, die dann von Künstlern angewendet wurde.

4.3 Integration von MaxControl in industriell übliche Produktionsprozesse

MaxControl kann viele Aufgaben übernehmen, die mit der Animation verbunden sind, inklusive der Simulation von Mechanik und der automatischen Erzeugung von Tonspuren. In professionellen Produktionen werden 3D-Computeranimationsfilme jedoch selten in einem Schritt und mit nur einem Werkzeug hergestellt. Meistens werden mehrere verschiedene Werkzeuge für verschiedene Aufgaben eingesetzt, z.B. für Filmschnitt, Nachvertonung, Animation, Modellierung, Rendering oder zum Zusammenfügen mehrerer Elemente.

Schon das Rendering einer Animation wird häufig nicht in einem Schritt durchgeführt, sondern aus verschiedenen Gründen, die auch zur Verfügung stehende Speicher- und Rechenkapazitäten involvieren, werden oft mehrere Bildebenen, so genannte „Layer“, einzeln erzeugt, die erst in einem späteren Arbeitsschritt, dem so genannten *Compositing*, miteinander kombiniert werden. Compositing wird in [Jack06] näher erläutert. Nach [Watt01] wurden Compositing-Techniken für 3D-Computeranimationen mit [Por84] und [Duff85] eingeführt. Arbeiten, die Compositing-Techniken für spezielle Probleme entwickeln und auf die an anderen Stellen in diesem Text näher eingegangen wird, sind [Matus02, Chuang02, Chuang03] (siehe jeweils S. 16/71, S. 16 bzw. S. 66). Nach [Jack06] wurden bei der Produktion des Computeranimationsfilms „*Final Fantasy – Die Mächte in Dir*“ (2001) bis zu sechs Layer übereinandergelegt. Bei einem solchen Vorgehen entstehen jedoch auch Probleme, die zu beachten sind. Wird z.B. zuerst ein Layer X mit einer Wasseroberfläche auf der Basis einer 3D-Szene A erzeugt, und dann ein weiterer Layer Y mit einem Boot auf der Basis einer anderen 3D-Szene B erzeugt, so ergeben sich zunächst Vorteile beim Rendering. Denn bei beiden Vorgängen müssen nur vereinfachte Szenen behandelt werden, die vom Rendering-Verfahren effizienter verarbeitet werden können und auch jeweils weniger Arbeitsspeicher erfordern als das Rendering einer kombinierten 3D-Szene in einem Schritt. Das Compositing beider Layer kann auch 3D-Informationen berücksichtigen, wenn zu jedem Bildpunkt der beiden Layer Tiefeninformationen für den 3D-Raum vorliegen, die angeben, wie weit jeder Bildpunkt von der virtuellen Kamera entfernt ist [Duff85]. Diese

Tiefeninformationen für jedes Pixel eines Bildes werden auch als Z-Puffer bezeichnet [Steph03]. Jedoch kann sich das Boot nicht in der Wasseroberfläche spiegeln, wenn es beim Rendering der Wasseroberfläche nicht in der entsprechenden Szene A vorhanden war. Der Rumpf des Bootes aus Layer Y kann beim Compositing bei jedem Bildpunkt des Ergebnisbildes entweder vor dem Layer X mit der Wasseroberfläche sichtbar sein oder von Layer X verdeckt werden, je nach den Tiefeninformationen beider Layer an diesem Bildpunkt. Unter Berücksichtigung eines eventuell vorhandenen Alpha-Kanals kann das Verdecken eines Layers durch den anderen abgeschwächt werden, so dass der verdeckte Layer an der entsprechenden Stelle etwas durch den anderen Layer hindurchscheint. Der Alpha-Kanal eines Bildes gibt im Allgemeinen für jedes Pixel die Transparenz eines Layers für Compositing an (siehe [Por84]). Der Teil des Bootes, der sich unter der Wasseroberfläche befindet, könnte so zwar durch das Wasser hindurchscheinen, jedoch könnte er nachträglich nicht mehr durch das Wasser verzerrt erscheinen, wenn das Boot nicht schon in der Szene A, auf deren Basis der Layer X mit der Wasseroberfläche durch Rendering erstellt wurde, als 3D-Objekt vorhanden war. Eine solche Arbeitsweise mit mehreren Layern hat also Vor- und Nachteile. In [Chuang03] ist es immerhin möglich, dass Objekte aus einem Layer A korrekt Schatten auf die Szene in einem dahinter liegenden Layer B werfen, sofern diese Szene aus Layer B, die auch eine Live-Action Aufnahme sein kann, vorher systematisch auf ihr Schattenwurfverhalten hin analysiert wurde. Nach [Steph03] haben Z-Puffer-Verfahren zusätzlich Schwächen bei der Behandlung von Transparenzen, da ein Objekt hinter einem transparenten Objekt sichtbar sein kann, aber pixelweise nur Tiefeninformationen für eines der beiden Objekte gespeichert werden können. Das Compositing von Layern mit transparenten Objekten muss daher manuell gut geplant werden. In [Por84] wird das Compositing für den Effekt der "Genesis-Welle" aus "*Star Trek II – Der Zorn des Kahn*" (1982) erläutert. Dabei wurden transparente Partikel gerendert, die vor und hinter einem separat gerenderten Planeten erscheinen sollen. Dazu wurden die Sterne in einen ersten Layer A gerendert, die hinter dem Rand des Planeten liegenden Partikel wurden in einen Layer B gerendert, der Planet in einen Layer C und die vor dem Rand des Planeten liegenden Partikel in einen Layer D. Zunächst wurden die nicht transparenten Teile von Layer C aus Layer B ausgeschnitten, so dass aus Layer B nur die Partikel übrig bleiben, die sich zwar von ihrer Tiefe im Raum her hinter dem Rand des Planeten befinden, aber nicht vom Planeten verdeckt werden, da sie sich weit genug von der Planetenoberfläche entfernt haben. Über das so entstandene Bild wird nun transparent der Layer D mit den vor dem Rand des Planeten liegenden Partikeln gelegt. So entsteht ein Bild, das nur aus den im Endergebnis sichtbaren Partikeln besteht. Diese neue Ebene wurde dann unter Berücksichtigung ihres Alpha-Kanals über den Planeten aus Layer C gelegt, und das Ergebnis wird analog über den Layer A mit dem Sternenhintergrund gelegt. Durch die Transparenz der Partikel im Vordergrund aus Layer D können die einzeln gerenderten Layer nicht in beliebiger Reihenfolge miteinander kombiniert werden. Ein Z-Puffer-Verfahren kann allein nicht entscheiden, ob ein Partikel aus Layer D gerendert werden soll oder ob vorher andere Layer in das Gesamtergebnis übernommen werden sollten, die Objekte enthalten, die hinter diesen transparenten Partikeln durchscheinen und von einem Z-Puffer-Verfahren nicht mehr gerendert würden, wenn sie erst nach Layer D in das Gesamtbild übernommen werden. Abhilfe könnte nur ein erweitertes Z-Puffer-Verfahren leisten, dass für jedes Pixel des Ergebnisbildes die Kombinationsreihenfolge der einzelnen Layer anhand ihrer Tiefeninformationen neu bestimmt, indem die am tiefsten im Raum liegenden Pixel zuerst in das Ergebnisbild übernommen werden. Doch auch ein solches Verfahren könnte nicht vollkommen exakt sein, weil z. B. Layer D mit den vor dem Planetenrand liegenden Partikeln in jedem einzelnen Pixel mehrere hintereinander liegende transparente Partikel zeigen kann, die jedoch unterschiedlich tief im Raum liegen. Da pro Pixel nur eine Tiefeninformation im Z-Puffer gespeichert werden kann, wäre für das Verfahren nicht erkennbar, ob Pixel eines anderen Layers, z.B. der Planetenoberfläche aus

Layer C, vor einigen Partikeln und hinter anderen Partikeln desselben Pixels des Layers D erscheinen sollten. Zudem sind diese Partikel aus einem Pixel des Layers D nicht mehr voneinander zu trennen, da nur noch ihre Gesamtfarbinformation in jedem Pixel gespeichert ist. Compositing-Verfahren müssen also gut durchdacht eingesetzt werden und lassen sich nur bedingt automatisieren.

MaxControl schließt solche Vorgehensweisen nicht aus und kann in solche relativ komplexen Produktionsverfahren eingebunden werden, die z.B. verschiedene Tools zur Produktion und Kombination einzelner Bild- und Tonelemente verwenden und auch beim Rendering mit mehreren Layern arbeiten.

Die von MaxControl erzeugten Tonspuren können weiterbearbeitet werden und mit weiteren Tonspuren kombiniert werden. Außerdem muss die Möglichkeit, Tonspuren mit MaxControl zu erzeugen, nicht zwingend genutzt werden, stattdessen können auch herkömmliche Techniken zur Nachvertonung von 3D-Computeranimationsfilmen verwendet werden, die mit Hilfe von MaxControl produziert wurden.

Grundsätzlich soll MaxControl die zur Verfügung stehenden Produktionsmöglichkeiten nur erweitern und diese niemals einschränken. Beispielsweise können manuell animierte Objekte zusammen mit durch MaxControl animierten Objekten in einer Szene existieren. Mit MaxControl animierte Objekte können außerdem nachträglich wie jedes andere Objekt in der Szene manuell animiert werden, so dass die durch MaxControl erstellten Animationen nicht endgültig sein müssen. Ebenso kann MaxControl mit anderen automatischen Animationstechniken kombiniert werden. So könnten durch Crowd-Systeme animierte Figuren in einer anschließenden MaxControl-Simulation als manuell kontrollierte 3D-Objekte berücksichtigt werden, indem z.B. durch MaxControl animierte Fahrzeuge diesen Figuren ausweichen. Ebenso könnten, wie bereits in Kapitel 2.1 erwähnt, nachträglich Partikelsysteme auf die durch MaxControl animierten 3D-Objekte reagieren, indem beispielsweise mit einem Partikelsystem animierte Regentropfen von der Karosserie der Fahrzeuge abprallen, die zuvor durch MaxControl animiert wurden. Zusätzlich kann MaxControl während der Simulation die Partikelsysteme und die inverse Kinematik des als Basiswerkzeug verwendeten 3D-Animationsprogramms nutzen.

Ebenso kann MaxControl für die Animation von 3D-Objekten einzelner Layer verwendet werden, die später durch Compositing mit anderen Layern kombiniert werden sollen. Sollen jedoch mehrere Layer nachträglich kombiniert werden, deren 3D-Objekte durch MaxControl animiert wurden, ist eine Interaktion von 3D-Objekten aus verschiedenen Layern mit MaxControl nur möglich, wenn diese Objekte zum Zeitpunkt der Simulation zusammen in einer Szene existiert haben. Für das spätere Erstellen der Layer können dagegen vor dem Rendering eines bestimmten Layers alle Objekte aus der Szene entfernt werden, die für diesen Layer optisch nicht relevant sind, auch wenn diese Objekte vorher durch MaxControl zusammen mit den 3D-Objekten animiert wurden, die in dem Layer bleiben sollen. Es entstehen hier also ähnliche Probleme beim Kombinieren verschiedener Layer wie bei der oben beschriebenen optischen Interaktion von 3D-Objekten.

MaxControl kann einerseits zusammen mit dem verwendeten 3D-Animationsprogramm für die gesamte Produktion eines 3D-Computeranimationsfilms verwendet werden, andererseits kann es auch in komplexere Produktionsmethoden integriert werden, die viele Werkzeuge und komplexes Compositing einsetzen, wie es bei professionellen Produktionen allgemein üblich ist.

5 Überblick über die in MaxControl verwendeten Werkzeuge

In diesem Kapitel wird näher auf die verschiedenen in MaxControl verwendeten Softwarekomponenten eingegangen. Es werden die Aufgaben dieser Komponenten beschrieben und es wird erläutert, aus welchen Gründen diese Komponenten gewählt wurden.

5.1 Betriebssystem Microsoft Windows XP

Microsoft Windows XP²⁴ wurde als Betriebssystem für die technische Ausarbeitung der vorliegenden Arbeit gewählt. Der Grund für diese Wahl ist unter anderem, dass alle weiteren hier verwendeten Softwarewerkzeuge von diesem Betriebssystem unterstützt werden, teilweise auch ausschließlich.

5.2 3D-Animationswerkzeug 3D-Studio-MAX

Als Basiswerkzeug für die Erstellung von 3D-Animationen wurde 3D-Studio-MAX gewählt. Es gibt viele verschiedene 3D-Animationssysteme auf dem Markt, in jeder Preis- und Leistungsklasse. Im professionellen Bereich sind insbesondere Cinema-4D²⁵, Lightwave²⁶, 3D-Studio-MAX²⁷, Maya²⁸ und „Softimage|XSI“²⁹ zu nennen. Einige Informationen zu Maya und Softimage werden auch in [Giesen00] genannt. Lightwave siedelt sich in der unteren Profiklasse an und wird wohl auch wegen des relativ günstigen Preises oft in Fernseh-Serienproduktionen eingesetzt. 3D-Studio-MAX und Maya konkurrieren um die Spitzenposition, wobei Maya einen leichten Vorsprung hat. Beide Produkte werden sehr häufig in Kinoproduktionen verwendet. Softimage ist ein weiteres Tool, das ebenfalls zunehmend bei Kinoproduktionen eingesetzt wird.

Dass die Wahl schließlich auf 3D-Studio-MAX fiel, ist nicht zuletzt auf persönliche Präferenz zurückzuführen. Prinzipiell spricht nichts gegen den alternativen Einsatz der Tools Maya und Softimage, die 3D-Studio-MAX in einigen Bereichen durchaus überlegen sind. Jedoch ist für die hier vorgestellte Entwicklung eine eingehende Kenntnis der Prinzipien, Funktionen und Funktionsweisen des Basiswerkzeugs nötig, was zur Wahl von 3D-Studio-MAX führte. Die im Rahmen dieser Arbeit entwickelte Programmarchitektur wurde jedoch allgemein genug gehalten, um sie ggf. auch um eine Unterstützung anderer Tools zu erweitern zu können. Diese Möglichkeit wird auch für die Zukunft nicht ausgeschlossen.

3D-Studio-MAX wird bei MaxControl zur Erstellung, zum Laden und Speichern der 3D-Szenen verwendet. Alle für die Simulation relevanten Daten werden in der zur aktuellen 3D-Szene gehörigen Datei gespeichert. Bedingt durch die Systemarchitektur von 3D-Studio-MAX werden lediglich einige datenintensive Informationen in externen Dateien gehalten. Dies sind Texturen, Tondateien und optional auch geometrische Modelle von Objekten. Abgesehen von den in Java spezifizierten Verhaltensdefinitionen für Objekttypen hält 3D-Studio-MAX somit das gesamte für die Simulation nötige Modell. Seine grafische Benutzeroberfläche wird für die Eingabe und Manipulation dieser dort gehaltenen Daten verwendet.

²⁴ <http://www.microsoft.com/windows/products/windowsxp/>, 23.04.2007

²⁵ http://www.maxon.net/pages/products/cinema4d/cinema4d_d.html, <http://www.maxon.net/>, 23.04.2007

²⁶ <http://www.newtek.com/products/lightwave/>, <http://www.newtek.com/>, 24.04.2007

²⁷ <http://www.discreet.com/products/3dsmax/>, <http://www.discreet.com/>, 24.04.2007

²⁸ <http://www.alias.com/eng/products-services/maya/>, <http://www.alias.com/>, 24.04.2007

²⁹ <http://www.softimage.com/products/xsi/>, <http://www.softimage.com/>, 24.04.2007

Da auch der für das Verhalten eines Objektes relevante Typ des Objektes in Form des zugehörigen Typbezeichners in der Szene gespeichert wird, ebenso wie für das Verhalten relevante Parameterwerte, enthält die Szene damit implizit auch Informationen über das Verhalten der Objekte.

Da 3D-Studio-MAX nicht nur ein Modellierungs- sondern vor allem auch ein Animationswerkzeug ist, kann es zeitliche Änderungen fast³⁰ aller in der Szene verfügbaren Eigenschaften der 3D-Objekte über das Zeitintervall der Animation festhalten. Dadurch ist es möglich, die Ergebnisse jedes Simulationsschrittes festzuhalten, so dass nach einer beendeten Simulation eine entsprechende Animation der Szene vorliegt. Diese Animation wird in der zur Szene gehörigen Datei gespeichert. Ebenso ist es möglich, manuell erstellte Animationen von Eigenschaften der Objekte bei der Simulation zu berücksichtigen.

Für die angestrebte Gesamtarchitektur waren zwei Fähigkeiten von 3D-Studio-MAX besonders wichtig:

Zum einen wird „Object Linking and Embedding³¹“ (OLE) unterstützt. Dadurch wird es möglich, das Programm von außen zu steuern. Dabei können vorher für OLE freigegebene Funktionen aufgerufen werden, die in der Skriptsprache MAXScript implementiert wurden. Durch einen daran anknüpfenden Mechanismus ist es möglich, auch längeren in MAXScript verfassten Programmcode von 3D-Studio-MAX ausführen zu lassen, der vorher durch das in Java geschriebene Simulations-Hauptprogramm dynamisch erzeugt wurde. So können Daten aus der aktuellen in 3D-Studio-MAX gespeicherten 3D-Szene vom Java-Programm ausgelesen und auch in die Szene hineingeschrieben werden. Ebenso können fast beliebige andere Befehle an das 3D-Animationsprogramm gegeben werden. Auf diese Weise kann auf der Basis einer Simulation in Java eine Animation in der 3D-Szene erzeugt werden.

Darüber hinaus können jedem Objekt in der Szene zusätzliche Attribute zugeordnet werden, so genannte „Custom Attributes“. Für diese Attribute kann ein grafisches Benutzerinterface definiert werden, so dass die Werte dieser Attribute vom Benutzer gelesen und verändert werden können. Abhängig vom Datentyp dieser Werte können meistens auch zeitliche Veränderungen dieser Werte als Animation gespeichert werden. Dadurch wird es möglich, Parameter in der Szene zu speichern, die sich auf das Verhalten der Objekte beziehen.

Werden solche Parameter manuell festgelegt und eventuell animiert, können diese Parameter zur abstrakten Steuerung der Verhaltensweisen von Objekten verwendet werden. Ein Fahrzeug kann für seine Verhaltensweise einen Parameter haben, der den Einschlagwinkel der Lenkung festlegt. Dieser Parameter würde dann im Normalfall automatisch vom Verhalten des Fahrzeugs bestimmt werden, wenn es z.B. versucht, dem Verlauf einer Straße zu folgen. Alternativ kann der Benutzer diesen Parameter in einem bestimmbar Zeitintervall in der Animation manuell kontrollieren. Damit würde er die Fahrtrichtung des Fahrzeugs bestimmen, **ohne** dabei andere automatische Verhaltensweisen wie die automatische Steuerung der Fahrtgeschwindigkeit unwirksam zu machen. Siehe dazu Kap. 6.6.

³⁰ Nicht alle Eigenschaften der 3D-Objekte sind in 3D-Studio-MAX animierbar. Nicht animierbaren Eigenschaften kann nur ein Wert zugewiesen werden, der sich im Verlauf der Animation nicht ändert.

³¹ <http://support.microsoft.com/kb/86008/en-us>, 24.04.2007

5.3 Tonerzeugung mit Foley Studio MAX

Foley Studio MAX³² ist ein Plugin für 3D-Studio-MAX. Es ermöglicht die Erstellung spezieller Objekte in einer Szene, die als Tonquellen dienen. Auf diese Weise kann die Simulation auch Toneffekte erzeugen. Die Vorteile einer automatischen Erzeugung von Tönen wurden in Kapitel 2.1 ab Seite 30 erläutert.

Jeder Tonquelle kann eine Tondatei zugeordnet werden. Viele Eigenschaften dieser Tonquellen sind animierbar, so dass für jeden Zeitpunkt im Animationsintervall unter anderem die Tonhöhe und die Lautstärke, die Position der Tonquelle sowie Einsetzen bzw. Enden des Tones festgelegt werden können.

Auf der Basis der Positionsveränderungen einer Tonquelle kann ein Dopplereffekt erzeugt werden. Neben weiteren Effekten ist es auch möglich festzulegen, wie stark andere Gegenstände in der 3D-Szene eine Tonquelle abschwächen, wenn sie sich zwischen der Tonquelle und der Kamera befinden. Es ist auch einstellbar, in welcher Form die Lautstärke einer Tonquelle mit zunehmender Entfernung zur Kamera abnimmt.

Beim Rendern eines 3D-Animationsfilmes wird eine Folge von Einzelbildern aus der Sicht einer virtuellen Kamera erzeugt. Analog dazu kann für eine Kamera eine Tondatei erzeugt werden, welche die erzeugten Töne aller Tonquellen in der 3D-Szene aus der Perspektive der Kamera unter Berücksichtigung der oben genannten Eigenschaften dieser Tonquellen enthält. Es können Mono- und Stereo-Ton erzeugt werden sowie Surround-Sound mit bis zu 6 Tonkanälen. Dabei wird unter Umständen für jeden Tonkanal eine eigene Audiodatei erzeugt.

Die so erzeugten Tondateien können mit dem zugehörigen erzeugten 3D-Film kombiniert werden, so dass viele zum Film passende Toneffekte bereits in dem Film enthalten sind, ohne dass eine aufwändige Nachbearbeitung nötig ist.

Foley Studio MAX kann bei der Erzeugung von Tonspuren durch die enge Integration in ein 3D-Animationstool auf eine Vielzahl an Daten zurückgreifen, die aus einem bereits gerenderten Film für eine manuelle Nachvertonung nur noch schwer rekonstruierbar sind. So kann nur ein Werkzeug wie Foley Studio MAX Daten darüber zur Verfügung haben, wie sich ein Objekt genau verhält, das sich **nicht** sichtbar **hinter** der Kamera befindet, und es kann damit sehr präzise auch für solche Objekte automatisch Toneffekte erzeugen. Dies wäre bei einer manuellen Nachvertonung nur schwer möglich.

Dieses Tool wird hier verwendet, um das Verhalten der Objekte in einer Szene auch zur Erzeugung von passenden Tönen zu verwenden. Bei Szenen mit sehr vielen Objekten ist eine manuelle Erzeugung von passenden Toneffekten ähnlich schwierig wie eine manuelle Erzeugung des **sichtbaren** Verhaltens so vieler Objekte. Deshalb sollte diese Möglichkeit keinesfalls ungenutzt bleiben. Foley Studio MAX ermöglicht dies, ohne dabei eine große technische Herausforderung zu stellen, da die tonerzeugenden Objekte in einer 3D-Szene genau so gesteuert und animiert werden können wie sichtbare geometrische Objekte. Es stellt keinen technischen Unterschied dar, ob der Radius einer Kugel durch eine Verhaltensweise gesteuert werden soll oder die Lautstärke einer Tonquelle. Die tatsächliche Erzeugung der resultierenden Tondaten wird von Foley Studio MAX übernommen, ähnlich wie das Rendern des aus der 3D-Animation resultierenden Filmes von 3D-Studio-MAX mit Hilfe von Final Render durchgeführt wird.

³² <http://www.boomerlabs.com/cart/home.php?cat=1>, 24.04.2007

5.4 Rendering mit Final Render

Das Produkt Final Render³³ der Firma Cebas³⁴ ist ein Plugin für 3D-Studio-MAX, welches das Rendern der Szenen in 3D-Studio-MAX übernehmen kann. Im Vergleich zum in 3D-Studio-MAX integrierten so genannten „Scanline-Renderer“ bietet Final Render verschiedene Vorteile bezüglich des Realismus der erzeugten Darstellungen. So ist Final Render eines der wenigen Renderingsysteme, die in der Lage sind, in ausreichend kurzer Zeit gleichzeitig sowohl echte 3D-Bewegungsunschärfe als auch echte 3D-Tiefenunschärfe zu berechnen. Dabei werden keine nachträglich angewendeten Bildfilter verwendet, die gegenüber der aufwändigeren Berechnung dieser Effekte im 3D-Raum Nachteile haben. Auch weitere neuere Technologien wie „Global Illumination“ werden unterstützt. Es soll hier nicht im Detail auf diese Effekte eingegangen werden, sie tragen jedoch alle zum Realismus der erzeugten Bilder bei.

Da MaxControl Techniken aus der Echtzeitanimation nutzt, wie sie z.B. in Computer- und Videospiele Verwendung finden, hätte man auch die Entwicklung eines reinen Echtzeitanimationssystems mit einer 3D-Darstellung in Echtzeit in Betracht ziehen können, zumal diese Systeme durch die stetig voranschreitende Hardwareentwicklung immer höhere Darstellungsqualität erreichen. Da MaxControl jedoch nicht dazu konzipiert ist, die automatisch generierten Animationen in Echtzeit darzustellen, kann ein Softwarerendingsystem eingesetzt werden, das die Animationen zwar nicht in Echtzeit darstellen kann, dafür aber eine überlegene Darstellungsqualität besitzt. Um die Vorteile dieser Herangehensweise gegenüber reinen Echtzeitanimationssystemen herauszustellen, ist die Wahl eines Softwarerendingsystems wie Final Render erforderlich, das allen bisher existierenden Echtzeitrenderingsystemen in der Qualität der erzeugten Bilder überlegen ist.

5.5 Volumetrisches Rendering mit Afterburn

Das Plugin Afterburn³⁵ der Firma Sitni Sati³⁶ findet aus vergleichbaren Gründen Verwendung wie Final Render. Afterburn erweitert das verwendete Rendering-System um Funktionen zur realistischen Darstellung von atmosphärischen Phänomenen wie Feuer, Rauch, Wolken und Explosionen. Die Darstellung dieser Phänomene wird dabei nicht mehr auf der Basis herkömmlicher Polygone durchgeführt, sondern es werden Gasvolumina auf der Basis mathematischer Funktionen definiert und dargestellt.

Es gibt verschiedene solcher nicht-polygonbasierten Rendering-Techniken. So genanntes Image-Based-Rendering wird z.B. in [Popes00] zur Darstellung natürlicher Szenen aus beliebigen Blickwinkeln verwendet, wobei Fotos dieser Szenen mit Tiefeninformationen für jeden Bildpunkt eingesetzt werden. [Alia01] geht hier einen Schritt weiter und ermöglicht virtuelle Rundgänge durch beliebig große Umgebungen mit Hilfe von Image-Based-Rendering auf der Basis gefilmter Rundumsichten entlang sich schneidender Pfade innerhalb der Umgebung. Das in [Carr03] vorgestellte System ermöglicht es, Filmsequenzen von Schauspielern interaktiv durch Image-Based-Rendering aus beliebigen Perspektiven zu betrachten. In [Matus02] können sonst schwierig zu erfassende Gegenstände z.B. mit Fell verarbeitet werden, so dass diese vor neuen Hintergründen aus unterschiedlichen Blickwinkeln und mit neuen Beleuchtungsverhältnissen dargestellt werden können, wobei

³³ <http://www.finalrender.com/products/products.php?UD=10-7888-35-788&PID=36>, 25.04.2007, <http://www.finalrender.com/>, 25.04.2007

³⁴ <http://www.cebass.com/>, 25.04.2007

³⁵ <http://www.afterworks.com/AfterBurn.asp>, <http://www.afterworks.com/>, 25.04.2007

³⁶ <http://www.afterworks.com/company.asp>, <http://www.afterworks.com/>, 25.04.2007

ebenfalls Image-Based-Rendering zum Einsatz kommt. Ähnlich ermöglicht auch [Mass03] eine Darstellung erfasster Gegenstände unter neuen künstlichen Beleuchtungsverhältnissen. Es wird bei beiden Systemen kein Rendering auf der Basis von Polygonen verwendet, sondern auf der Basis der ursprünglich von dem Objekt erfassten Bilder zusammen mit Messwerten über das Verhalten der Oberfläche des Objektes bei Lichteinfall aus verschiedenen Richtungen. Ähnlich geht auch [Chen02] vor, wobei hier auch ein 3D-Modell aus Polygonen erzeugt wird und Echtzeitfähigkeit durch den Einsatz von 3D-Grafik-Hardware erreicht wird. In [Schödl02] werden Filmsequenzen von realen Objekten wie Tieren, z.B. von Hamstern oder Fliegen, so aufbereitet, dass die Einzelbilder dieser Sequenzen automatisch zu neuen Animationen zusammengesetzt werden können. Diese Sequenzen können als Objekte in eine Szene eingebaut werden, so dass mehrere einzeln gefilmte Tiere beispielsweise entlang festgelegter Pfade laufen oder sich gegenseitig ausweichen.

Auf Image-Based-Rendering basiert auch [Xu01]. Hier werden realistisch die Fasern von Stoffen dargestellt. Die dort verwendete Rendering-Technik ist dem Volumetric-Rendering zuzuordnen (siehe [Jack06]), das am häufigsten zum Darstellen von Gasvolumina verwendet wird. Das erste Renderingsystem dieser Art wurde in [Kaji84] entwickelt. Afterburn implementiert ebenfalls eine solche Rendering-Technik. Diese Art der Darstellung von Gasvolumina ist Thema vieler weiterer Arbeiten wie [Schpok03, Schpok05, Foster97, Dob00, Fedkiw01, Lam02, Harris03, Silver97], die sich in den meisten Fällen auch mit der Simulation des Verhaltens von Gasvolumina (Rauchentwicklung, Flammen, Explosionen) beschäftigen (siehe Kap. 4.1.2 und 4.1.1). Allgemeinere Szenen mit komplexer Geometrie werden in Arbeiten wie [Neyret98] ebenfalls mit volumetrischen Verfahren gerendert.

5.6 Mechaniksimulation mit Macromedia Shockwave

Macromedia Shockwave³⁷ wird hier als Komponente zur Simulation von Mechanik verwendet. Auf diesem Anwendungsgebiet ermöglicht diese Softwarekomponente im Wesentlichen die Simulation des mechanischen Verhaltens von starren Körpern und Federn.

Macromedia Shockwave ist eigentlich ein Entwicklungssystem, mit dem man verschiedenartige Multimediainhalte erstellen kann, die mit Hilfe eines speziellen Abspielprogramms in einem Internet-Browser dargestellt werden können. Diese Inhalte können jedoch auch so konfiguriert werden, dass sie unabhängig von einem Internet-Browser laufen können.

Shockwave besteht dabei aus zwei Werkzeugen. Der „Macromedia Director“ wird zum Erstellen der Shockwave-Inhalte verwendet. Diese Inhalte können kleine Animationen sein, aber auch komplexe interaktive Anwendungen. Die Interaktivität wird dabei insbesondere dadurch unterstützt, dass das Verhalten von Inhalten mit Hilfe einer speziellen Skriptsprache definiert werden kann. Die so erstellten Projekte können mit dem Plugin „Shockwave Player“ in einem Internet-Browser dargestellt und in Webseiten eingebunden werden.

Durch die oben genannte Skriptsprache wird eine Steuerung der mit „Macromedia Director“ erstellten Multimediainhalte von außen möglich, ähnlich der externen Steuerung von 3D-Studio-MAX. Shockwave ist unter anderem für die Erstellung von Spielen konzipiert. Um interaktive Spiele mit 3D-Grafik und realistischen Bewegungen zu ermöglichen, beinhaltet

³⁷ Entwicklungsprogramm: <http://www.macromedia.com/software/director/>, 25.04.2007,
Abspielprogramm: <http://www.adobe.com/products/shockwaveplayer/>, 25.04.2007

Shockwave unter anderem die Mechanik-Simulationssoftware Havok Physics³⁸. Havok Physics ist eine auf dem Spielmarkt führende Simulationssoftware für Mechanik. Viele Spielehersteller verwenden diese Softwarekomponente, statt eine entsprechende Komponente zeit- und kostenintensiv selbst zu entwickeln³⁹. Auch das in Kapitel 4.1.3 erwähnte Tool „Reactor“ verwendet Havok Physics, um mechanische Vorgänge zu simulieren.

Da Shockwave allerdings im Gegensatz zu Reactor stark für die Erstellung und Darstellung interaktiver Inhalte konzipiert ist, ist auch die Einbindung der Mechanik-simulationskomponente „Havok Physics“ hier stärker auf Interaktionen von außen ausgerichtet als bei dem Tool Reactor. Daher ist es hier möglich, bei einer Simulation physikalische Eigenschaften (z.B. Massen, Zustände (z.B. Kräfte, Impulse, Objektpositionen und Beschleunigungen) und Ereignisse wie Kollisionen während der Simulation sowohl von außen zu lesen als auch von außen zu beeinflussen. Dadurch ist es nun möglich, jedes physikalische Objekt während der Simulation aufgrund seiner Verhaltensweisen mit seiner Umwelt interagieren zu lassen, indem es z.B. selbständig neue Kräfte setzt.

Ein Argument dafür, hier Shockwave zu verwenden und nicht direkt das Havok Physics SDK, ist vor allem auch, dass Shockwave mit derzeit ab 1.712,41€⁴⁰ preislich sehr viel günstiger ist als das reine Havok Physics SDK, das sogar in einer Grundversion zum Zeitpunkt der Erstellung dieser schriftlichen Arbeit ab 50.000€ kostete. Durch den Erwerb des Havok Physics SDK hätten sich Vorteile ergeben, welche die Preisdifferenz zu Shockwave jedoch nicht gerechtfertigt hätten.

Im Vergleich zu dem oben genannten Werkzeug „Reactor“ unterliegt die Mechaniksimulationskomponente von Shockwave einigen Einschränkungen. So können z.B. nur starre Körper simuliert werden und keine verformbaren Körper, Seile oder Stoffe (Kleidung, Tischdecke etc.), was in Reactor möglich ist. Außerdem können keine Dreh- und Schiebegelenke wie in Reactor definiert werden, diese können allerdings mit Einschränkungen durch spezielle Anordnungen von Federn ersetzt werden.

Um Shockwave nun für die Simulation von Mechanik in einer gegebenen 3D-Szene zu verwenden, werden alle hierzu erforderlichen Daten an eine mit „Macromedia Director“ entwickelte Anwendung übergeben. Dazu gehören unter anderem die Position, die Masse und die geometrische Struktur der in ihrem mechanischen Verhalten zu simulierenden Objekte. Die Shockwave-Anwendung kann nun jeweils einen Schritt des physikalischen Verhaltens simulieren, und die Ergebnisse werden an das in Java implementierte Hauptprogramm MaxControl zurückgegeben. MaxControl, welches das nicht-mechanische Verhalten der Objekte steuert, kann nun die simulierten Objekte auf das Ergebnis des jeweils letzten Schrittes der Mechaniksimulation ggf. intelligent reagieren lassen, indem z.B. neue Kräfte gesetzt werden, welche die Shockwave-Anwendung dann im nächsten Mechanik-Simulationsschritt berücksichtigt.

³⁸ <http://www.havok.com/content/view/17/30/>, 25.04.2007

³⁹ Spiele, die Havok Physics verwenden: <http://www.havok.com/content/blogcategory/29/73/>, 25.04.2007

⁴⁰ Derzeitiger Preis für das Shockwave-Entwicklungssystem „Macromedia Director MX“ (das Abspielprogramm ist kostenlos): https://store2.adobe.com/cfusion/store/index.cfm?store=OLS-DE&nr=1#loc=de_de&view=ols_prod&store=OLS-DE&categoryOID=285292&distributionOID=103&nr=1, 25.04.2007

5.7 Definition von Verhaltensweisen mit Java

Java⁴¹ wurde als Programmiersprache für MaxControl verwendet, um damit das Hauptprogramm zu entwickeln und einem Benutzer die Spezifikation von Objekttypen und Verhaltensweisen für die 3D-Objekte in einer Szene zu ermöglichen. Davon ausgenommen ist das mechanische Verhalten der 3D-Objekte, das durch eine externe in Shockwave implementierte Komponente simuliert wird (siehe vorangehendes Kap. 5.6). Bei der Verwendung von Java kommt hier insbesondere die Objektorientierung zum Einsatz, wie bereits in Kapitel 2.1 ab Seite 24 erwähnt. Jedem Objekt in einer 3D-Szene wird dazu eine in Java definierte Klasse zugeordnet, die das Verhalten dieses Objekts grundsätzlich festlegt. Durch Simulation der so festgelegten Verhaltensweisen wird dann automatisch eine Animation erzeugt. Die entsprechenden Klassen enthalten zu diesem Zweck direkt oder indirekt⁴² Methoden, die in jedem Simulationsschritt für jedes zur jeweiligen Java-Klasse gehörende 3D-Objekt aufgerufen werden. Dadurch werden Zustandsänderungen spezifiziert, die in jedem Simulationsschritt für die Objekte durchgeführt werden. Dies wird in Kapitel 8.6 näher erläutert.

Alle Eigenschaften der Objekte, die für den durch Java bestimmten Teil der Simulation erforderlich sind, werden aus dem 3D-Animationsprogramm ausgelesen und an die entsprechenden Java-Objekte weitergegeben. Nicht relevante Informationen werden nicht ausgelesen, um Speicherplatz und Kommunikationszeit zwischen den Softwarekomponenten einzusparen. So wird in vielen Fällen die geometrische Struktur eines Objektes gar nicht an das zugehörige Java-Objekt übergeben, weil diese Struktur oft nur für die eventuelle Simulation des mechanischen Verhaltens dieses Objektes erforderlich ist, und das mechanische Verhalten der Objekte wird wie in Kapitel 5.6 beschrieben durch eine mit Macromedia Shockwave implementierte Komponente und nicht in Java simuliert.

Eigenschaften von Java-Objekten werden in Java als „Felder“ der zugehörigen Klasse bezeichnet.

Es werden hier die Java-Laufzeitumgebung der Firma Sun sowie deren Entwicklungssystem „Java-SDK“ verwendet. Beide Produkte sind in der „Standard Edition“ frei erhältlich. Da die Firma Microsoft⁴³ nach einem Rechtsstreit mit der Firma Sun ihre eigene Java-Laufzeitumgebung und ihre Java-Entwicklungsumgebung nicht mehr weiterentwickeln darf, unterstützt nur die Java-Laufzeitumgebung der Firma Sun den aktuellsten Stand der Java-Technologie.

5.8 Transformationen mit Java-3D

Java-3D⁴⁴ ist ein spezielles Java-API der Firma Sun, welches die Erstellung, Verarbeitung und Darstellung von 3D-Szenen durch Java-Applikationen oder Java-Applets ermöglicht.

Hier wurden Klassen aus diesem API verwendet, die den Umgang mit vielen hier erforderlichen mathematischen Konstrukten erleichtern. So stellt dieses API Klassen zum Umgang mit Transformationen (siehe Kapitel 3.2), Punkten und Vektoren in 3D-Raum zur Verfügung. Der Einsatz dieser Klassen wird in Kapitel 10.1.3 näher erläutert.

⁴¹ <http://java.sun.com/>, 27.04.2007

⁴² Siehe Erläuterungen zu Unter-Verhaltensweisen in den Kapiteln 7.1 und 7.2.1.

⁴³ <http://www.microsoft.com/>, 29.04.2007

⁴⁴ <https://java3d.dev.java.net/>, 29.04.2007

Dieses API wird hier also nur zum Umgang mit rein mathematischen Konstrukten und nicht zur 3D-Darstellung verwendet.

5.9 Microsoft Internet Explorer als Basis-Software

Der Microsoft Internet Explorer⁴⁵ ist eigentlich ein Werkzeug, um Internet-Seiten darzustellen und zwischen ihnen zu navigieren. Da Internet-Seiten inzwischen aus verschiedensten Inhalten bestehen, müssen Internet-Browser viele Datenformate und Darstellungsformen unterstützen. Neben verschiedenen Bildformaten, die dargestellt werden können, unterstützt der Internet Explorer auch Plugins von Drittherstellern, um erweiterte Inhalte darzustellen. Um interaktive Seiten darzustellen, werden neben Java-Applets auch die Skriptsprachen Java-Skript⁴⁶ und Visual Basic⁴⁷ Script unterstützt.

Diese Fähigkeiten machen den Internet Explorer inzwischen zu einer nicht zu unterschätzenden Laufzeitumgebung für Anwendungsprogramme. So wurde der Internet Explorer hier verwendet, um die verwendeten Softwarekomponenten miteinander zu verbinden und deren Kommunikation zu ermöglichen.

Das hier entwickelte Hauptprogramm wird als Java-Applet realisiert, das im Internet Explorer läuft. Damit das Java-Applet keinen Zugriffseinschränkungen unterliegt, wird es digital signiert. Es läuft in der Sun-Java-Laufzeitumgebung, die wiederum als Plugin im Internet Explorer läuft. Java-Applets können Befehle vom Internet-Explorer empfangen und ihm auch Befehle geben. Somit können alle Fähigkeiten des Internet-Explorers vom Java-Applet genutzt werden. Dies wird hier unter anderem dafür genutzt, dass sich die Benutzeroberfläche des Java-Applets zu einem großen Teil in der einbettenden HTML-Seite befindet und mit dem Applet kommuniziert. Es werden die vom Internet Explorer unterstützten Sprachen Java-Skript und Visual-Basic-Skript verwendet, um das Applet mit dem 3D-Animationswerkzeug 3D-Studio-MAX kommunizieren zu lassen. Die in Shockwave implementierte Anwendung für die Simulation mechanischer Vorgänge (siehe Kapitel 5.6) läuft mit Hilfe des Shockwave-Plugins in derselben HTML-Seite wie das Applet. Über Java-Skript können das Applet und die Shockwave-Anwendung miteinander kommunizieren.

⁴⁵ <http://www.microsoft.com/windows/ie/>, 29.04.2007

⁴⁶ <http://msdn2.microsoft.com/en-us/library/ms970435.aspx>, 29.04.2007

⁴⁷ <http://msdn.microsoft.com/vbasic/>, 29.04.2007,
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/0a8270d7-7d8f-4368-b2a7-065acb52fc54.asp>, 29.04.2007

6 Simulationskonzept von MaxControl

6.1 Vereinfachende Annahmen

Die folgenden Kapitel 6 bis 9.5 beschreiben die konzeptionellen und programmiersprachlichen Grundlagen von MaxControl. Es werden verschiedene Annahmen gemacht, welche die Beschreibung des Grundkonzeptes erleichtern und klarere Strukturen ermöglichen.

Diese Annahmen können jedoch in der technischen Umsetzung, die in Kapitel 10 beschrieben wird, nicht vollständig beibehalten werden. Dennoch wird das unter diesen Annahmen entwickelte Konzept weitgehend realisiert, es muss lediglich aus technischen Gründen von Voraussetzungen ausgegangen werden, welche die Umsetzung des Konzeptes erschweren oder nur über Umwege realisierbar machen.

Ein Hauptunterschied zwischen dem hier erläuterten Konzept und der technischen Umsetzung besteht in der hohen Anzahl verschiedener Werkzeuge, die bei der Umsetzung verwendet wurden. Diese Werkzeuge müssen in MaxControl intensiv miteinander kommunizieren und jeweils verschiedene Teile des Modells speichern, wobei deren Schnitte nicht-leer sind. Hier wird hingegen angenommen, dass es nur eine einzige Softwarekomponente gibt, welche allein das gesamte Modell speichert.

Im Folgenden werden einige Eigenschaften dieser Softwarekomponente vorgestellt. Zunächst wird angenommen, dass nicht unbedingt ein spezifisches 3D-Animationswerkzeug wie 3D-Studio-MAX verwendet wird, sondern ein beliebiges, das die typischen Lösungskonzepte und Fähigkeiten der meisten gängigen 3D-Animationstools beinhaltet. Beispiele dafür sind die Darstellung und Speicherung dreidimensionaler Objekte durch Polygone, Animationen auf Keyframe-Basis sowie das Entwerfen von Szenen und das Rendering der Szenen.

Es wird weiterhin angenommen, dass dieses Werkzeug eine direkte Unterstützung von Java-Code aufweist. So kann jedes 3D-Objekt zusätzlich durch eine Instanz einer Java-Klasse repräsentiert werden. Dabei ist die Java-Klasse `MC_Object`⁴⁸ der Basistyp, der jedem 3D-Objekt direkt nach dessen Erstellung zunächst zugeordnet ist. Daraufhin kann der Benutzer dem 3D-Objekt einen anderen Java-Typ zuordnen, der ein Untertyp von `MC_Object` sein muss. Der Java-Typ wird auch als Objekttyp bezeichnet. Die Simulation verwendet für jedes 3D-Objekt aus der Szene eine entsprechende Instanz des zugeordneten Java-Typs. Diese Instanzen werden weiter unten auch als Simulationsobjekte bezeichnet. Der Java-Typ hat hauptsächlich die Aufgabe, das Verhalten des 3D-Objektes bei Simulationen festzulegen. Der Java-Typ bestimmt außerdem, welche zusätzlichen nicht-geometrischen Eigenschaften für die zugehörigen Simulationsobjekte zur Verfügung stehen sollen (z.B. Steuerrichtung bei einem Auto), und welche Eigenschaften aus dem geometrischen Grundtyp (z.B. Kugel, Quader oder Lichtquelle) für die zugehörigen Simulationsobjekte übernommen werden sollen, wie beispielsweise der Radius bei einer Kugel. Jedem 3D-Objekt sind damit sowohl ein geometrischer Grundtyp als auch ein Java-Typ zugeordnet.

Ebenso wird das Vorhandensein einer Physik-Simulationskomponente direkt im 3D-Animationsprogramm vorausgesetzt, welche die hier benötigten Fähigkeiten beinhaltet.

In dieser Umgebung kann das Simulationssystem MaxControl als Java-Programm entwickelt werden, das direkt mit den 3D-Objekten und den Instanzen ihres jeweils zugeordneten Java-

⁴⁸ Siehe Kap. 8.4.

Typs arbeiten kann, um deren zeitliches Verhalten zu simulieren. Es soll keine spezielle Kommunikation zwischen dem 3D-Animationsprogramm und dem Java-Programm MaxControl erforderlich sein, MaxControl soll direkt auf die Datenstrukturen und Befehlsschnittstellen des 3D-Animationsprogrammes zugreifen können. Dies wäre am einfachsten möglich, wenn auch das 3D-Animationsprogramm in Java implementiert wäre und das in Java entwickelte Simulationsprogramm MaxControl eine Erweiterung des 3D-Animationsprogrammes wäre.

So kann konzeptionell eine Software entwickelt werden, die alle erforderlichen Aufgaben wie das Entwerfen von Szenen, das Simulieren des Verhaltens von Objekten sowie das *Rendering* von Filmen erfüllen kann. Es entsteht das Konzept einer idealen Lösung, deren direkte Umsetzung in dieser Form jedoch einen hier nicht vertretbaren Aufwand bedeuten würde (vgl. Kapitel 4.1.7). Denn eine solche Lösung wäre nur durch die Implementierung eines umfangreichen Plugins z.B. für 3D-Studio-MAX möglich oder durch eine Standalone-Lösung, bei der ein Programm implementiert wird, das alle in den Annahmen beschriebenen Anforderungen von sich aus erfüllt. Dabei müsste dann mindestens ein 3D-Animationsprogramm wie 3D-Studio-MAX erstellt werden, was für sich schon den Rahmen dieser Arbeit übersteigen würde.

6.2 Basisprinzipien

MaxControl soll das vorher in Verhaltensweisen festgelegte Verhalten von Objekten einer 3D-Szene simulieren und so automatisch eine Animation dieser Objekte erzeugen. Wie diese Simulation genau abläuft, wird in den Kapiteln 8.6, 9 und 10.2.2 erklärt, hier sollen nur einige wichtige Grundkonzepte vorgestellt werden.

Traditionell wurden Animationen mit 3D-Animationsprogrammen meistens dadurch festgelegt, dass man in die Frames eines vorgegebenen Animationsintervalls manuell die Werte aller für die Darstellung der Szene relevanten animierbaren Eigenschaften einträgt, in der Regel unterstützt durch Keyframe-Techniken.

Bei MaxControl wird davon ausgegangen, dass ein großer Teil der 3D-Objekte mit Verhaltensweisen versehen werden kann, mit deren Hilfe automatisch die gewünschten Animationen dieser Objekte erstellt werden können. Dadurch wird es möglich, diejenigen Teile der Frames, die 3D-Objekte mit Verhaltensweisen betreffen, ausgehend von einem total definierten Anfangszustand durch Simulation herzustellen. Damit wird die manuelle Tätigkeit des Animierens weitgehend reduziert. Auf weiterhin erwünschte und erlaubte manuelle Eingriffe auch bei Objekten mit Verhaltensweisen wird in späteren Kapiteln eingegangen.

Das Verhalten eines 3D-Objekts ist im Allgemeinen nicht nur von seinen geometrischen Grundeigenschaften abhängig, sondern die Verhaltensweisen verwenden zusätzliche Eigenschaften, wie die aktuelle Geschwindigkeit eines Fahrzeugs oder die Blinkfrequenz einer Blinklampe. Damit diese zusätzlichen Eigenschaften auch in dem verwendeten 3D-Animationsprogramm sichtbar und manipulierbar werden, müssen sie dem 3D-Objekt dort hinzugefügt werden. Zu jeder Szene gehört eine Folge von Zuständen, die so genannte Szenenzustandsfolge, welche zu jedem Frame der Animation einen eigenen Zustand für alle 3D-Objekte der Szene speichert. Diese Zustände werden jeweils um die hinzugefügten Eigenschaften erweitert.

MaxControl ist so konzipiert, dass die eigentliche Simulation auf Zuständen abläuft, welche weitgehend⁴⁹ durch diejenigen Objekteigenschaften bestimmt sind, die für die Simulation freigegeben wurden (siehe Kap. 6.3). Neben den neu hinzugefügten Eigenschaften müssen auch Grundeigenschaften der 3D-Objekte wie der Radius einer Kugel oder die Position des 3D-Objektes im Raum dem Simulationssystem zugänglich gemacht werden, denn MaxControl kann solche Grundeigenschaften der 3D-Objekte weder lesen noch verändern, wenn diese nicht explizit für das Simulationssystem freigegeben wurden. Dies wird in Kapitel 6.3 näher erläutert.

Die Simulation des Verhaltens der Objekte in einer 3D-Szene läuft in einzelnen Simulationsschritten ab. Grundsätzlich soll jeder Simulationsschritt die Zustände der 3D-Objekte in einem bestimmten Frame der Animation bestimmen, allerdings beschränkt auf die hinzugefügten neuen Eigenschaften und die freigegebenen Basiseigenschaften der 3D-Objekte. Ein Simulationsschritt wird hier als „Simulationsschritt zu Frame n“ bezeichnet, wenn er ausgehend vom Zustand zu Frame n-1 den nächsten Zustand der Animation bestimmt, der zu Frame n gehört. Der erste Frame der Animation trägt den Anfangszustand, mit dem die Simulation beginnt. Die so erzeugten Zustände stellen in der Szene als Zustandsfolge eine Animation dar. Das insgesamt simulierte Zeitintervall entspricht dem Animationsintervall, das im Zusammenhang mit der Simulation auch als Simulationsintervall bezeichnet wird.

Die Verhaltensweisen können für 3D-Objekte, die als starre Körper nach den Gesetzen der Mechanik simuliert werden, auch Kräfte erzeugen, die dann implizit das mechanische Verhalten solcher Objekte steuern.

Die Verhaltensweisen arbeiten oft mit Näherungsverfahren, welche die gewünschten Vorgänge nachbilden. Die Genauigkeit solcher Verfahren kann in der Regel durch die Berechnung von Zwischenschritten erhöht werden. Dies gilt insbesondere für die Simulation von Mechanik. Daher werden bei der Berechnung des Zustandsübergangs von einem Frame n-1 zum folgenden Frame n eventuell Simulations-Zwischenschritte durchgeführt. Dies wird in Kapitel 6.5 genauer erklärt.

6.3 Simulationseigenschaften (SPs)

Die Abkürzung SP steht hier für „Simulation Property“. SPs sind genau die Objekteigenschaften, auf welche die Verhaltensweisen der Objekte lesend oder auch schreibend zugreifen sollen. Das Simulationssystem MaxControl kann nur auf Objekteigenschaften zugreifen, die ihm als SPs zugänglich gemacht wurden. Im Java-Typ (=„Objektyp“), der jedem 3D-Objekt zugeordnet wird, ist festgelegt, welche Objekteigenschaften des 3D-Objekts zu SPs werden.

Ob eine Objekteigenschaft dem Simulationssystem als SP zugänglich gemacht werden sollte, hängt in erster Linie nicht von der Eigenschaft selbst ab, sondern vom Verhalten der 3D-Objekte. Ob z.B. die Helligkeit einer Lichtquelle als SP freigegeben werden sollte, hängt davon ab, ob die Verhaltensweise der Lichtquelle oder die Verhaltensweisen anderer 3D-Objekte diese Eigenschaft berücksichtigen oder verändern sollen. Man kann also nicht grundsätzlich festlegen, dass die Helligkeit einer Lichtquelle immer als SP freigegeben werden sollte, sondern das durch den Java-Typ jeweils zugeordnete Verhalten der Objekte muss für diese Entscheidung betrachtet werden. Die Wahl der freigegebenen SPs bleibt dann

⁴⁹ Es können die Werte interner Variablen der Verhaltensweisen und externe Datenquellen hinzukommen, siehe dazu Kapitel 6.4.

allerdings für den zugehörigen Objekttyp fest. Das heißt, für jedes 3D-Objekt, dem ein bestimmter Objekttyp „A“ zugeordnet wird, werden die gleichen SPs dem Simulationssystem zugänglich gemacht. Der Objekttyp muss entsprechend implementiert werden.

Auf welche SPs eines 3D-Objektes „A“ die Verhaltensweisen **anderer** 3D-Objekte eventuell zugreifen müssen, wird im Allgemeinen ebenfalls mit dem Objekttyp zusammenhängen, der dem 3D-Objekt „A“ zugeordnet wurde. Wollen Fahrzeuge z.B. auf den Signalzustand von Ampeln zugreifen, so liegt dies daran, dass die entsprechenden 3D-Objekte den Objekttyp „Ampel“ haben und sich damit auch so verhalten, dass sie z.B. zeitgesteuert ihren Signalzustand ändern. Dass Ampeln also ihren Signalzustand als SP dem Simulationssystem zugänglich machen sollten, ist also auch vom zugeordneten Objekttyp „Ampel“ abhängig. Deshalb ist es auch in einem solchen Fall sinnvoll, dass der Objekttyp die als SPs freizugebenden Objekteigenschaften bestimmt.

Welcher Objekttyp einem 3D-Objekt zugeordnet wird, hängt hauptsächlich von dem gewünschten Verhalten des Objektes ab, muss sich jedoch auch nach dem Grundtyp richten, den dieses Objekt bereits vor der Zuordnung eines Objekttyps im verwendeten 3D-Animationsprogramm hat. Ein spezieller Grundtyp wie „Lichtquelle“ kann eine notwendige Voraussetzung für die Zuordnung eines bestimmten Objekttyps zu diesem 3D-Objekt sein, damit im Objekttyp festgelegte SPs, die ursprünglich bereits im 3D-Objekt vorhandene Eigenschaften als SPs freigeben sollen (so genannte Standard-SPs, siehe unten), auch in dem 3D-Objekt zu finden sind. So gibt es in MaxControl den Objekttyp „blinkende Lichtquelle⁵⁰“. Dieser Objekttyp darf nur 3D-Objekten zugewiesen werden, die schon im Grundtyp eine Helligkeit als Eigenschaft haben, da in diesem Objekttyp die Helligkeit als SP freigegeben ist. Dieser Objekttyp kann also im Allgemeinen nur einer Lichtquelle zugeordnet werden. Der Typ „blinkende Lichtquelle“ legt als Verhalten fest, dass diese Lichtquellen ihren Helligkeitswert in Intervallen ändern. Für solche Lichtquellen ist klar, dass ihre Helligkeit durch ihre Verhaltensweisen geändert werden soll und somit als SP freigegeben werden muss. Der Typ „blinkende Lichtquelle“ muss jedoch nicht jedem Objekt in einer 3D-Szene zugewiesen werden, das bereits im 3D-Animationsprogramm den Grundtyp „Lichtquelle“ hat. Einer Lichtquelle kann über MaxControl auch ein anderer Java-Typ zugewiesen werden. Wird einer Lichtquelle z.B. nur der Java-Basistyp `MC_Object` für 3D-Objekte zugeordnet, der dieser Lichtquelle kein spezielles Verhalten gibt und auch keine SPs definiert, so wird auch keine ihrer Eigenschaften dem Simulationssystem als SP zugänglich gemacht.

SPs können Standard-Werte eines 3D-Objektes repräsentieren, wie den Radius einer Kugel, die Position eines 3D-Objektes im Raum oder die Höhe eines Zylinders. Die 3D-Objekte haben schon bei ihrer Erstellung durch das 3D-Animationsprogramm diese Eigenschaften, noch bevor ihnen ein Java-Typ zugeordnet wird. Die Eigenschaften können dann durch das Zuordnen der entsprechenden SPs zu ihrem Objekttyp der Simulation als so genannte Standard-SPs zugänglich gemacht werden. Durch sie kann beispielsweise der Radius einer Kugel von der Simulation berücksichtigt werden, um eine Kollision mit einer anderen Kugel festzustellen. Umgekehrt kann eine Simulation den Wert einer Standard-SP ändern, um eine erkennbare Veränderung im aktuellen Frame der Szene vorzunehmen, z.B. den Radius oder auch die Position einer Kugel zu ändern. Es muss nicht jede ursprüngliche Eigenschaft eines 3D-Objektes dem zugeordneten Objekttyp als Standard-SPs zugänglich gemacht werden, sondern nur die, welche auch von Verhaltensweisen geändert oder gelesen werden sollen.

⁵⁰ In der tatsächlichen technischen Umsetzung wird dieser Typ als „MCD_LightBlinking“ bezeichnet.

Die den 3D-Objekten für die Simulation neu hinzugefügten Eigenschaften, die ursprünglich nicht Eigenschaften der 3D-Objekte waren, werden immer als SPs im zugeordneten Objekttyp festgelegt. So kann ein Fahrzeug neben seinen Standard-SPs wie seiner Position zusätzliche SPs erhalten, wie die aktuelle Stellung des Gaspedals oder des Lenkrades oder die gewünschte Höchstgeschwindigkeit. Diese neuen SPs können dann durch Verhaltensweisen gelesen, berücksichtigt und verändert werden und sie sind auch im verwendeten 3D-Animationsprogramm als neue Eigenschaften des 3D-Objektes durch eigene automatisch generierte GUI⁵¹-Komponenten sichtbar. So kann der Benutzer über die Benutzeroberfläche des 3D-Animationsprogramms die Höchstgeschwindigkeit eines Fahrzeugs einstellen und die Verhaltensweisen des Fahrzeugs können diese Größe lesen und darauf entsprechend reagieren. Solche zusätzlichen SPs bilden gegenüber den Standard-SPs den Normalfall und erhalten daher auch meistens keine gesonderte Bezeichnung außer „SP“, zur Unterscheidung von Standard-SPs werden sie in Ausnahmefällen als „nicht-standard SPs“ bezeichnet. Der Begriff SP kann sowohl Standard-SPs als auch nicht-standard SPs bezeichnen.

MaxControl verwendet zur Simulation nicht die 3D-Objekte der Szene, sondern die in Kap. 6.1 genannten Simulationsobjekte. Für jedes in der Szene vorhandene 3D-Objekt wird eine Instanz seines Objekttyps erzeugt, die dann das zugehörige Simulationsobjekt ist. Das Simulationsobjekt hat die dem Objekttyp zugeordneten SPs als Eigenschaften, ihm fehlen die nicht als Standard-SPs freigegebenen Grundeigenschaften des zugehörigen 3D-Objekts.

Für jede als SP freigegebene oder hinzugefügte Eigenschaft eines 3D-Objektes wird ein Java-Objekt erzeugt, das diese SP repräsentiert. Technisch sind eine im 3D-Objekt vorhandene Eigenschaft und die zugehörige SP im entsprechenden Simulationsobjekt nicht identisch, sondern die SP stellt eine bidirektionale Verbindung zwischen sich selbst im Simulationsobjekt und der im 3D-Objekt vorhandenen Eigenschaft her.

SPs können wie die übrigen Eigenschaften eines 3D-Objektes entweder animierbar oder nicht animierbar sein. Nicht-animierbare SPs können durch die Simulation zwar gelesen aber nicht verändert werden. Folglich kann ihr jeweiliger Wert nur manuell eingestellt werden und gilt dann konstant für das gesamte Animationsintervall. Eine 3D-Szene hat möglicherweise globale Eigenschaften, die gar keinem 3D-Objekt angehören (siehe Kap. 3.3) und damit nicht als SPs gewählt werden können.

Eigenschaften der Szene und ihrer 3D-Objekte, die dem Simulationssystem nicht in als SPs zugänglich gemacht sind, werden als Nicht-SPs bezeichnet. Nicht-SPs können auch animierbare Eigenschaften sein, so dass für sie eine Animation vorliegen kann. Das Simulationswerkzeug MaxControl ignoriert Nicht-SPs vollständig, sie werden also niemals durch eine Simulation direkt⁵² gelesen oder verändert. So können Nicht-SPs von 3D-Objekten vor oder nach der Simulation auf herkömmliche Weise animiert werden, ohne dass sie durch die Simulation manipuliert oder von ihr berücksichtigt werden.

Die Werte jeder SP haben einen bestimmten Datentyp. Um dies für eine SP angeben zu können, werden SPs hier auch entsprechend genannt. So wird eine SP mit Fließkommawerten doppelter Genauigkeit auch als Double-SP bezeichnet. Handelt es sich um eine Standard-SP,

⁵¹ GUI ist die Kurzform von „Graphical User Interface“.

⁵² Für spezielle Anwendungen ist es in 3D-Studio-MAX jedoch möglich, Eigenschaften querverbinden, so dass eine Nicht-SP A, die mit einer SP B querverbunden wurde, dennoch indirekt durch eine Simulation gelesen oder verändert werden könnte, wenn die Simulation auf die SP B zugreift. Diese Möglichkeit wird in dieser Arbeit nicht berücksichtigt.

so wird das Präfix „Standard-“ vor dem Datentyp in die Bezeichnung eingefügt, beispielsweise in der Form „Standard-Double-SP“.

Die Werte von SPs können auch Befehle darstellen. Ein Fuß kann die boolesche SP „Walk ON“ enthalten, die bestimmt, ob der Fuß einen Schritt ausführen soll oder nicht. Einige SPs haben mehr den Charakter eines Parameters, der selten verändert wird und oft nur im ersten Frame der Animation vom Benutzer eingestellt wird. So könnte ein Fuß die SP „Max Speed“ enthalten, deren Wert die maximale Bewegungsgeschwindigkeit dieses Fußes grundsätzlich festlegt.

Es ist zu erwähnen, dass die in Java implementierten Objekttypen und Verhaltensweisen der 3D-Objekte auch weitere interne Eigenschaften in Form von Feldern und Variablen haben können, die zwar für das Verhalten der zugehörigen Objekte relevant sind, aber nicht als Objekteigenschaften im 3D-Animationsprogramm vorhanden sind und dort auch nicht als SPs hinzugefügt werden. Solche Eigenschaften werden dann auch nicht als SPs bezeichnet.

Es ist möglich, dass Werte von SPs im aktuellen Simulationsschritt nicht durch die Simulation bestimmt werden sollen, sondern durch manuelle Kontrolle. Dazu wird jeder SP eine boolesche Eigenschaft MP (=Manual Property) zugeordnet, deren Werte die manuelle Kontrolle für bestimmte Zeitintervalle festlegen. Dadurch kann der Benutzer in solchen Zeitintervallen manuell eine Animation für eine SP vornehmen, die für die Simulation bindend ist und von ihr nicht verändert wird. So ist eine im Voraus festgelegte manuelle Steuerung der simulierten Objekte möglich. Die genaue Wirkung der MPs wird in Kapitel 6.6 erläutert, das mit Abbildung 25 auch eine grafische Darstellung der GUI-Komponenten einiger Beispiel-SPs im 3D-Animationsprogramm 3D-Studio-MAX beinhaltet.

6.4 Zwei Datenebenen für die Simulation

Auch wenn die Kommunikation zwischen dem 3D-Animationsprogramm und MaxControl erst in Kapitel 10.3.1 behandelt wird, so ist auch bei der hier angenommenen hochintegrierten Lösung zwischen zwei Ebenen zu unterscheiden, welche jeweils die Szene bzw. Teile davon repräsentieren. Dies gilt sogar in dem denkbaren Fall, dass sowohl das 3D-Animationsprogramm als auch MaxControl in Java implementiert sind und somit eine sehr direkte Integration beider Werkzeuge möglich ist.

Die erste Ebene, auf der eine Szene repräsentiert wird, liegt im 3D-Animationsprogramm. In dieser Ebene werden die für den audiovisuellen⁵³ Teil der Szene wichtigen Eigenschaften der Szene und ihrer 3D-Objekte gespeichert, wie in Kapitel 5.2 und 3.3 beschrieben. Dazu gehören die in Kapitel 6.3 behandelten SPs und Nicht-SPs sowie die in Kapitel 6.6 behandelten MPs. Auch die Animation der animierbaren Objekteigenschaften und der animierbaren globalen Eigenschaften der Szene werden unter Verwendung von Keyframe-Techniken (siehe Kap. 3.5) im 3D-Animationsprogramm gespeichert. Auf dieser Ebene fehlen lediglich die Implementierungen der den 3D-Objekten zugeordneten Objekttypen und ihrer Verhaltensweisen, die in Java entwickelt werden, sowie bei der Simulation entstehende zusätzliche Daten wie die Werte temporärer Variablen im Java Code. Diese Ebene trägt nicht nur einen Zustand sondern eine Folge von Zuständen. Diese „Szenenzustandsfolge“ enthält die Animation der Szene. Jedem Frame der Animation ist ein Zustand aus der Szenenzustandsfolge zugeordnet. Werden also z.B. Daten „aus der Szenenzustandsfolge gelesen“, so werden Daten aus dieser Ebene gelesen. Dabei wird in diesem Text im

⁵³ In der Szene können auch Informationen über virtuelle Tonquellen gespeichert sein, siehe dazu Kap. 5.3 und 10.2.4.

Allgemeinen der Zustand aus der Szenenzustandsfolge, auf den zugegriffen werden soll, explizit angegeben. Ein solcher „Szenenzustand“ kann anhand des zugehörigen Frames referenziert werden. So kann z.B. gesagt werden, dass Daten „aus Frame n der Szenenzustandsfolge“ gelesen werden. Zu einem Frame n gehört der „Szenenzustand n“. Er bietet Zugriff auf alle Daten dieser Ebene für den zugehörigen Frame n, z.B. neben dem Zuständen der 3D-Objekte in Frame n auch ihre geometrischen Eigenschaften und implizit den hierarchischen Aufbau des Szenengraphen.

Die oben eingeführten SPs bilden die Schnittstelle zur zweiten Ebene, die im Simulationsprogramm MaxControl liegt. Auf dieser Ebene werden die 3D-Objekte durch Instanzen zugeordneter Objekttypen (kurz OTs genannt) repräsentiert, die in Java spezifiziert sind. Diese Instanzen werden hier als Simulationsobjekte (kurz SOs) bezeichnet. Jedem OT sind (bis auf Ausnahmen) Verhaltensweisen (kurz VHs) zugeordnet, die dann jedes SO hat, dem dieser OT zugeordnet wurde (siehe Kap. 7.2.1). In den OTs und ihren Verhaltensweisen ist definiert, welche Eigenschaften der 3D-Objekte aus der ersten Ebene (=Szenenzustandsfolge) den entsprechenden SOs dieser Ebene als Standard-SPs zugänglich gemacht werden und welche neuen Eigenschaften die 3D-Objekte auf beiden Ebenen zusätzlich in Form von nicht-standard SPs erhalten (siehe Kap. 6.3). Ebenso sind die zugehörigen MPs auch auf dieser Ebene zu finden. Es wird in den Kapiteln 6-9.5 davon ausgegangen, dass mit der Wahl eines OTs für ein 3D-Objekt auch die nicht-standard SPs sowie die MPs der animierbaren SPs den 3D-Objekten in der Szenenzustandsfolge zusammen mit entsprechenden GUI-Komponenten hinzugefügt werden, so dass diese neu hinzugekommenen Eigenschaften dort sichtbar und manipulierbar sind. Dagegen wird in Kapitel 10.2.1 erläutert, dass dazu ein spezieller Arbeitsschritt erforderlich ist, der vor der Simulation einer Szene durchgeführt sein muss.

Die zweite Ebene wird mit dem Begriff „MaxControl-Zustand“ referenziert. Wie die erste Ebene enthält sie auch die in Kapitel 3.3 erläuterten Informationen über die Szenenhierarchie oder z.B. die Namen der Objekte in der Szene. Sie kann weitere Daten enthalten, wie z.B. die Werte temporärer Variablen im Java Code, die während der Simulation entstehen oder von externen Quellen hinzugezogen werden, da der Java-Code auch auf Dateien zugreifen und sogar mit dem Internet kommunizieren kann. Werden also z.B. Daten „in den MaxControl-Zustand geschrieben“, so werden Daten in diese Ebene transferiert. Im Gegensatz zur Szenenzustandsfolge im 3D-Animationsprogramm speichert MaxControl in der Regel nur den aktuellen Zustand der Simulation, sofern die Simulationsobjekte oder ihre Verhaltensweisen nicht zusätzlich Daten über vorangehende Zustände der Simulation speichern, was in Einzelfällen sinnvoll sein kann.

Alle SPs werden in beiden Ebenen gespeichert, sie haben eine Repräsentation sowohl in der Szenenzustandsfolge als auch im MaxControl-Zustand. Auch die zugehörigen MPs sind auf beiden Ebenen gespeichert. Aus technischen Gründen können die Werte der SPs auf beiden Ebenen unterschiedlich sein, insbesondere, wenn es sich um Fließkommawerte handelt, da das verwendete 3D-Animationsprogramm solche Werte möglicherweise mit einer anderen Genauigkeit speichert als MaxControl. Darauf wird in Kapitel 10.3.1.2 näher eingegangen.

6.5 Zustandsübergänge im Verlauf der Simulation

Zur genauen Erklärung der Abläufe im Simulationszyklus werden die Zustandsübergänge während der Simulation und beim Datenaustausch zwischen der Szenenzustandsfolge und dem MaxControl-Zustand erläutert. Hierbei wird zunächst die manuelle Kontrolle von SPs nicht beachtet, um die grundsätzlichen Abläufe klarer darstellen zu können.

Damit eine Simulation mit MaxControl gestartet werden kann, muss eine 3D-Szene als Szenengraph im verwendeten 3D-Animationsprogramm vorliegen. Den 3D-Objekten müssen Objekttypen und die zugehörigen SPs zugeordnet worden sein. Der Objekttyp ist für die Simulation erforderlich, weil er festlegt, welche Verhaltensweisen den 3D-Objekten zugeordnet werden. Die SPs müssen den 3D-Objekten zugeordnet worden sein, weil die Simulation mit diesen Eigenschaften arbeitet, indem sie gelesen und verändert werden. Die Änderungen an den Werten der SPs im Verlauf der Simulation repräsentieren die automatisch entstehende Animation. Diese Änderungen müssen daher in den SPs der 3D-Objekte gespeichert werden können, damit die so entstandenen Werteverläufe nach der Simulation in der Szene des 3D-Animationsprogramms als Animation verfügbar sind.

Neben dem Szenengraph muss auch das Simulationsintervall mit seiner Unterteilung in k Teilintervalle festgelegt sein. Es geht von Frame a bis Frame b . Damit enthält dieses Intervall $(b-a+1)$ Frames und erfordert $k=(b-a)$ Simulationsschritte, um Zustände für alle Frames von $a+1$ bis b zu erzeugen.

Im verwendeten 3D-Animationsprogramm ist schon vor der Simulation immer eine vorläufige Szenenzustandsfolge $(\delta_a, \dots, \delta_b)$ gespeichert. Jeder Szenenzustand δ_n ist eine Belegung der Eigenschaften aller 3D-Objekte und aller globalen Eigenschaften der Szene mit Werten, wobei jeder Szenenzustand δ_n total definiert ist. Die Objekteigenschaften umfassen dabei Nicht-SPs, Standard-SPs und die neu hinzugekommenen nicht-standard SPs. Ein Szenenzustand δ_n ist der Zustand der Szene zu Frame n .

Die vorläufige Szenenzustandsfolge $(\delta_a, \dots, \delta_b)$ trägt bereits die bezüglich der Simulation endgültigen Werteverläufe für alle Nicht-SPs, da diese durch die Simulation nicht verändert werden. Bezüglich der Nicht-SPs können 3D-Objekte vor oder nach der Simulation auf herkömmliche Weise animiert werden, ohne dass sie durch die Simulation manipuliert oder von ihr berücksichtigt werden.

Die vorläufige Szenenzustandsfolge kann manuell erstellt oder durch eine vorangehende Simulation erzeugt worden sein. Eine Mischung beider Methoden kann ebenfalls vorliegen, sogar für eine einzelne SP können Werteverläufe in einem Intervall A der Animation durch eine Simulation erzeugt und gleichzeitig in einem anderen Intervall B manuell erstellt worden sein.

Sowohl manuelle Erstellung als auch Simulation benutzen bei der Festlegung der vorläufigen Szenenzustandsfolge Key-Techniken. Damit ist der Werteverlauf einer Objekteigenschaft in den durch Keys bestimmten Intervallen durch die gewählte Interpolationsart definiert. Außerhalb dieser Intervalle verwendet MaxControl die Fortsetzungsart „Constant“ aus 3D-Studio-MAX (siehe Kapitel 3.5). Falls für eine Objekteigenschaft kein Key erzeugt wurde, wird davon Gebrauch gemacht, dass in 3D-Studio-MAX einer Eigenschaft auch ohne Key ein Wert zugewiesen werden kann, der dann im gesamten Animationsintervall konstant auf dem so gesetzten Wert bleibt. Dies gilt insbesondere für nicht-animierbare Eigenschaften, für die keine Keys festgelegt werden können. Wenn der Benutzer einer Eigenschaft keinen Wert zuweist, so ist dieser Eigenschaft immer ein Vorgabewert zugewiesen, so dass keine Eigenschaft undefiniert⁵⁴ sein kann. Deshalb ist eine vorläufige Szenenzustandsfolge

⁵⁴ Eine Ausnahme sind in 3D-Studio-MAX Eigenschaften, die Referenzen auf andere Objekte speichern können. Diese können den Wert „undefined“ tragen, der mit dem Java-Wert „NULL“ vergleichbar ist und angibt, dass diese Variable kein Objekt referenziert. Damit ist jedoch auch dieser „Nullwert“ in der Eigenschaft gespeichert und kann ohne Fehlermeldung gelesen werden, so dass die Eigenschaft auch in diesem Fall nicht undefiniert (=nicht mit einem Wert belegt) ist.

$(\delta_a, \dots, \delta_b)$ immer vollständig definiert. Da 3D-Studio-MAX erst in der technischen Umsetzung verwendet wird, wird vorausgesetzt, dass auch das hier angenommene 3D-Animationswerkzeug (siehe Kap. 6.1) dieses Verhalten bezüglich der Wertebelegungen für Eigenschaften zeigt.

Durch die Simulation soll die vorläufige Szenenzustandsfolge in die endgültige Szenenzustandsfolge $(\delta'_a, \dots, \delta'_b)$ umgewandelt werden, indem neue Werteverläufe für die SPs erzeugt werden. Da Frame a der Anfangszustand der Simulation ist, wird festgelegt:

Definition 1:

$$\delta'_a \stackrel{df}{=} \delta_a$$

Die eigentliche Simulation wird in der Ebene der MaxControl-Zustände durchgeführt. Durch jeden Simulationsschritt zu einem Frame n wird ausgehend von einem MaxControl-Zustand σ_{n-1} ein neuer MaxControl-Zustand σ_n erzeugt. Die Werte aus diesem neuen MaxControl-Zustand werden nach dem Simulationsschritt auf den zu Frame n gehörenden Szenenzustand δ_n übertragen, wodurch der Zustand δ'_n entsteht. Die Nicht-SPs aus δ'_n fehlen in σ_n , weshalb ihre Werte bei diesem Übertragungsvorgang auch nicht verändert werden können und somit aus δ_n unverändert übernommen werden.

Ein MaxControl-Zustand σ_n kann damit für $n > a$ als Beschränkung von δ'_n auf die Menge S der SPs gesehen werden. Für den Start der Simulation legen wir fest:

Definition 2:

$$\sigma_a \stackrel{df}{=} \delta_a \Big|_S$$

Damit gilt nach Definition 1 auch $\sigma_a = \delta'_a \Big|_S$.

Vor dem Start der Simulation liegen die in der folgenden Abbildung 23 gezeigten Zustände σ_a und δ'_a sowie die vorläufige Szenenzustandsfolge $(\delta_a, \dots, \delta_b)$ vor. Die Zustände σ_a und δ_a sind lediglich auf der Ebene der SPs identisch, weshalb das in Abbildung 23 zwischen diesen Zuständen gezeigte senkrechte Gleichheitssymbol mit einem Stern-Symbol (*) versehen ist. Dagegen sind die Zustände δ_a und δ'_a nach Definition 1 vollkommen identisch. Die vorläufige Szenenzustandsfolge $(\delta_a, \dots, \delta_b)$ trägt bereits die endgültigen Werteverläufe für alle Nicht-SPs. Sie wird durch die Simulation schrittweise in eine neue Szenenzustandsfolge $(\delta'_a, \dots, \delta'_b)$ umgewandelt. Diese neue Szenenzustandsfolge repräsentiert dann die Animation, die in einen Film umgewandelt werden kann.

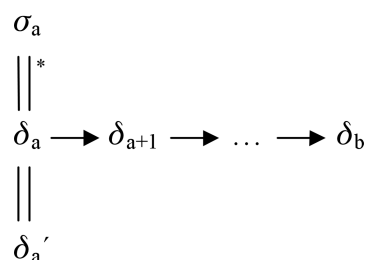


Abbildung 23: Diagramm zur Anfangssituation vor der Simulation

Wie oben bereits erwähnt startet MaxControl jeden Simulationsschritt mit einem MaxControl-Zustand σ_{n-1} und erzeugt durch den Simulationsschritt den Folgezustand σ_n . Dabei startet der erste Simulationsschritt mit σ_a . Durch Simulationszwischenritte, die zur Erhöhung der Genauigkeit von Näherungsverfahren in Verhaltensweisen verwendet werden (siehe auch Kap. 6.2 ab S. 78), entsteht dabei für den Übergang von σ_{n-1} nach σ_n eine Folge von Zwischenzuständen:

Definition 3:

$$\left(\begin{array}{l} \text{Für } a < n \leq b : \\ \sigma_{n-1} = \sigma_{n-1,0}, \sigma_{n-1,1}, \sigma_{n-1,2}, \dots, \sigma_{n-1,m(n)} = \sigma_n \end{array} \right)$$

Dabei ist m von n abhängig, da in jedem Simulationsschritt eine andere Anzahl von Simulationszwischenritten durchgeführt werden kann (siehe Kap. 10.2.2).

Oben wurde die Bezeichnung S für die Menge aller SPs eingeführt. Sei N die Menge aller Nicht-SPs, also aller Eigenschaften der 3D-Objekte aus der Szenenzustandsfolge, die weder als Standard-SPs dem Simulationssystem MaxControl zugänglich gemacht wurden noch als nicht-standard SPs den 3D-Objekten neu hinzugefügt wurden. Somit gilt $S \cap N = \emptyset$. Wie in Kapitel 6.2 erwähnt, werden die Werte der Nicht-SPs durch die Simulation niemals verändert.

Für die folgenden Definitionen sei $\sigma(q)$ der Wert einer Eigenschaft $q \in S$ in einem MaxControl-Zustand σ , analog sei der Wert einer Eigenschaft $q \in S \cup N$ in einem Szenenzustand δ oder δ' als $\delta(q)$ bzw. $\delta'(q)$ bezeichnet. Ist $q \in N$, so existiert diese Nicht-SP nur auf der Ebene der Szenenzustandsfolge. Die Menge N der Nicht-SPs ist also nicht Teil des Definitionsbereiches eines MaxControl-Zustandes σ .

Der Simulationsschritt zu Frame n wird im Zwischenzustand $\sigma_{n-1,0}$ gestartet, der nach Definition 3 dem Zustand σ_{n-1} entspricht, d.h.:

Definition 4:

$$\left(\begin{array}{l} \text{Für } q \in S \text{ und } a < n \leq b : \\ \sigma_{n-1,0}(q) =_{df} \sigma_{n-1}(q) \end{array} \right)$$

Der Folgezustand, der aus einem MaxControl-Zustand σ durch einen Simulationszwischenritt entsteht, sei als sim_σ bezeichnet. Die Simulationsfunktion, welche diesen Zustand erzeugt, sei als „sim“ bezeichnet, so dass $\text{sim}_\sigma = \text{sim}(\sigma)$ gilt. Diese Funktion ändert einen vorliegenden MaxControl-Zustand σ , indem sie auf diesem Zustand alle Verhaltensweisen aller Simulationsobjekte in einer festen Reihenfolge gemäß dem Gauß-Seidel⁵⁵-Prinzip (Einzelschrittverfahren) genau einmal ausführt und dadurch den MaxControl-Zustand sim_σ erzeugt. SP-Werte, auf die keine VH schreibend zugreift, werden unverändert von σ nach sim_σ übertragen. Die genaue Arbeitsweise der Funktion „sim“ wird in Kapitel 9 erläutert. Der Übergang von einem Zwischenzustand $\sigma_{n-1,j}$ zum folgenden Zwischenzustand $\sigma_{n-1,j+1}$ ist damit definiert als:

⁵⁵ Dies bedeutet, dass innerhalb eines Simulationszwischenrittes jede von „sim“ ausgeführte Änderung eines SP-Wertes sofort gültig ist und beim anschließenden Ausführen der restlichen VHs berücksichtigt wird. Dieses Vorgehen ist mit dem Gauß-Seidel-Prinzip [Stoer73] vergleichbar (siehe Kap. 9.1).

Definition 5:

$$\text{Für } q \in S \text{ und } a < n \leq b : \\ \sigma_{n-1,j+1}(q) =_{df} \text{sim}_{\sigma_{n-1,j}}(q), \text{ für } j = 0, \dots, m(n) - 1$$

Da der letzte so erzeugte Zwischenzustand $\sigma_{n-1,m(n)}$ nach Definition 3 dem Zustand σ_n entspricht, kann jetzt der neue Szenenzustand festgelegt werden durch:

Definition 6:

$$\text{Für } q \in S \cup N \text{ und } a < n \leq b : \\ \delta'_n(q) =_{df} \begin{cases} \delta_n(q), & \text{falls } q \in N \\ \sigma_n(q), & \text{sonst} \end{cases}$$

Der Simulationsschritt zu einem Frame n mit $a < n \leq b$, d.h., die Festlegung des endgültigen Szenenzustandes δ'_n bei gegebener endgültiger Teilfolge $(\delta'_a, \dots, \delta'_{n-1})$, gegebenem MaxControl-Zustand $\sigma_{n-1} = \delta_{n-1} \upharpoonright_S$ und gegebener vorläufiger Teilfolge $(\delta_n, \dots, \delta_b)$, wird in der folgenden Abbildung 24 dargestellt:

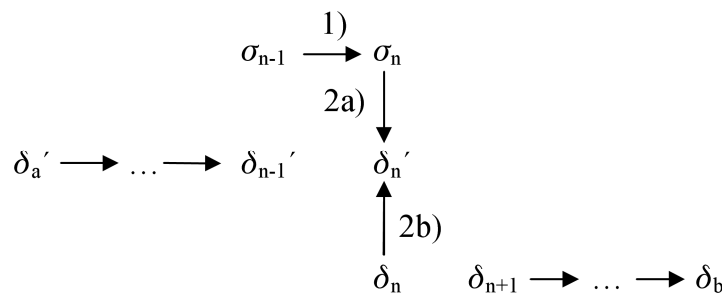


Abbildung 24: Diagramm zu den Zustandsübergängen während der Simulation

Der Simulationsschritt setzt sich folgendermaßen zusammen:

- 1) Der MaxControl-Zustand σ_{n-1} wird in den Folgezustand σ_n überführt, indem die Simulationsfunktion $m(n)$ mal hintereinander ausgeführt wird. Es gilt also:

$$\sigma_n = \text{sim}^{m(n)}(\sigma_{n-1}).$$

- 2a) Die Werte der SPs aus dem neuen MaxControl-Zustand σ_n werden in den neuen Szenenzustand δ'_n übertragen.

- 2b) Die Werte der Nicht-SPs werden aus δ_n nach δ'_n übertragen. Der Szenenzustand δ_n ist nun überflüssig geworden. In der Praxis bedeutet dies, dass er durch δ'_n überschrieben wird. Auch dieser neue Szenenzustand δ'_n gehört wie δ_n zu Frame n .

Das Diagramm in Abbildung 24 zeigt, dass vor dem Simulationsschritt die Szenenzustandsfolge $(\delta'_a, \dots, \delta'_{n-1}, \delta_n, \delta_{n+1}, \dots, \delta_b)$ vorliegt, während sie danach die Form $(\delta'_a, \dots, \delta'_{n-1}, \delta'_n, \delta_{n+1}, \dots, \delta_b)$ hat. Die vorläufige Szenenzustandsfolge $(\delta_a, \dots, \delta_b)$ wird also schrittweise in die endgültige Szenenzustandsfolge $(\delta'_a, \dots, \delta'_b)$ umgewandelt.

Im Folgenden wird auf einige Details und Besonderheiten hingewiesen, die bezüglich der Zustände und der Übergänge zwischen ihnen zu beachten sind.

Das Animationsintervall und damit auch das Simulationsintervall sind auch nach einer Animation frei wählbar. Dabei wird ausgenutzt, dass die Werteverläufe von SPs und Nicht-SPs stets vollständig definiert sind, durch die Verwendung der Fortsetzungsart „Constant“ sogar vor und nach dem letzten Key einer Eigenschaft. Somit sind die Werteverläufe auch außerhalb des Animationsintervalls definiert. Auch Keys können außerhalb dieses Intervalls liegen und dort Werteverläufe festlegen. Eine Verschiebung von Anfangs- und Endpunkt des Animationsintervalls durch Änderung der Parameter a und b nach einer Simulation verschiebt lediglich den Bereich der Animation, der aktuell betrachtet wird, oder ändert die Größe dieses Bereichs. Bei solch einer Verschiebung bleiben alle Werteverläufe unverändert, auch die Keys behalten ihre Position in dem jeweiligen Frame, dem sie zugeordnet wurden.

Die MaxControl-Zustände σ_n werden hier nur bezüglich der SPs betrachtet, interne Variablen oder externe Datenquellen (siehe Kap. 6.4) bleiben unbeachtet. Die möglichen Wirkungen solcher Variablen und Datenquellen sind in die oben genannte Simulationsfunktion „sim“ integriert.

Jeder Simulationsschritt zu einem Frame n simuliert ein bestimmtes Zeitintervall $T(n)$. Um Zeitlupen- und Zeitraffereffekte zu ermöglichen, kann jeder Simulationsschritt eine eigene Zeitintervalllänge erhalten (siehe Erläuterungen zur Methode „getDeltaT“ in Kap. 8.1.2 sowie Kap. 10.2.2). Wenn m Simulationszwischenritte für einen Simulationsschritt vorgesehen sind, wird das Zeitintervall $T(n)$ des gesamten Simulationsschrittes in m gleichlange Teilintervalle der Länge $\Delta t(n) = \frac{T(n)}{m}$ eingeteilt.

Die in Kapitel 6.4 erläuterte Besonderheit, dass sich die Werte einer SP auf der Ebene der MaxControl-Zustände und auf der Ebene der Szenenzustandsfolge aus technischen Gründen unterscheiden können, wird hier nicht betrachtet, so dass bei den hier eingeführten Definitionen von einer Gleichheit solcher Werte auf beiden Ebenen ausgegangen wird. Ein

Beispiel hierfür ist die Definition 2 mit $\sigma_a =_{df} \delta_a|_S$.

6.6 Manuelle Kontrolle von Simulationseigenschaften

Wie in Kapitel 6.3 erwähnt, kann eine Simulation in MaxControl im Verlauf der Animation auch manuell gesteuert werden, indem man die Werte **animierbarer** SPs in jeweils wählbaren Zeitintervallen der Animation als **manuell kontrolliert** gelten lässt. Das bedeutet, dass die Simulation die Werte einer SP in einem solchen Intervall nicht ändern kann, sondern die in der vorläufigen Szenenzustandsfolge vorhandenen Werte und Werteverläufe für diese SP verwendet. Diese Werteverläufe können vor der Simulation mit herkömmlicher Keyframe-Animation manuell erstellt werden.

Allen **animierbaren** SPs ist deshalb, jeweils eine zusätzliche boolesche Eigenschaft MP („Manual Property“) zugeordnet. Diese MP ist auch in der grafischen Benutzeroberfläche des 3D-Animationsprogrammes sichtbar und wird ausschließlich vom Benutzer eingestellt. Die booleschen Werte der MPs bestimmen in jedem Frame der Animation, ob der Wert der zugehörigen SP in diesem Frame als manuell kontrolliert gelten soll oder nicht. Wie die animierbaren SPs selbst sind auch ihre MPs animierbar. Die Werteverläufe der MPs im Animationsintervall können mit Keyframe-Techniken erstellt werden.

In der technischen Umsetzung haben auch nicht-animierbare SPs jeweils eine zugehörige MP mit einem vordefinierten Werteverlauf, was bis einschließlich Kapitel 9.5 jedoch nicht

berücksichtigt wird. Auf diese technische Besonderheit wird in Kapitel 10.3.1.1 ab Seite 206 näher eingegangen.

Animationen von SPs in einer vorläufigen Szenenzustandsfolge können zwar manuell in dem 3D-Animationsprogramm festgelegt worden sein, sie können aber auch von einer vorangehenden Simulation der Szene stammen. MaxControl kann diese Fälle nicht unterscheiden. Ob eine SP in einem Frame also wirklich als manuell kontrolliert gelten soll, wird deshalb erst durch die Werte der zugehörigen MP festgelegt. Nur wenn der Wert der zugehörigen MP in einem Frame „true“ ist, gilt auch der Wert der SP in diesem Frame als manuell kontrolliert und wird in der Simulation entsprechend berücksichtigt.

In Abbildung 25 folgt eine grafische Darstellung der GUI-Komponenten einiger Beispiel-SPs und -MPs im 3D-Animationsprogramm 3D-Studio-MAX.

Es folgt eine Diskussion der genauen Wirkungsweise der manuellen Kontrolle. Der Wert einer im aktuellen Frame n manuell kontrollierten SP soll **konstant** für das gesamte zu simulierende Intervall gelten, das direkt nach dem vorangehenden Frame $n-1$ beginnt und einschließlich bis zum aktuellen Frame n geht. Dieses links offene Intervall wird im Folgenden auch als Simulationsintervall zu Frame n bezeichnet. In dem Intervall können aufgrund von Simulationszwischenritten Zwischenzustände vorliegen (siehe Kap. 6.2 und 6.5). Abbildung 26 macht diese Situation anschaulich.

Für die Zustandsübergänge bedeutet dies: Ist der boolesche Wert einer MP im aktuellen Frame n „true“, so wird der Wert der zugehörigen SP aus dem vorläufigen Szenenzustand δ_n gelesen, bevor der Simulationsschritt zur Bestimmung von δ_n' beginnt. Die Gültigkeit des aus δ_n gelesenen Wertes wird für den gesamten Simulationsschritt erzwungen. Ist der boolesche Wert einer MP in Frame n dagegen „false“, so wird der Wert der zugehörigen SP nicht aus δ_n gelesen. Stattdessen wird nach dem Simulationsschritt der neue (durch Simulation entstandene) Wert der SP in den endgültigen Szenenzustand δ_n' übertragen.

Der Wert einer MP bestimmt also für den Simulationsschritt zu einem Frame n , ob der in δ_n bereits gültige Wert der zugehörigen SP für jeden Simulationszwischenritt verwendet werden soll oder nicht. Ist der Wert der MP in Frame n „true“, kann die Simulation im aktuellen Simulationsschritt den Wert dieser SP nicht verändern.

So kann beispielsweise die Lenkbewegung eines automatisch gesteuerten Fahrzeugs in frei bestimmbaren Intervallen der Animation manuell festgelegt werden, indem man in den entsprechenden Frames die SP für den Einschlagwinkel der Lenkung manuell einstellt und die zugehörige MP in diesen Frames auf den Wert „true“ setzt. Das Fahrzeug wird dann in diesen Intervallen der Animation seine Räder entsprechend den manuell eingestellten Werten der SP einschlagen, auch wenn seine automatische Steuerung einen anderen Einschlagwinkel bestimmt hat.

Es wirkt zunächst überraschend, dass Werte, die in einem Frame n als manuell kontrolliert vorgegeben werden, schon in dem gesamten davor liegenden (links offenen) Intervall gültig sein sollen. Die Gültigkeit eines manuell kontrollierten SP-Wertes wurde so festgelegt, damit auch **indirekte** Auswirkungen einer manuell kontrollierten Einstellung bereits in dem Frame sichtbar werden können, in dem diese Einstellung gemacht wurde.

Ein Beispiel zur Verdeutlichung sei eine Lichtquelle des Typs `MCD_Light` (siehe Abbildung 29 auf Seite 96). Diese kann über die Werte ihrer booleschen SP „lightOn“ an- und ausgeschaltet werden, weil ihre Verhaltensweise `MCB_LightSwitch` bei der Simulation die

Helligkeit der Lichtquelle, die in der Standard-SP „lightMultiplier“ gespeichert ist, entsprechend höher (z.B. auf den Wert 1) oder niedriger (z.B. auf den Wert 0) einstellt.

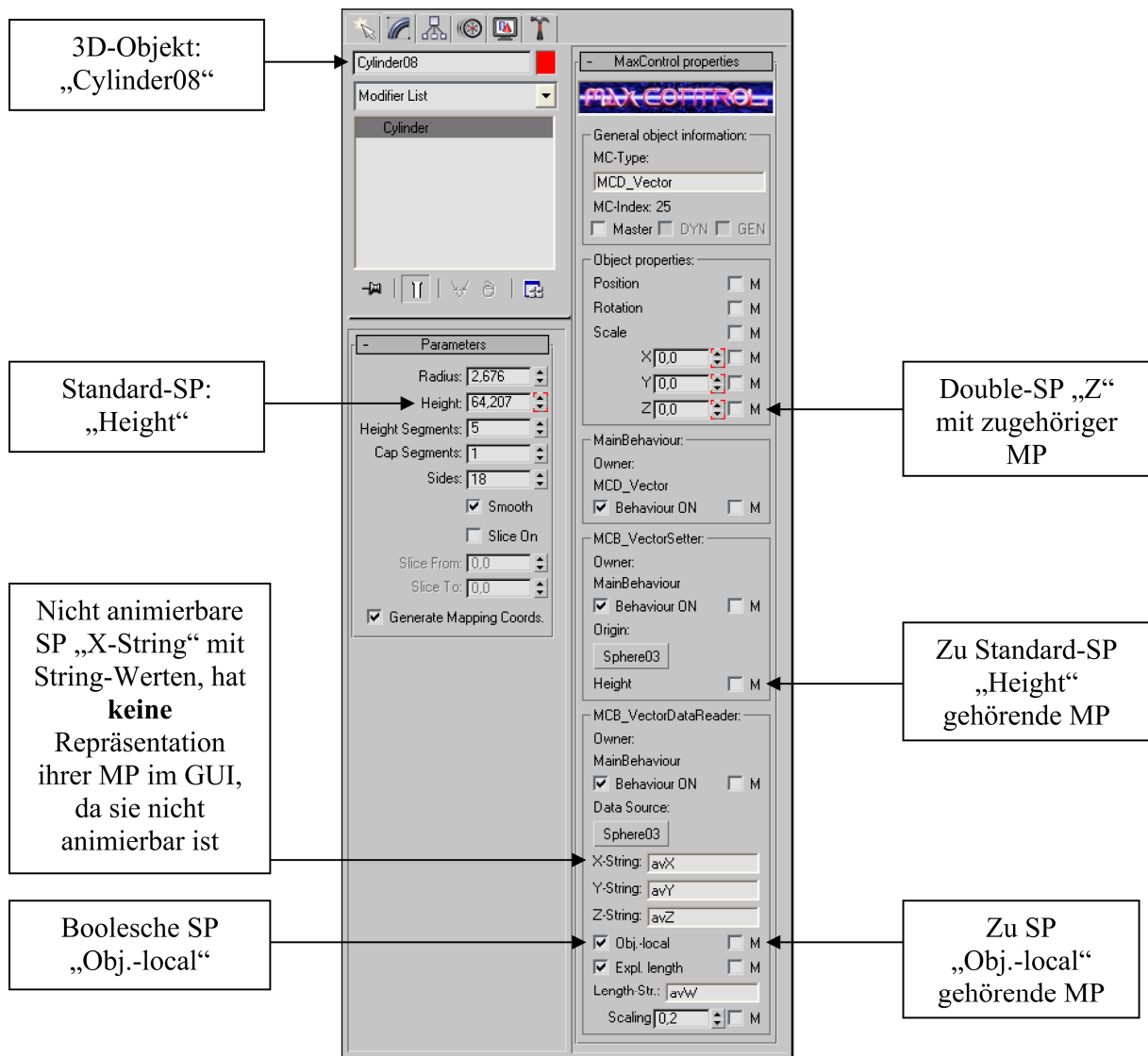


Abbildung 25: GUI-Repräsentation einiger SPs

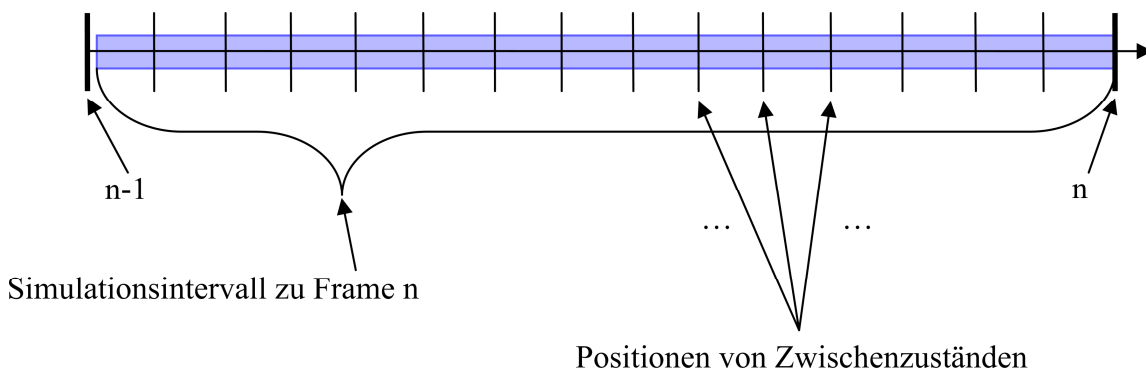


Abbildung 26: Simulationsintervall zu Frame n

Zur Erläuterung folgt hier in vereinfachter Form das entsprechende Codefragment aus der Verhaltensweise „MCB_LightSwitch“, das in jedem Simulationszwischen Schritt genau einmal für jede Lichtquelle des Typs „MCD_Light“ in der Szene ausgeführt wird. Dabei trage für dieses Beispiel die Variable multOn den Wert 1 und die Variable multOff den Wert 0:

```

if (lightOn)
{
  lightMultiplier=multOn;
}
else
{
  lightMultiplier=multOff;
}

```

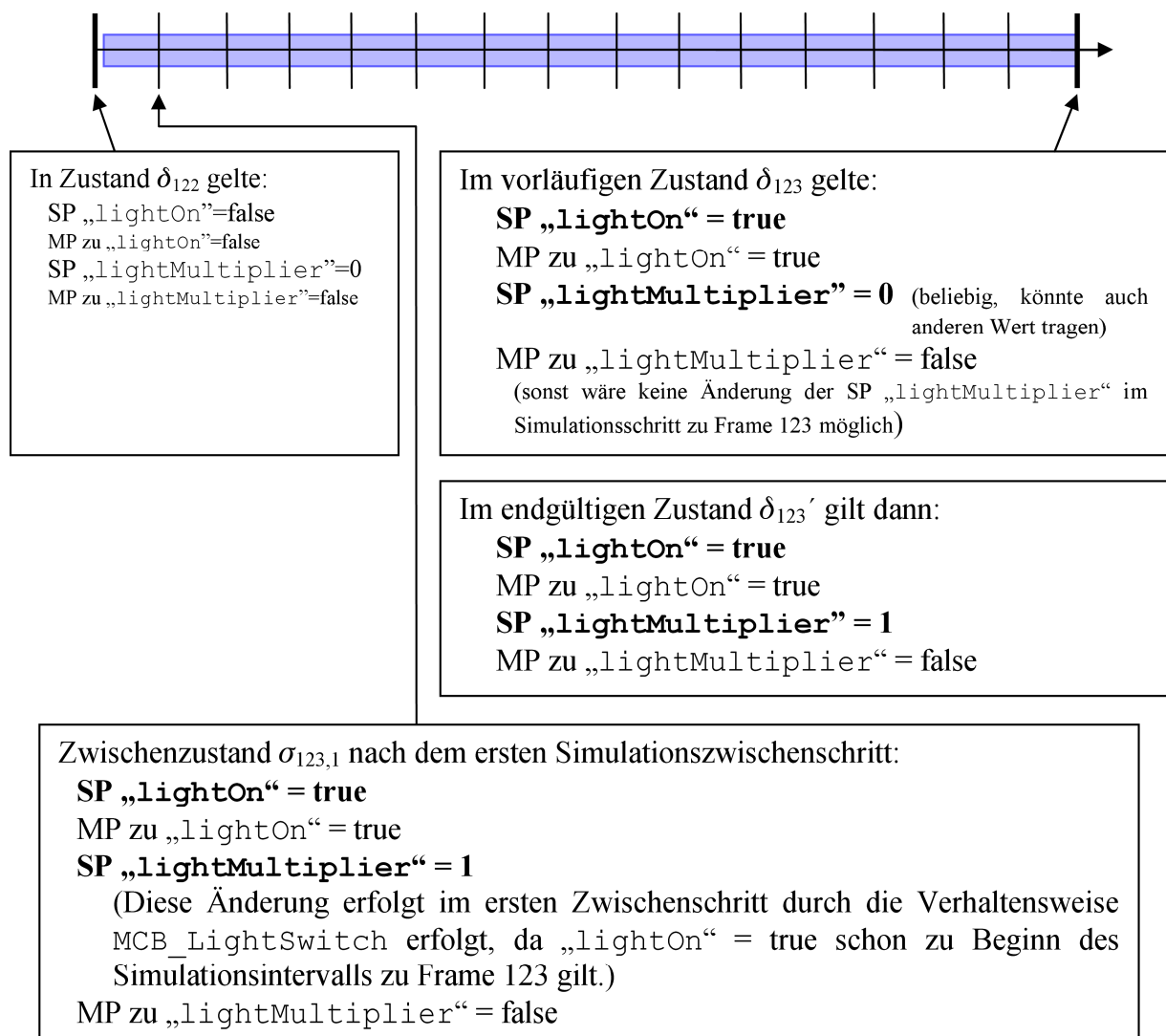


Abbildung 27: Zeitlicher Wirkungsbereich der manuellen Kontrolle

Dies zeigt, dass eine durch diese VH bewirkte Änderung des Wertes von „lightMultiplier“ nur als **Folge** einer entsprechenden Einstellung von „lightOn“ geschehen kann. Denn nach einer Änderung der SP „lightOn“ muss mindestens ein Simulationszwischen Schritt durchgeführt werden, damit die Verhaltensweise „MCB_LightSwitch“ wenigstens einmal ausgeführt wird und den Inhalt der SP

„lightMultiplier“ entsprechend ändern kann. Wie Verhaltensweisen genau bei der Simulation eingesetzt werden wird in den Kapiteln 8.6 und 9 erläutert.

Will man diese Lichtquelle nun manuell kontrolliert in Frame 123 leuchten lassen, so könnte man die SP „lightOn“ und die zugehörige MP in Frame 123 auf „true“ stellen. Würde der Wert true für die SP „lightOn“ erst ab Frame 123 gelten, so wäre beginnend bei Frame 123 mindestens ein Simulationszwischen Schritt nötig, damit auch die SP „lightMultiplier“ durch die Verhaltensweise MCB_LightSwitch den Wert 1 erhält. Das Ergebnis der simulierten Verhaltensweise der Lampe würde erst in Frame 124 sichtbar werden, nämlich dass die Lampe nun leuchtet. Man wird aber erwarten, dass die Lampe bereits in Frame 123 leuchtet, da sie hier auch eingeschaltet wurde. Daher wurde der Gültigkeitsbereich einer manuell kontrollierten SP-Einstellung wie oben beschrieben gewählt. Die daraus resultierende beabsichtigte Auswirkung auf die Simulation dieses Beispiels wird in Abbildung 27 verdeutlicht.

6.7 Zustandsübergänge unter Berücksichtigung der manuellen Kontrolle

Dieses Kapitel setzt die Erläuterungen zu den Zustandsübergängen während der Simulation aus Kapitel 6.5 voraus und verwendet die dort eingeführten Definitionen, Folgerungen und Mengenbezeichnungen. Bei den nun folgenden Betrachtungen werden auch die MPs berücksichtigt, die den animierbaren SPs sowohl im MaxControl-Zustand als auch in der Szenenzustandsfolge zugeordnet sind. In der vorläufigen Szenenzustandsfolge $(\delta_a, \dots, \delta_b)$ sind auch die booleschen Werteverläufe der MPs festgelegt. Die SPs tragen in den Intervallen, in denen ihre zugehörigen MP-Werte „true“ sind, die Werteverläufe, welche sie auch in der endgültigen Animation haben sollen.

Die Menge der in Frame n manuell kontrollierten SPs, deren MPs in diesem Frame der Szenenzustandsfolge also den Wert „true“ haben, sei als $M(n)$ bezeichnet. Es gilt $M(n) \subseteq S$ und damit $M(n) \cap N = \emptyset$, da nur (animierbare) SPs eine MP besitzen.

Der Simulationsschritt zu Frame n wird im Zwischenzustand $\sigma_{n-1,0}$ gestartet, der aus σ_{n-1} hervorgeht, indem $\sigma_{n-1,0}(q)$ für $q \in M(n)$ den Wert $\delta_n(q)$ erhält und für alle anderen q die jeweiligen Werte aus σ_{n-1} übernommen werden. Die Definition 4 aus Kapitel 6.5 wird entsprechend durch die folgende Definition ersetzt:

Definition 7:

| |
|--|
| <p>Für $q \in S$ und $a < n \leq b$:</p> $\sigma_{n-1,0}(q) =_{df} \begin{cases} \delta_n(q), & \text{falls } q \in M(n) \\ \sigma_{n-1}(q), & \text{sonst} \end{cases}$ |
|--|

Der Übergang von einem Zwischenzustand $\sigma_{n-1,j}$ zum folgenden Zwischenzustand $\sigma_{n-1,j+1}$ wird unter Berücksichtigung der MPs folgendermaßen neu definiert und ersetzt Definition 5 aus Kapitel 6.5:

Definition 8:

| |
|--|
| <p>Für $q \in S$ und $a < n \leq b$:</p> $\sigma_{n-1,j+1}(q) =_{df} \begin{cases} \sigma_{n-1,j}(q), & \text{falls } q \in M(n) \\ \text{sim}_{\sigma_{n-1,j}}(q), & \text{sonst} \end{cases} \quad \text{für } j = 0, \dots, m(n) - 1$ |
|--|

Dabei folgt aus Definition 7 und Definition 8, dass **für $q \in M(n)$** und für $j=0, \dots, m(n)$ die Gleichheit $\sigma_{n-1,j}(q) = \delta_n(q)$ gilt, da dieser Wert nach Definition 8 für jeden Zwischenzustand $\sigma_{n-1,j}$ mit $j>0$ aus dem vorangehenden Zwischenzustand $\sigma_{n-1,j-1}$ unverändert übernommen wird. Diese Kette beginnt bei Zustand $\sigma_{n-1,0}$, in den nach Definition 7 der ursprüngliche Wert $\delta_n(q)$ übertragen wird und endet bei $\sigma_{n-1,m(n)} = \sigma_n$. Für jede in Frame n manuell kontrollierte SP q wird also in jedem Zwischenzustand $\sigma_{n-1,j}$ der Wert $\delta_n(q)$ festgehalten, wie es auch im vorangehenden Kapitel 6.6 gefordert wurde. Der Wert einer solchen manuell kontrollierten SP q wird auch beim Übergang zu Zustand δ_n' nicht geändert, wie die Definition 6 aus Kapitel 6.5 zeigt, welche hier noch einmal wiederholt wird:

Definition 6:

| |
|--|
| Für $q \in S \cup N$ und $a < n \leq b$: $\delta_n'(q) =_{df} \begin{cases} \delta_n(q), & \text{falls } q \in N \\ \sigma_n(q), & \text{sonst} \end{cases}$ |
|--|

Dabei gilt nach den Erläuterungen zu Definition 8, dass **für $q \in M(n)$** $\sigma_{n-1,m(n)}(q) = \delta_n(q)$ gilt. Daraus folgt mit $\delta_n'(q) = \sigma_n(q)$ ⁵⁶ = $\sigma_{n-1,m(n)}(q)$ ⁵⁷, dass $\delta_n'(q) = \delta_n(q)$ gilt, so dass auch bei diesem Übergang der Wert der manuell kontrollierten SP erhalten bleibt. Zusammengefasst gilt also:

①:

| |
|---|
| Für $q \in M(n)$ und $a < n \leq b$: $\delta_n'(q) = \delta_n(q)$ |
|---|

Auch beim Schreiben der SP-Werte aus σ_n in δ_n' bleiben die in Frame n manuell kontrollierten SP-Werte aus δ_n also erhalten, ebenso bleiben die Werte der Nicht-SPs unverändert.

Die Festlegung eines Zustandes δ_n' in der endgültigen Szenenzustandsfolge auf der Basis von δ_n , σ_{n-1} und eines Simulationsschrittes in MaxControl wird im Folgenden schematisch dargestellt:

⁵⁶ Folgt aus Definition 6 und $q \in M(n) \Rightarrow q \notin N$ (wegen $M(n) \cap N = \emptyset$ (s.o.)).

⁵⁷ Siehe Definition 3 aus Kapitel 6.5.

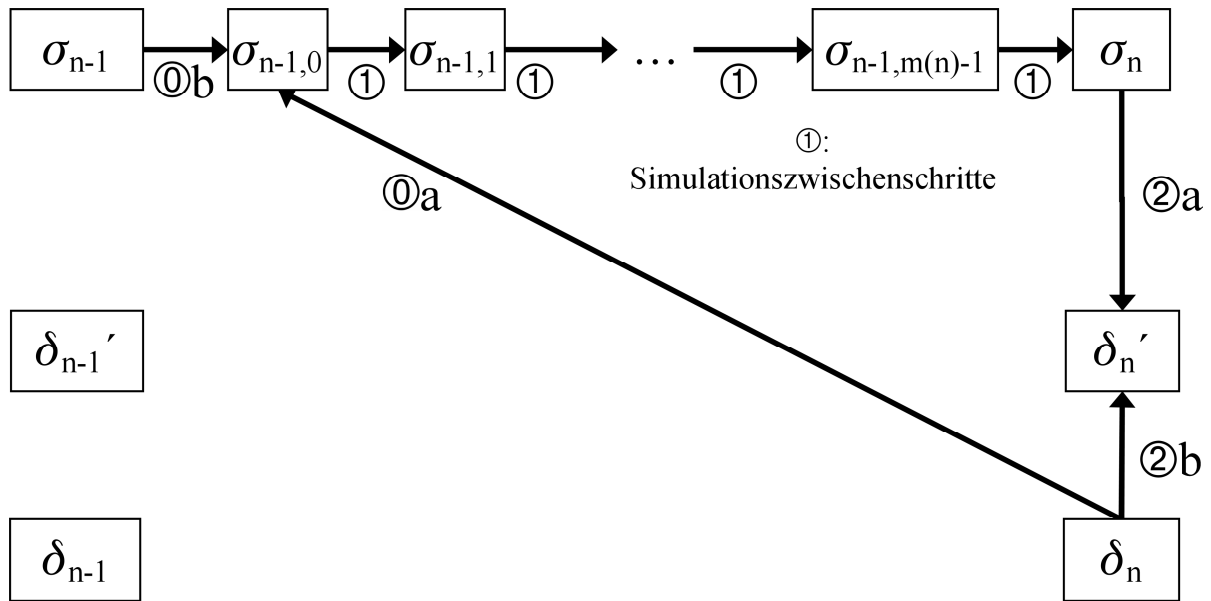


Abbildung 28: Zustandsübergänge unter Berücksichtigung der MP

Es werden die folgenden Schritte in der angegebenen Reihenfolge durchgeführt, die auch im Diagramm an den Pfeilen entsprechend gekennzeichnet sind:

- ③a: Die Werte der in Frame n manuell kontrollierten SPs werden aus δ_n in den Zustand $\sigma_{n-1,0}$ übertragen.
- ③b: Hier werden die Werte aller SPs, die in Frame n **nicht** manuell kontrolliert sind, aus Zustand σ_{n-1} in den Zwischenzustand $\sigma_{n-1,0}$ übertragen.
- ①: Hier werden die Zustandsübergänge von $\sigma_{n-1,0}$ bis $\sigma_{n-1,m(n)} = \sigma_n$ in mehreren Simulationszwischenschritten berechnet, wobei die aus δ_n gelesenen manuell kontrollierten Werte festgehalten werden.
- ②a: Die Werte der SPs aus dem neuen MaxControl-Zustand σ_n werden in den neuen Szenenzustand δ_n' übertragen.
- ②b: Die Werte der Nicht-SPs werden aus δ_n nach δ_n' übertragen. Der Szenenzustand δ_n ist nun überflüssig geworden.

Zu ③a/b und ②a/b: In der Praxis bedeuten diese Übergänge, dass im ersten Fall σ_{n-1} durch $\sigma_{n-1,0}$, im zweiten Fall δ_n durch δ_n' überschrieben werden.

Wir gehen nochmals kurz auf die Möglichkeiten ein, wie man in einer vorläufigen Szenenzustandsfolge ($\delta_a, \dots, \delta_b$) die Ausgangssituation für eine Simulation modellieren kann. In der vorläufigen Szenenzustandsfolge sind Werteverläufe für alle SPs und alle Nicht-SPs gespeichert. Weiterhin sind für jede SP Intervalle manueller Kontrolle festgelegt. Die Werteverläufe von Nicht-SPs und die Werteverläufe von SPs in Bereichen manueller Kontrolle werden durch die Simulation nicht verändert und in die endgültige Szenenzustandsfolge übernommen. Das heißt, für eine gewünschte Simulation müssen zunächst modelliert werden:

- der Anfangszustand δ_a , der sowohl die Anfangswerte aller SPs als auch aller Nicht-SPs enthält
- die Werteverläufe der Nicht-SPs, d.h., $(\delta_a|_N, \dots, \delta_b|_N)$ (N: Menge der Nicht-SPs) (Werteverläufe für Nicht-SPs können auch nach der Simulation festgelegt werden, da sie auf diese keinen Einfluss haben und auch nicht von ihr verändert werden.)
- die Werteverläufe aller SPs in manuell kontrollierten Intervallen

Alle restlichen Werteverläufe, also die Verläufe der SP-Werte in nicht manuell kontrollierten Intervallen, werden automatisch, d.h. durch Simulation, bestimmt.

Auf eine Besonderheit muss hingewiesen werden, die aus den obigen Definitionen folgt: Es kann SPs geben, die in keiner Simulation jemals verändert werden. Man denke dabei an SPs, die in keiner Zuweisung in einer VH linksseitig vorkommen. Die Werteverläufe dieser SPs werden in nicht manuell kontrollierten Intervallen durch die Simulation immer konstant auf den Anfangswert in δ_a bzw. den letzten Wert im unmittelbar linksseitig vorkommenden manuell kontrollierten Intervall gesetzt, weil sich der Wert einer solchen SP in nicht manuell kontrollierten Intervallen durch die Simulation von Frame zu Frame nicht verändert. Folglich werden in diesen Intervallen die in der vorläufigen Szenenzustandsfolge vorhandenen Werteverläufe durch den konstanten Wert überschrieben. Soll ein im Voraus gesetzter Werteverlauf einer SP erhalten bleiben, so muss diese im gesamten Simulationsintervall als manuell kontrolliert festgelegt werden.

Damit ist die Modellierung der Zustandsübergänge während der Simulation im Prinzip festgelegt, wobei Key-Techniken außer Acht gelassen wurden. MaxControl verwendet in der technischen Umsetzung die Keyframe-Animation. Beim Lesen aus der Szenenzustandsfolge werden die vorhandenen Keys genutzt, und beim Schreiben in die Szenenzustandsfolge werden eventuell neue Keys erzeugt, um so die durch Simulation bestimmten zeitlichen Veränderungen der 3D-Objekte als Animation in der Szenenzustandsfolge zu speichern. Das genaue Verfahren beim Einsatz dieser Technik wird in Kapitel 10.3.1 behandelt.

7 Datenmodell und Designrichtlinien

7.1 Einführung

Dieses Kapitel gibt einen Überblick über die in MaxControl verwendeten Datenstrukturen. Dabei werden zwar schon die meisten Konzepte genannt, jedoch werden sie für ein vollständiges Verständnis erst in den folgenden Kapiteln ausreichend detailliert erläutert. Von der genauen internen Speicherung der 3D-Objekte im 3D-Animationsprogramm wird vorerst abstrahiert. Für die Datenstrukturen sind hier allein die Java-Klassen relevant, welche den 3D-Objekten jeweils zugeordnet wurden. Diese Klassen werden hier als „Objekttypen“ oder kurz „OTs“ bezeichnet.

Ein Objekttyp, der einem 3D-Objekt zugeordnet wird, bestimmt, welche Eigenschaften des Objektes nun in Java zur Verfügung stehen. Man kann dann auf diese Eigenschaften zugreifen, indem man das Java-Objekt (SO oder Simulationsobjekt, siehe Kap. 6.4) verwendet, welches das 3D-Objekt in Java repräsentiert und aufgrund der Zuordnung einer Java-Klasse zu dem 3D-Objekt erzeugt wird. Dieses Java-Objekt ist eine Instanz der zugeordneten Klasse.

Die SPs dieser Java-Objekte sind selbst Objekte, deren jeweilige Klasse hauptsächlich vom Datentyp abhängt, den sie speichern. Als Objekte können SPs ihre MPs selbst verwalten und den Zugriff auf ihre Daten nach dem Zustand der MPs richten.

Die Java-Klasse eines 3D-Objektes bestimmt insbesondere das zeitliche Verhalten eines Objektes, indem es die Verhaltensweisen, im Folgenden kurz VHs genannt, festlegt, denen das Objekt bei einer Simulation folgt. Auch die Verhaltensweisen werden als Java-Klassen definiert. Zur Vereinfachung werden sowohl diese Klassen als auch ihre Instanzen als Verhaltensweisen bezeichnet. SPs basieren zwar auch auf Klassen, jedoch wird der Begriff SP hier meistens für Instanzen dieser Klassen verwendet. Zur Steuerung von außen durch den Benutzer oder durch andere Verhaltensweisen und zur Kommunikation der Verhaltensweisen untereinander können Verhaltensweisen selbst SPs enthalten, ebenso können sie weitere Unter-Verhaltensweisen besitzen.

In MaxControl kann ein Objekttyp einen Aufbau haben, wie er in der folgenden Abbildung 29 illustriert wird. Es folgt die Bedeutung der Pfeile in Abbildung 29:

A -----► B: A ist Unterklasse von B (Vererbung).

A ———► B: Jedes Objekt der Klasse B besitzt ein Feld, dessen Inhalt zur Klasse A gehört. Dies bedeutet hier auch, dass dieses Feld bereits bei seiner Deklaration mit einem Objekt des Typs A besetzt wird. Ein solches Feld wird in dem obigen Diagramm **fettgedruckt** gezeigt. Es wird eine Besitzstruktur festgelegt, die bestimmt, welche Verhaltensweisen und SPs zu einem Objekttyp oder zu einer anderen Verhaltensweise gehören. Da SPs im Gegensatz zu VHs in der Regel nicht über ihre Klasse A, sondern über den Namen des sie in der Klasse B referenzierenden Feldes identifiziert werden, ist der Name dieses Feldes für SPs am Pfeil angegeben.

Zu den Klassenbezeichnungen:

Die Bezeichnungen sind dem Quellcode von MaxControl entnommen. Die Präfixe „MCD_“, „MCB_“ und „MCP_“ weisen auf OTs (dynamic), VHs (behaviour) bzw. SPs (property) hin. Die Konventionen für die Namenspräfixe werden in Kapitel 7.3 näher erläutert.

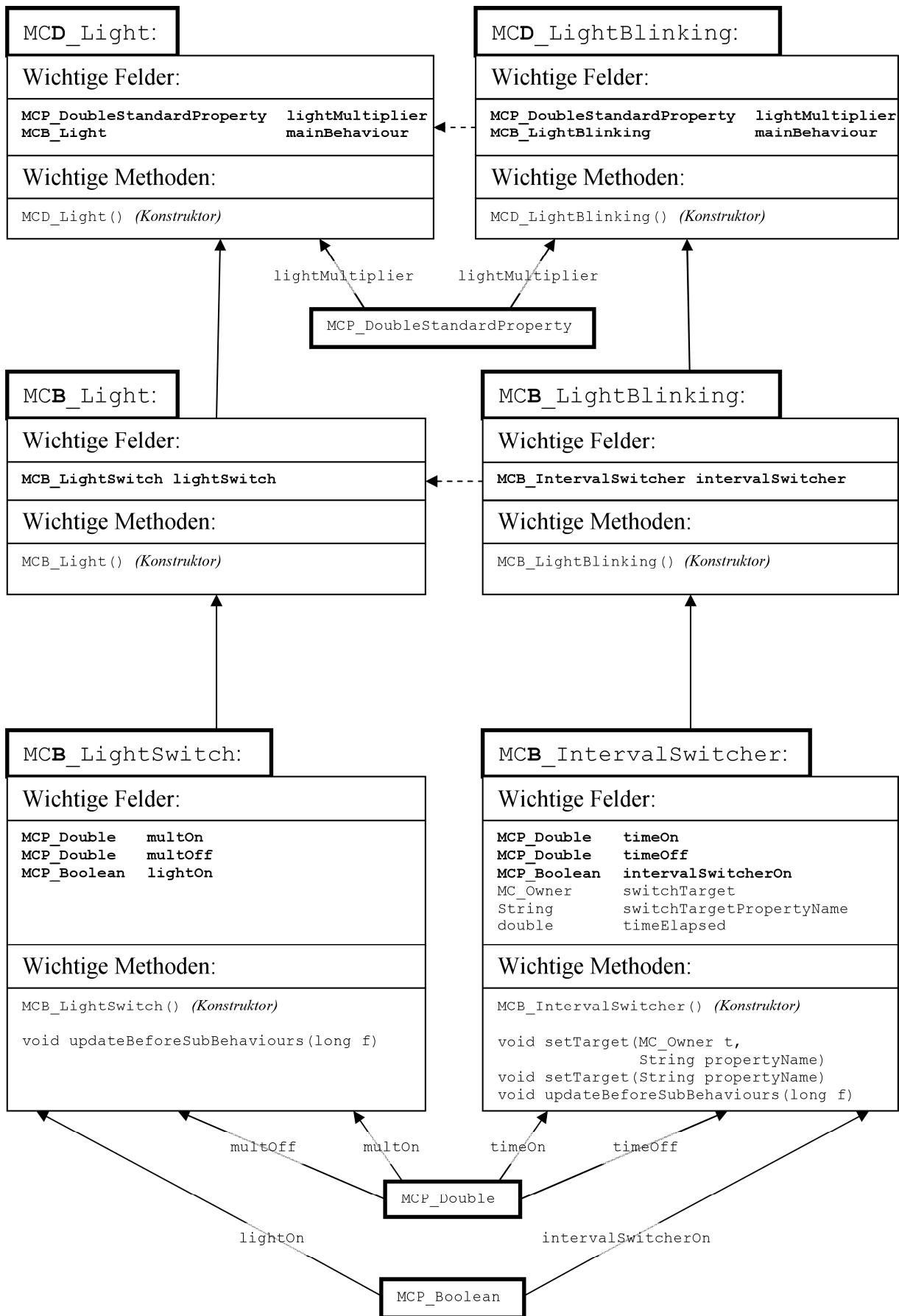


Abbildung 29: Klassenzusammenhänge zu „MCD_LightBlinking“

Die in Abbildung 29 gezeigte Klasse „MCD_LightBlinking“ ist der Objekttyp für ein blinkendes Licht. Sie kann nur 3D-Objekten in der 3D-Szene zugeordnet werden, welche für sich schon die Eigenschaften besitzen, auf die im Java-Code über Standard-SPs zugegriffen werden soll. Die Oberklasse „MCD_Light“ definiert ein Licht, das nicht selbständig blinkt sondern nur ein- und ausgeschaltet werden kann.

Die für ein Blinklicht des Typs „MCD_LightBlinking“ erforderliche Standard-SP „lightMultiplier“ wird aus der Klasse „MCD_Light“ geerbt. Diese Eigenschaft kontrolliert die Helligkeit einer Lichtquelle. Sie hat den Typ „MCP_DoubleStandardProperty“. Ein Objekt dieses Typs stellt eine Standard-Double-SP dar und baut eine bidirektionale Verbindung zu der entsprechenden Eigenschaft des zugehörigen 3D-Objektes auf, die ebenfalls Fließkommawerte speichert. Im Deklarationsteil von „MCD_Light“ wird mit der Anweisung

```
public final MCP_DoubleStandardProperty lightMultiplier
    =new MCP_DoubleStandardProperty("Multiplier",this);
```

eine Instanz des Typs „MCP_DoubleStandardProperty“ erzeugt und dem Feld lightMultiplier zugewiesen.

Dem Konstruktor einer Standard-SP muss als String der Name der entsprechenden Eigenschaft im ursprünglichen 3D-Objekt übergeben werden, sonst würden bei einem Zugriffsversuch auf diese Standard-SP Laufzeitfehler entstehen. Ebenso müssen die Standard-SP und die ursprünglich vorhandene Eigenschaft kompatible Datentypen haben. Eine so erzeugte Standard-SP kann jedoch einem beliebig benannten Feld zugewiesen werden, das allerdings als `public` und `final` deklariert werden muss (dies wird in Kap. 7.2.1 näher begründet). Die Standard-SP, die dem Feld „lightMultiplier“ zugewiesen wird, erhält auf diese Weise eine Verbindung zur ursprünglichen Eigenschaft „Multiplier“. Daraus ergibt sich, dass der Objekt-Typ „MCD_LightBlinking“ nur einem 3D-Objekt zugeordnet werden darf, das auch ohne zugeordneten Java-Typ bereits eine Eigenschaft mit der Bezeichnung „Multiplier“ und Fließkommawerten besitzt. Dies trifft im Allgemeinen nur auf Lichtquellen zu.

Es wurde für „MaxControl“ als Konvention festgelegt, dass jeder Objekttyp mit Verhaltensweisen genau eine so genannte Hauptverhaltensweise (kurz Haupt-VH) hat, die im Feld `mainBehaviour` referenziert wird. Die Typbezeichnung dieser Verhaltensweise soll aus der Bezeichnung des Objekttyps abgeleitet sein. Die Hauptverhaltensweise für den OT „MCD_Light“ hat daher den Typ „MCB_Light“. Direkt bei der entsprechenden Felddeklaration wird dem Feld „mainBehaviour“ eine Instanz der Klasse „MCB_Light“ zugewiesen, wie dieser Auszug aus dem Quellcode zeigt:

```
public final MCB_LightBlinking mainBehaviour = new MCB_LightBlinking(this);
```

Der Typ „MCD_LightBlinking“ erbt das Feld `mainBehaviour` von der Klasse „MCD_Light“. Nach einer entsprechenden Konvention für MaxControl (siehe Kapitel 7.2.3) wird hier jedoch auch ein eigenes Feld mit der Bezeichnung „mainBehaviour“ deklariert. Dieses Feld überdeckt nun das gleichnamige geerbte Feld. Dem neuen Feld wird der Typ „MCB_LightBlinking“ und eine Instanz dieser Klasse zugeordnet. So wie MCD_LightBlinking eine Unterklasse von MCD_Light ist, wird auch die Klasse MCB_LightBlinking nach dieser Konvention als Unterklasse von MCB_Light definiert. So erhalten beide Objekttypen eigene Hauptverhaltensweisen, die wie die Objekttypen selbst voneinander erben. So kann die erbende VH ohne Seiteneffekte auf die vererbende VH modifiziert werden. Dies wäre nicht möglich, wenn „MCD_LightBlinking“ einfach die von

„MCD_Light“ geerbte Haupt-VH übernehmen würde und sich so beide OTs diese VH teilen würden.

Wird nun ein Objekt des Typs MCD_LightBlinking simuliert, wird nicht mehr die geerbte Verhaltensweise des Typs MCB_Light ausgeführt, sondern die neu deklarierte VH des Typs MCB_LightBlinking. Dies ist nicht selbstverständlich, denn Java führt zwar eine dynamische Bindung für Methodenaufrufe durch, jedoch eine statische Bindung für Felder. Auf welches Feld bei einem Aufruf von `X.mainBehaviour` zugegriffen wird, hängt also nicht vom tatsächlichen Typ des in `X` referenzierten Objektes ab, sondern vom deklarierten Typ des Feldes `X`.

Für die folgende Betrachtung sei angenommen, dass sowohl die Klasse „MCB_Light“ als auch die davon ererbende Unterklasse „MCB_LightBlinking“ jeweils eigene **unterschiedliche** Methoden `updateBeforeSubBehaviours(long f)` implementieren. Diese Methoden werden bei der Simulation einer VH neben weiteren Methoden ausgeführt. Sei `X` nun vom Typ „MCD_Light“, es enthalte aber ein Objekt des Typs „MCD_LightBlinking“. Wird nun durch einen Befehl wie `X.mainBehaviour.updateBeforeSubBehaviours(f)` für `X` in dessen Feld `mainBehaviour` die Methode `updateBeforeSubBehaviours(f)` aufgerufen, so würde aufgrund des Typs von `X` auf das ursprüngliche Feld `mainBehaviour` aus „MCD_Light“ mit dem Typ „MCB_Light“ zugegriffen werden, das in „MCD_LightBlinking“ eigentlich durch ein gleichnamiges Feld überdeckt ist. Dieses überdeckte Feld enthält eine VH des Typs MCB_Light, und somit würde deren Methode `updateBeforeSubBehaviours(f)` aufgerufen werden und nicht die entsprechende gleich bezeichnete Methode der Klasse „MCB_LightBlinking“. Das Simulationssystem „MaxControl“ führt jedoch für Felder in OTs und VHs, die SPs oder VHs referenzieren, die in Java fehlende dynamische Bindung für Felder selbst durch.

Die Verhaltensweise des Typs MCB_Light enthält eine Unterverhaltensweise des Typs MCB_LightSwitch, die in dem Feld „lightSwitch“ referenziert wird. Das rekursive Finden und Ausführen solcher Unterverhaltensweisen wird ebenfalls von „MaxControl“ automatisch durchgeführt.

Eine VH des Typs MCB_LightSwitch hat die zwei Double-SPs „multOn“ und „multOff“ sowie eine boolesche SP „lightOn“. Der Werte der SPs „multOn“ und „multOff“ bestimmen die Helligkeit der Lichtquelle im eingeschalteten Zustand bzw. im ausgeschalteten Zustand. Das Verhalten dieser VH besteht darin, je nach Zustand der SP „lightOn“ der Lichtquelle die entsprechende Helligkeit aus „multOn“ oder „multOff“ zuzuweisen. Um die Helligkeit der Lichtquelle zu beeinflussen, greift die VH auf die Standard-SP „lightMultiplier“ zu, die im OT „MCD_Light“ zu finden ist.

Da die Verhaltensweise des Typs MCB_LightBlinking von der Verhaltensweise MCB_Light erbt, enthält sie automatisch auch deren Unterverhaltensweise „lightSwitch“.

Zusätzlich erhält sie die Verhaltensweise „MCB_IntervalSwitcher“. Diese kann eine beliebige boolesche SP in bestimmten Zeitintervallen abwechselnd auf `true` und `false` setzen und simuliert somit einen Intervallschalter. Über die Methode `setTarget(String propertyName)` kann der Name der zu beeinflussenden SP `X` festgelegt werden. Die alternative Methode `setTarget(MC_Owner t, String propertyName)` kann zusätzlich den Besitzer dieser SP angeben, also die VH oder den OT, der die gesuchte SP `X`

enthält. Sowohl SOs als auch VHs sind Instanzen von Unterklassen der Klasse MC_Owner, weshalb sowohl VHs also auch SPs als Parameter `t` dieser Methode übergeben werden können. Eine Referenz auf den ggf. angegebenen Besitzer `t` wird im Feld `switchTarget` gespeichert, der Name der SP selbst in `switchTargetPropertyName`.

Die Double-SPs „`timeOn`“ und „`timeOff`“ der VH „`MCB_IntervalSwitcher`“ bestimmen, wie lange die zu beeinflussende SP den Wert `true` bzw. `false` behält. Die boolesche SP „`intervalSwitcherOn`“ bestimmt, ob der Intervallschalter eingeschaltet sein soll und somit der Wert der zu beeinflussenden SP in den festgelegten Intervallen abwechselnd geändert wird (wenn `intervalSwitcherOn=true`) oder ob die SP konstant den Wert `false` erhält (wenn `intervalSwitcherOn=false`). Die seit dem Einschalten des Intervallschalters bisher verstrichene Zeit wird in dem Feld `timeElapsed` gespeichert.

Wird der OT `MCD_LightBlinking` einer Lichtquelle in der Szene zugewiesen, kann bei einer entsprechenden Einstellung der SPs diese Lichtquelle durch die Simulation automatisch in wählbaren Zeitintervallen blinken. So kann z.B. der Blinker bei einem Auto animiert werden oder ein Baustellenwarnlicht mit Blinkfunktion.

7.2 Prinzipien für den Aufbau von Objekttypen und Verhaltensweisen

Bei der Entwicklung von MaxControl wurde besonderer Wert darauf gelegt, dem Benutzer möglichst viele Hilfen zur Verfügung zu stellen, ihn jedoch nicht zu deren Verwendung zu zwingen oder darauf zu beschränken. Schon die Entscheidung für Java-Code ermöglicht eine große Anzahl von Konzepten und Methoden zur Implementation von Verhaltensweisen. Hierzu kann der volle Sprachumfang von Java genutzt werden, und durch externe Schnittstellen wie das Java Native Interface⁵⁸ können auch andere Programmiersprachen wie C++ eingebunden werden. Weitere Schnittstellen wie die Scripting-Schnittstelle des Internet-Explorers, über die Java-Applets sowohl Java Skripte als auch Visual Basic Skripte ausführen können (siehe Kap. 5.9), ermöglichen eine Integration weiterer Werkzeuge. Diese Scripting-Methoden sind in der technischen Umsetzung von MaxControl verfügbar, da es als Applet im Microsoft Internet Explorer läuft. Auf diese Weise wird in der technischen Umsetzung eine Verbindung zu einer Shockwave-Anwendung sowie zu 3D-Studio-MAX realisiert.

Wie bereits in Kapitel 2.1 erwähnt wurde, gibt es für Java frei verfügbare Klassenbibliotheken, die sowohl die Verwendung von Fuzzy Methoden als auch von neuronalen Netzen unterstützen. Eine Nutzung dieser Konzepte ist also für die Definition von Verhaltensweisen auch auf der Basis von Java-Code möglich und könnte so in MaxControl verwendet werden.

MaxControl ist ein erweiterbares Simulationssystem. Ein Benutzer dieses Systems ist nicht auf die gegebenen Objekttypen beschränkt, sondern ein wichtiges Konzept dieses Systems besteht darin, die bereits vorhandenen Objekttypen zu erweitern und neue zu implementieren, um so jedem 3D-Objekt in einer Szene das gewünschte Verhalten geben zu können.

Dabei sollte man einige Konventionen und Designrichtlinien beachten, die im Folgenden angegeben werden. Sie wurden speziell für MaxControl entwickelt und sichern die Beibehaltung klarer verständlicher und modularer Strukturen. Die Konventionen und Designkriterien sollen den Benutzer eher unterstützen als einschränken, eine Beachtung dieser Kriterien ist im Allgemeinen deshalb nicht zwingend erforderlich.

⁵⁸ <http://java.sun.com/j2se/1.5.0/docs/guide/jni/>, 29.04.2007

7.2.1 Richtlinien für die Strukturierung von Objekttypen

Jeder OT kann mehrere SPs und VHs besitzen. VHs können selbst weitere Unterverhaltensweisen (kurz „Unter-VHs“) und eigene SPs besitzen. OTs und VHs werden im Folgenden daher auch als „Besitzer“ der VHs und SPs bezeichnet, die sie direkt enthalten. Dieses „Besitzen“ wird erst bei der tatsächlichen Instanzierung eines OTs manifestiert. In der Klasse eines OTs selbst ist lediglich festgelegt, wie ihm bei seiner Instanzierung jeweils neu erzeugte Instanzen von VHs und SPs zugewiesen werden. Gleiches gilt für die VHs. Dabei werden auch die Bezeichnungen der Felder definiert, die bei einer Instanzierung eines OTs oder einer VH deren zugehörige SPs oder VHs referenzieren. Die zugehörigen SPs und VHs werden allerdings erst bei der Instanzierung ihres Besitzers als Objekte erzeugt, weshalb die hierarchische Besitzstruktur eines OTs erst bei der Instanzierung des OTs wirklich entsteht. Da jedoch durch die Klassendefinitionen bereits festgelegt ist, welche Besitzhierarchien bei einer Instanzierung des OTs entstehen werden, kann man schon auf Typebene diese Hierarchien angeben.

Es folgen Beispiele dafür, wie man in MaxControl eine SP oder eine VH ihrem Besitzer zuordnet. Die dabei verwendeten Konstruktoren von SPs und VHs werden in den Kapiteln 8.2.1 bzw. 8.6.2 näher erläutert.

Einem OT oder einer VH kann mit einer Anweisung der folgenden Art eine SP zugewiesen werden:

```
public final MCP_Double springLength = new MCP_Double("Length",this);
```

Die so neu erzeugte SP wird in ihrem Besitzer dem Feld `springLength` zugewiesen. Der Name dieses Feldes kann frei gewählt werden. Die SP wird dann im Text oder in Diagrammen auch mit dem Namen dieses Feldes bezeichnet, z.B. als SP „springLength“. SPs werden nicht mit ihrer zugehörigen Klasse bezeichnet, weil dies im Allgemeinen kein eindeutiges Merkmal dieser SP ist.

Gemäß einer bereits in Kapitel 7.1 genannten Konvention wird jedem OT höchstens eine VH, die Hauptverhaltensweise (kurz Haupt-VH), direkt zugewiesen, die dann selbst aber weitere Unter-VHs besitzen kann. Die Klassenbezeichnung der Haupt-VH soll sich aus dem Namen des zugehörigen OTs ableiten, damit eine Zuordnung zwischen OT und Haupt-VH bereits anhand der Klassenbezeichner leicht möglich ist. So enthält z.B. der OT „MCD_Car“ die Haupt-VH „MCB_Car“. Die hierbei genutzten Präfixkonventionen werden in Kapitel 7.3 erläutert.

Ein SO muss keine Haupt-VH besitzen. Es kann auch ohne jede VH nur SPs zur Verfügung stellen, die z.B. von anderen SOs ausgelesen werden können. Dies trifft auf Fahrbahnmarkierungen des Typs `MCD_LaneMarker` zu, deren SPs Fahrzeugen des Typs `MCD_Car` den Verlauf einer Rennstrecke anzeigen können.

Die folgende Beispielanweisung zeigt, wie einem OT oder einer VH eine VH zugeordnet wird:

```
public final MCB_CarControlAI carControlAI=new MCB_CarControlAI(this);
```

Die so neu erzeugte VH wird in ihrem Besitzer dem Feld `carControlAI` zugewiesen. Auch der Name dieses Feldes kann frei gewählt werden, wenn es sich bei der VH nicht um die oben erläuterte Haupt-VH handelt, die in dem OT dem public-Feld „mainBehaviour“ zugewiesen werden muss. Die VH wird im Text oder in Diagrammen im Gegensatz zu SPs mit dem Namen ihrer zugehörigen Klasse bezeichnet, die hier „MCB_CarControlAI“ ist,

da ihr Besitzer in der Regel nur eine VH dieses Typs besitzen wird und eine VH anhand ihres Typs klar zu erkennen ist. Dagegen lässt die Bezeichnung des Feldes, das diese VH in ihrem Besitzer referenziert, in der Regel nicht auf den Typ der VH schließen, der aber entscheidend ist für das Verhalten, das diese VH spezifiziert.

Damit MaxControl erkennt, dass eine VH `X` oder eine SP `X` direkt zu einem Besitzer `A` gehört, so dass `A` der direkte Besitzer von `X` ist, muss `X` in `A` durch ein Feld referenziert werden, das als `public` deklariert ist. Dies hat zwar auch technische Gründe, es ist so aber auch möglich, dass `A` in Feldern Referenzen auf VHs und SPs **anderer** VHs oder SOs speichert, ohne dass diese von MaxControl als direkt zu `A` gehörend erkannt werden, wenn diese Felder **nicht** als `public` deklariert werden. So kann `A` für einen vereinfachten Zugriff auf die VHs und SPs anderer SOs und VHs Referenzen auf deren VHs und SPs in Feldern speichern, ohne dabei die Liste der tatsächlich zu `A` gehörenden VHs und SPs zu verändern. Ein als `public` deklariertes Feld, das eine VH `X` oder eine SP `X` im jeweiligen Besitzer referenziert, wird hier auch als Container-Feld bezeichnet. Ein solches Feld muss zusätzlich als `final` deklariert werden, um zu verhindern, dass der Inhalt dieses Feldes zur Laufzeit durch eine Neuzuweisung geändert wird. Eine solche Änderung ist nicht erlaubt, weil die Besitzhierarchie eines OTs wie oben erläutert schon auf Typebene fest definiert und ablesbar sein soll. Dies ist auch der Grund dafür, dass eine SP oder eine VH wie in den obigen Beispielen gezeigt direkt bei der Deklaration des sie referenzierenden Container-Feldes diesem Feld zugewiesen werden müssen, da nach der Deklaration eines `final`-Feldes keine erneute Zuweisung an dieses Feld möglich ist. Ob ein Container-Feld als `final` deklariert ist, wird von MaxControl zur Laufzeit überprüft.

In jedem OT entsteht durch seine VHs und Unter-VHs eine hierarchische Struktur. Wird diese Hierarchie in Baumform dargestellt, so bildet die Haupt-VH die Wurzel dieser Baumstruktur. Um die Gesamtstruktur des OTs darzustellen, wird im Folgenden auch der OT selbst in die Darstellung eingebunden und bildet dann die neue Wurzel, ebenso werden die SPs des OTs und seiner VHs als Blätter eingefügt. Eine solche Hierarchie wird auch als OT-Hierarchie bezeichnet. Eine instanziierte OT-Hierarchie wird auch SO-Hierarchie genannt.

Wie in den Kapiteln 8.3.3 und 8.3.4 erläutert wird, können die SPs und VHs über die Bezeichnungen der sie referenzierenden `public`-Felder für bequeme Zugriffe automatisch in einer OT-Hierarchie gefunden werden. Wie dort weiter erläutert wird, können VHs auch anhand ihres Typs gefunden werden, was bei VHs zu bevorzugt ist.

Diagramme zu OT-Hierarchien und auch zu SO-Hierarchien verwenden die Präfixe „OT“, „VH“, „SP“ und „SO“, um jeweils OTs, VHs, SPs bzw. SOs als solche zu kennzeichnen. Die Bezeichner der Klassen bzw. Objekte folgen diesen Präfixen in tiefgestellter Schrift. Solche Bezeichnungen sind zunächst für diese Beispiele noch frei gewählt, folgen später aber auch den Namenskonventionen für tatsächlichen Quellcode, die in Kapitel 7.3 erläutert werden. Während OTs, VHs und SOs in Diagrammen die Namen ihrer jeweiligen Java-Klasse als Bezeichner erhalten, werden SPs in Diagrammen mit dem Namen des `public`-Feldes bezeichnet, das sie in ihrem Besitzer referenziert. Die Pfeile beginnen bei SPs bzw. VHs und zeigen jeweils auf deren direkten Besitzer.

Die hierarchische Struktur der VHs eines OTs sollte semantische Zusammenhänge zwischen den VHs widerspiegeln. Ein Beispiel einer OT-Hierarchie soll dies verdeutlichen:

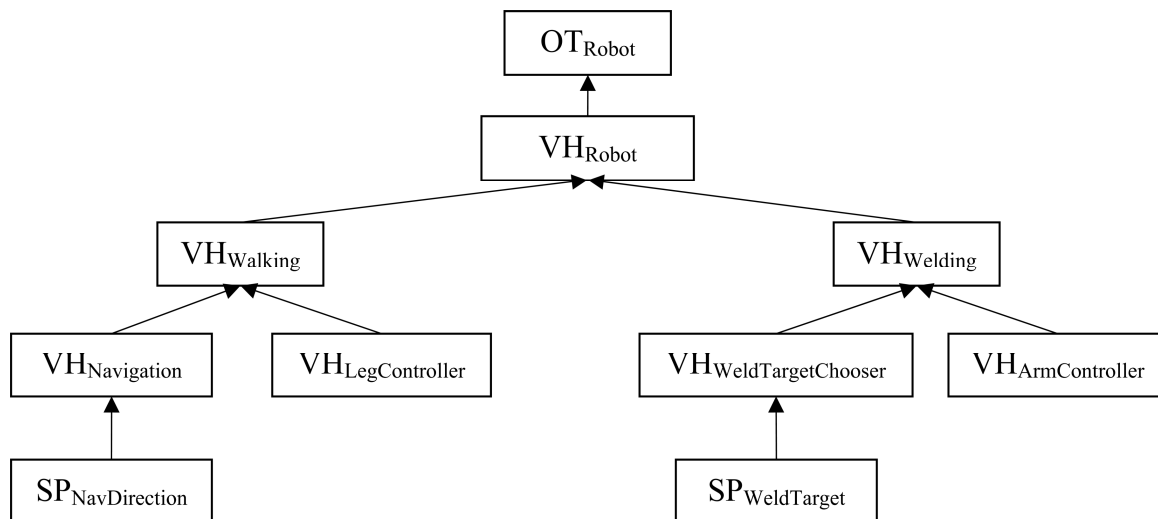


Abbildung 30: OT-Hierarchie zu „OT_{Robot}“

„OT_{Robot}“ definiert das Verhalten eines Arbeitsroboters, der sowohl selbständig auf Beinen laufen als auch Werkstoffe schweißen kann. Seine Haupt-VH erhält der oben genannten Konvention folgend mit „VH_{Robot}“ eine Bezeichnung, die aus „OT_{Robot}“ abgeleitet ist. „VH_{Robot}“ enthält die VHs „VH_{Walking}“, die das Laufen steuert, und „VH_{Welding}“, die das Schweißen steuert. „VH_{Walking}“ enthält die zwei Unter-VHs „VH_{Navigation}“ und „VH_{LegController}“, die mit dem Laufen verbundene Aufgaben übernehmen. Es ist daher sinnvoll, wenn auch technisch nicht zwingend notwendig, diese beiden VHs als Unter-VHs von „VH_{Walking}“ zu definieren. „VH_{Navigation}“ übernimmt die Navigation und bestimmt die Laufrichtung des Roboters. Sie speichert diese Laufrichtung in „SP_{NavDirection}“. „VH_{LegController}“ liest die durch „VH_{Navigation}“ bestimmte Laufrichtung aus dieser SP und steuert die Beine des Roboters entsprechend. „VH_{Welding}“ enthält die zwei Unter-VHs „VH_{WeldTargetChooser}“ und „VH_{ArmController}“, für Aufgaben im Zusammenhang mit dem Schweißen. „VH_{WeldTargetChooser}“ wählt ein Objekt zum Schweißen in der direkt (ohne zu laufen) erreichbaren Umgebung aus. Sie speichert eine Referenz auf dieses Objekt in ihrer SP „SP_{WeldTarget}“. „VH_{ArmController}“ liest das zu schweißende Objekt aus dieser SP und steuert den Schweißarm des Roboters entsprechend.

Die VHs aus dem obigen Beispiel, die Unter-VHs enthalten, fassen ihre Unter-VHs zu Gruppen zusammen und erzeugen so eine übersichtliche Struktur.

VHs mit Unter-VHs können jedoch auch selbst schon ein bestimmtes Verhalten haben, und dabei ihre Unter-VHs als Ergänzung verwenden. So kann eine Navigations-Verhaltensweise die Bewegung eines Objektes im Raum steuern, so dass sich dieses Objekt beispielsweise auf einen Zielpunkt zu bewegt. Dabei kann sie als Unter-VH eine Verhaltensweise haben, die eine Annäherung an andere Objekte feststellen und einen Ausweichkurs berechnen kann. Dieser Ausweichkurs könnte von der übergeordneten Navigations-Verhaltensweise berücksichtigt werden. Um solche Hintereinanderausführungen von VHs und Unter-VHs in der gewünschten Reihenfolge zu ermöglichen, kann jede VH Programmschritte definieren, die **vor** den Programmschritten ihrer Unter-VHs und weitere Programmschritte, die **danach** ausgeführt werden sollen. Diese Mechanismen werden in Kapitel 8.6.3 genauer erläutert.

Der Vorteil einer solchen Strukturierung gegenüber einer einzelnen VH, die sowohl die Navigation in Richtung eines Zielpunktes also auch die Ausweichkurse berechnet, liegt darin, dass bei der Verwendung kompatibler Datenschnittstellen sowohl die Navigations-Verhaltensweise als auch deren Unter-VH zur Berechnung von Ausweichkursen einzeln

jeweils durch andere VHs ersetzt werden können, die z.B. veränderte Bedingungen erfüllen. Eine VH zur Berechnung von Ausweichkursen, die nur die Abstände zwischen Objektmittelpunkten zur Erkennung von Annäherungen verwendet, könnte durch eine VH ersetzt werden, die auch die genaue Form der Objekte berücksichtigt.

7.2.2 Wiederverwendung von Verhaltensweisen

Ein Teil von „OT_{Robot}“ aus Kapitel 7.2.1 könnte in einem neuen OT „OT_{RobotCommunicating}“ wieder verwendet werden, auch wenn dieser neue OT nicht von „OT_{Robot}“ erbt. „OT_{RobotCommunicating}“ möge zusätzlich über eine Kommunikationsschnittstelle mit der Außenwelt verfügen und könnte die folgende Struktur haben:

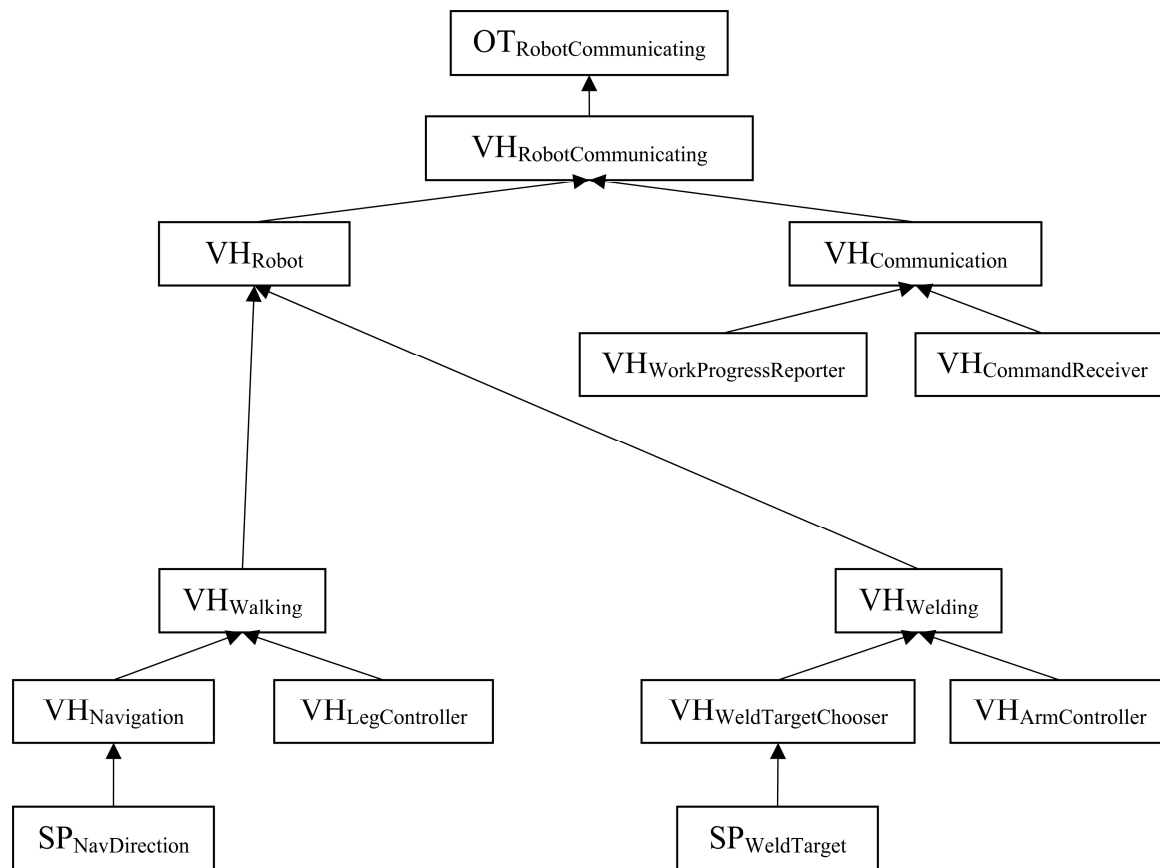


Abbildung 31: OT-Hierarchie zu „OT_{RobotCommunicating}“

Durch die oben genannte Konvention, dass jedem OT höchstens eine Haupt-VH direkt zugeordnet wird, die dann selbst weitere Unter-VHs enthalten kann, können andere OTs entweder nur einzelne seiner Unter-VHs wieder verwenden oder seine gesamte Haupt-VH, die dann stets auch alle zukünftig hinzugefügten VHs des OTs enthalten wird.

Verwendet ein OT „A“ eine VH „V“ von OT „B“ wieder und werden in einer späteren Implementation der VH „V“ des OTs „B“ z.B. weitere Unter-VHs hinzugefügt, so würde dann auch der OT „A“ automatisch diese weiteren Unter-VHs erhalten.

Der neue „OT_{RobotCommunicating}“ nutzt diese Möglichkeit. Dieser gegenüber „OT_{Robot}“ um eine Kommunikationsschnittstelle erweiterte Roboter-Objektyp verwendet zunächst die Haupt-VH „VH_{Robot}“ aus dem oben definierten „OT_{Robot}“ wieder, als Unter-VH seiner eigenen Haupt-VH. Jede spätere Änderung an der Implementation von „VH_{Robot}“, wie das Hinzufügen neuer Unter-VHs, wirkt so nicht nur auf „OT_{Robot}“, sondern automatisch auch auf

„OT_{RobotCommunicating}“ . Würde „OT_{RobotCommunicating}“ nur direkt die Unter-VHs „VH_{Walking}“ und „VH_{Welding}“ aus „VH_{Robot}“ wieder verwenden, könnten diese in einer späteren Implementation von „VH_{Robot}“ neue Geschwister erhalten, die dann nicht automatisch Teil der OT-Hierarchie zu „OT_{RobotCommunicating}“ würden. Daher wird direkt die Haupt-VH „VH_{Robot}“ übernommen, weil jeder OT nur seine Haupt-VH direkt enthalten kann und seine Haupt-VH damit auch in einer späteren Implementation keine Geschwister erhalten kann, die dann in „OT_{RobotCommunicating}“ fehlen würden.

Aus diesem Grund wird die Verwendung höchstens einer Haupt-VH vorgeschrieben, damit andere OTs diese Haupt-VH wieder verwenden können und dann sichergestellt ist, dass sie stets alle VHs des zugehörigen OTs enthält, auch wenn dem OT neue VHs hinzugefügt werden.

Weiterhin hat er die Unter-VH „VH_{Communication}“, welche die Kommunikation mit der Außenwelt steuert und deren Unter-VHs spezielle Kommunikationsaufgaben übernehmen. „VH_{WorkProgressReporter}“ meldet den aktuellen Arbeitsfortschritt an die Außenwelt, „VH_{CommandReceiver}“ empfängt Arbeitsbefehle von außen.

Wird ein OT instanziiert, werden rekursiv auch Instanzen der VHs und SPs aus der zugehörigen OT-Hierarchie erzeugt. Damit bei einer solchen Instanzierung keine nicht-terminierende Schleife entsteht, darf auf Klassenebene keine VH sich selbst als direkte oder indirekte Unter-VH besitzen. MaxControl kann eine solche Schleife zur Laufzeit erkennen und erzeugt ggf. einen entsprechenden Laufzeitfehler.

7.2.3 Vererbung von Verhaltensweisen und Objekttypen

Der erweiterte Objekttyp „OT_{RobotCommunicating}“ aus Kapitel 7.2.2 kann auch durch Vererbung aus dem Objekttyp „OT_{Robot}“ hervorgehen. Er könnte dann die in Abbildung 32 folgende Struktur haben. Die gestrichelten Pfeile zeigen Vererbungen an. Sie zeigen von der erbenden Klasse auf die vererbende Klasse.

Ein erbender OT oder eine erbende VH erben auch die gesamte Besitzstruktur ihrer jeweiligen vererbenden Oberklasse. Diese geerbten Strukturen werden in den Diagrammen zu OT-Hierarchien allerdings in der Regel nicht in der Hierarchie des erbenden OTs bzw. der erbenden VH wiederholt, um das Diagramm möglichst einfach zu halten.

So erbt im obigen Beispiel VH_{RobotCommunicating}² von VH_{Robot} die Verhaltensweisen VH_{Walking} und VH_{Welding} mit ihren jeweiligen Unter-VHs, dennoch werden diese im Diagramm nicht als Unter-VHs von VH_{RobotCommunicating}² gezeigt, um die Übersichtlichkeit des Diagramms zu erhalten. Auch ein Objekttyp wie OT_{Robot} wird im Allgemeinen weitere SPs und VHs von Oberklassen erben, wie z.B. die SPs SP_{Position} und SP_{Rotation} aus dem Objekttyp OT_{Dynamic}, welche die Position und Rotation aller simulierten Objekte speichern. Dennoch zeigt das Diagramm weder den vererbenden Objekttyp OT_{Dynamic}, noch die von ihm geerbten SPs, da diese für den hier zu erläuternden Sachverhalt nicht relevant sind. Die Diagramme zu OT-Hierarchien erheben also keinen Anspruch auf Vollständigkeit.

Der durch die gezeigte Vererbung entstehende Objekttyp „OT_{RobotCommunicating}²“ hat eine sehr ähnliche OT-Hierarchie wie der vorangehend definierte OT „OT_{RobotCommunicating}“. Durch die Vererbung von „OT_{Robot}“ auf „OT_{RobotCommunicating}²“ erhält „OT_{RobotCommunicating}²“ zunächst dieselben direkt zugehörigen SPs und VHs, die auch „OT_{Robot}“ hat. Die so durch Vererbung auch in „OT_{RobotCommunicating}²“ zu findende Haupt-VH „VH_{Robot}“ wird allerdings durch die neue Haupt-VH „VH_{RobotCommunicating}²“ ersetzt, die ihrerseits wie oben erwähnt von „VH_{Robot}“ erbt

und somit auch deren Unter-VHs erbt. Die direkten Unter-VHs von „VH_{RobotCommunicating}²“ sind damit insgesamt „VH_{Walking}“, „VH_{Welding}“ und „VH_{Communication}“.

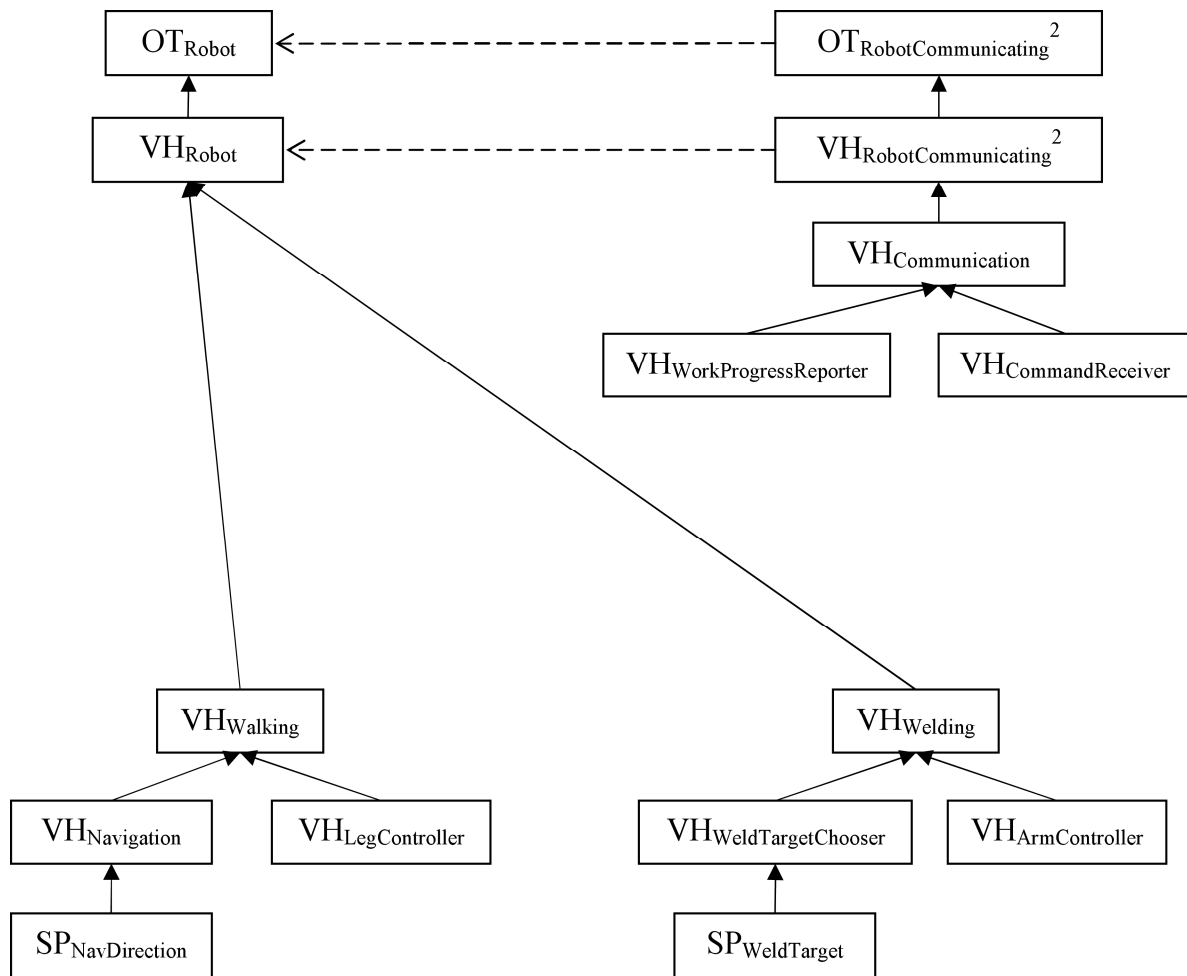


Abbildung 32: OT-Hierarchie zu „OT_{RobotCommunicating}²“

Das Ersetzen der geerbten Haupt-VH „VH_{Robot}“ durch die neue von „VH_{Robot}“ ererbende Haupt-VH „VH_{RobotCommunicating}²“, ist in der hier gezeigten Form für MaxControl als Designregel vorgeschrieben. Diese Ersetzung wird im Folgenden erläutert. In der Klassendefinition zu „OT_{Robot}“ wird sich mit den für MaxControl üblichen Bezeichnungen eine Felddeklaration finden wie:

```
public final MCB_Robot mainBehaviour=new MCB_Robot(this);
```

So wird das Feld `mainBehaviour` gleichzeitig deklariert und initialisiert. In der ererbenden Klasse „OT_{RobotCommunicating}²“ wird dieses Feld durch eine erneute Deklaration der folgenden Form überdeckt:

```
public final MCB_RobotCommunicating mainBehaviour
    =new MCB_RobotCommunicating(this);
```

Auf diese Weise können grundsätzlich sowohl VHs als auch SPs in ererbenden Klassen durch andere VHs und SPs ersetzt werden.

Die ersetzenden VHs und SPs müssen dabei jedoch entweder genau den Typ der VH bzw. der SP haben, die sie jeweils ersetzen, oder einen Subtyp davon. Wird diese Vorschrift verletzt, so

erzeugt MaxControl einen entsprechenden Laufzeitfehler. Diese Vorschrift setzt die Prinzipien der Vererbung aus objektorientierten Programmiersprachen auf natürliche Weise fort. Erbt eine Klasse A von einer Klasse B, so kann im Sinne der Vererbungsprinzipien aus Java erwartet werden, dass die erbende Klasse A mindestens über die Methoden und Felder verfügt, die auch die vererbende Klasse B hat. Werden geerbte Felder aus B in der Klasse A neu deklariert, so überdecken diese neuen Felder die geerbten Felder nur, die überdeckten Felder aus B bleiben jedoch auch in A verfügbar. Insbesondere kann die Signatur⁵⁹ aus B geerbter Methoden in A nicht verändert werden, es können lediglich neue Methoden mit anderer Signatur hinzugefügt werden und geerbte Methoden gleicher Signatur neu implementiert werden. Auf SPs und VHs ausgedehnt würde man nun auch erwarten, dass ein Besitzer A (also ein OT oder eine VH), der von einem Besitzer B erbt, mindestens über die gleichen SPs und VHs verfügt, wie Besitzer B. Diese strenge Forderung kann gelockert werden, indem man nur verlangt, dass der Besitzer A die aus B geerbten SPs und VHs zwar durch andere SPs und VHs ersetzen darf, wobei diese jedoch zum Typ der ersetzten SPs und VHs passen müssen, was auch Subtypen erlaubt. Hat ein Besitzer B beispielsweise eine SP „X“ des Typs MCP_Double, so darf der von B erbende Besitzer A diese SP „X“ durch eine neue SP „X“ des Subtyps MCP_DeltaT ersetzen, da MCP_DeltaT eine Unterklasse von MCP_Double ist. Damit ist garantiert, dass die neue SP „X“ mindestens über die gleichen Felder und Methoden verfügt, wie die geerbte SP „X“, wobei jedoch die Felder überdeckt (aber dennoch verfügbar) und die Methoden neu implementiert (aber dennoch mit derselben Signatur) sein können. Hat ein Besitzer B eine VH des Typs MCB_LightBlinking, so darf der von B erbende Besitzer A diese VH durch eine neue VH des Subtyps MCB_LightBlinkingDelayed ersetzen, da MCB_LightBlinkingDelayed eine Unterklasse von MCB_LightBlinking ist. Damit ist auch hier garantiert, dass die neue VH „MCB_LightBlinkingDelayed“ mindestens über die gleichen Felder, Methoden, VHs und SPs verfügt, wie die geerbte VH des Typs MCB_LightBlinking, wobei jedoch auch hier die Felder überdeckt (aber dennoch verfügbar) und die Methoden neu implementiert (aber dennoch mit derselben Signatur) sein können. Ebenso können die aus MCB_LightBlinking geerbten VHs und SPs in MCB_LightBlinkingDelayed zwar ersetzt worden sein, jedoch sind sie nach diesen Ausführungen ebenfalls als zulässig zu betrachten, da auch hier nur ein Ersetzen durch gleiche Typen oder Subtypen erlaubt ist.

Wird nun z.B. mit dem `instanceof`-Operator überprüft, ob ein SO beispielsweise den Typ B hat, so wird dieser Test auch für ein SO Z des Subtyps A als „wahr“ ausgewertet. Wird das hier erläuterte Designkriterium bei der Vererbung von B auf A eingehalten, kann man nun sicher sein, dass man in Z in Bezug auf die Container-Feld-Bezeichnungen die gleichen SPs findet wie in einem SO W des Typs B, das **kein** Objekt des Typs A ist. Diese SPs in Z arbeiten auch jeweils mit Werten des gleichen Typs, der von den ersetzten SPs aus B verwendet wird, da der Typ dieser SPs jeweils höchstens ein Subtyp der aus B geerbten SPs sein kann und der Typ einer SP auch ihren Datentyp bestimmt. Ebenso wird man in Z in Bezug auf die Container-Feld-Bezeichnungen die gleichen VHs finden wie in W. Auch die in Z eventuell neuen VHs haben dann den gleichen Typ oder einen Subtyp der ersetzten VHs aus B.

Auch die in Kap. 8.3.3 erläuterte Baumsuche nach VHs und SPs in einer SO-Hierarchie wird in Z bis auf die erlaubten Ersetzungen die gleichen SPs und VHs finden wie in W. Denn auch bei Ersetzungen bleiben die Bezeichner der Container-Felder unverändert und auch der Typ

⁵⁹ Die Signatur einer Methode setzt sich in Java aus den Typen der Eingabeparameter und dem Typ des eventuell spezifizierten Ausgabewertes zusammen.

einer ersetzten VH bleibt „kompatibel“, weil eine Suche nach einer VH eines bestimmten Typs X auch VHs aller Subtypen von X findet.

Programmcode, der für den Umgang mit SOs des Typs B konzipiert wurde, muss damit im Allgemeinen nicht angepasst werden, um mit SOs des Subtyps A umzugehen, sofern dafür keine rein semantischen Gründe vorliegen (wenn z.B. mit SOs des Typs A wirklich anders verfahren werden **soll** als mit SOs des Typs B).

Es ist für MaxControl eine Konvention, dass Container-Felder, welche eine SP oder eine VH eines OTs oder einer VH festlegen und referenzieren, jeweils mit dem Typ deklariert werden, den das zugewiesene Objekt tatsächlich hat und nicht mit einer Oberklasse dieses Typs. So wechselt bei der Überdeckung der Haupt-VH aus dem obigen Beispiel auch der Typ des Feldes `mainBehaviour` von `MCB_Robot` auf `MCB_RobotCommunicating`. Dies erhöht die Lesbarkeit automatisch generierter Code-Dokumentationen, die in der Regel nur den Typ eines Feldes angeben, nicht aber den Typ eines diesem Feld eventuell direkt bei seiner Deklaration zugewiesenen Objektes. Solche Dokumentationen können beispielsweise mit dem Werkzeug Javadoc⁶⁰ erstellt werden. Die Einhaltung dieser Designvorschrift wird zur Laufzeit von MaxControl überprüft. Ist diese Vorschrift verletzt, so erzeugt MaxControl einen entsprechenden Laufzeitfehler.

Da MaxControl automatisch eine dynamische Bindung für Felder vornimmt, die eine SP oder eine VH eines Besitzers festlegen und referenzieren, wird im OT „`OT_RobotCommunicating`“² nur noch die neue Haupt-VH „`VH_RobotCommunicating`“² verwendet werden und nicht mehr die überdeckte Haupt-VH „`VH_Robot`“.

Erbt ein OT B von einem OT A, so soll also stets auch die Haupt-VH MA von OT A durch eine neue Haupt-VH MB in OT B ersetzt werden. Diese neue Haupt-VH MB soll dann von MA erben. Wie nach Kapitel 7.2.1 für Haupt-VHs üblich, soll dabei der Name der neuen erbenden Haupt-VH MB vom Namen des zugehörigen OTs B abgeleitet werden. Deshalb erhält im Beispiel der neue OT `MCD_RobotCommunicating`, der von `MCD_Robot` erbt, die neue Haupt-VH `MCB_RobotCommunicating`, die ihrerseits von `MCB_Robot` erbt. Genaueres zu dieser Namensableitung und den hierbei verwendeten Namenskonventionen wird in Kapitel 7.3 erläutert. So wird der Vererbungszusammenhang zwischen OT A und OT B auf deren Haupt-VHs übertragen. Auf diese Weise ist es möglich, MB zu verändern, ohne dass dies Auswirkungen auf MA hat. Dennoch werden alle SPs und Unter-VHs von MA auf MB vererbt, so dass man bei der Definition von MB auf MA aufbauen kann. Außerdem kann nun gezielt auch die neue Haupt-VH MB in anderen OTs und VHs wieder verwendet werden.

7.2.4 Einbinden von Simulationseigenschaften

Sowohl OTs als auch VHs können direkt SPs enthalten. Auch die Zuordnung der SPs innerhalb einer OT-Hierarchie sollte semantische Zusammenhänge widerspiegeln. Grundeigenschaften eines Objektes, die nicht direkt dessen Verhalten beeinflussen, sollten auch direkt dem zugehörigen OT zugeordnet werden.

Solche Grundeigenschaften basieren oft den Standard-Eigenschaften, die das 3D-Objekt bereits vor der Zuordnung eines Java-Typs im 3D-Animationsprogramm hat. Standard-SPs **müssen** direkt dem OT zugeordnet werden. Der Radius einer Kugel muss also direkt dem OT

⁶⁰ <http://java.sun.com/j2se/javadoc/>

zugeordnet sein, der für diese Kugel gewählt wurde, sofern der Radius als Standard-SP verwendet werden soll.

Dagegen sollte z.B. die Richtung, in die sich ein Objekt aufgrund seiner Navigationsverhaltensweise „VH_{Nav}“ bewegen möchte, auch der zugehörigen VH „VH_{Nav}“ zugeordnet werden, da diese SP weniger eine in der 3D-Szene akustisch oder visuell direkt erkennbare Eigenschaft des Objektes ist sondern lediglich auf eine VH wirkt oder von ihr verändert wird. Nur die indirekten Auswirkungen dieser SP sind in der 3D-Szene visuell erkennbar, beispielsweise als veränderte Bewegungsrichtung des Objektes.

VHs können ihre SPs verwenden, um Ergebnisse ihrer Berechnungen oder Entscheidungen anderen VHs zugänglich zu machen. Sowohl VHs, die zu demselben SO gehören, als auch VHs anderer SOs können diese SPs lesen und so deren Werte in ihre eigenen Berechnungen oder Entscheidungen einbinden. Die Navigationsverhaltensweise eines Roboters kann z.B. eine bestimmte Bewegungsrichtung für den Roboter festlegen und diese in einer ihrer SPs speichern. Eine andere VH dieses Roboters, welche die Bewegungen des Roboterrumpfes und die Bewegungen seiner Füße steuert, kann den Wert dieser SP auslesen und entsprechende Bewegungen errechnen. Ebenso kann ein Fahrzeug bei der Entscheidung, ob es einem anderen Fahrzeug ausweichen sollte, dessen Bewegungsrichtung und Geschwindigkeit aus entsprechenden SPs dieses Fahrzeugs lesen und diese Daten dann in die Entscheidungsfindung einfließen lassen.

Da SPs auch in die grafische Benutzeroberfläche des 3D-Animationsprogramms eingebunden werden und die meisten SPs auch animierbar sind, können die Verläufe ihrer Werte, die durch die Simulation erzeugt werden, nach der Simulation in der entstandenen Animation angezeigt und untersucht werden. So sind SPs auch für Entwickler von VHs nützlich, um z.B. Fehler zu finden oder die internen Abläufe einer VH im Verlauf einer Simulation genauer nachvollziehen zu können.

Umgekehrt können VHs ihre SPs auch dazu verwenden, sich von außen steuern zu lassen. Eine Lichtquelle des Typs `MCD_Light` hat in ihrer Unter-VH des Typs `MCB_LightSwitch` beispielsweise die boolesche SP „lightOn“, die sie nicht selbst verändert, sondern nur zur Entscheidung verwendet, ob das Licht ein- oder ausgeschaltet werden soll. Diese SP kann von außen gesetzt werden, ihr Wert kann also durch andere VHs desselben SOs oder durch VHs anderer SOs verändert werden. So könnte beispielsweise eine Ampel ihre verschiedenen Lichter ansteuern, indem sie deren jeweilige SP „lightOn“ mit entsprechenden Werten belegt. Ebenso können solche SPs in bestimmten Intervallen der Animation manuell kontrolliert werden, indem ihre MP dort auf „true“ gesetzt wird.

Mit einer solchen SP hat die VH `MCB_LightSwitch` eine einheitliche Schnittstelle, die sowohl von anderen VHs als auch von Benutzern verwendet werden kann, um die zugehörige Lichtquelle zu steuern.

VHs können sich gegenseitig auch steuern, indem sie Methoden in der jeweils anderen VH aufrufen, jedoch wird empfohlen, zur Kommunikation von VHs untereinander statt Methodenaufrufen lesenden und schreibenden Zugriff auf SPs zu verwenden, da diese Kommunikationsschnittstellen dann automatisch auch dem Benutzer für eine manuelle Steuerung zur Verfügung stehen.

Als Beispiel wird im folgenden Diagramm der OT „MCD_Vector“ erläutert. Die gestrichelten Pfeile zeigen wieder Vererbungen an. Die im tatsächlichen Programmcode verwendeten Namen für den OT, die VHs und die SPs, die auch im folgenden Text verwendet

werden, entsprechen hier den tiefgestellten Teilen der im Diagramm verwendeten Bezeichnungen und folgen den Präfixkonventionen aus Kapitel 7.3.

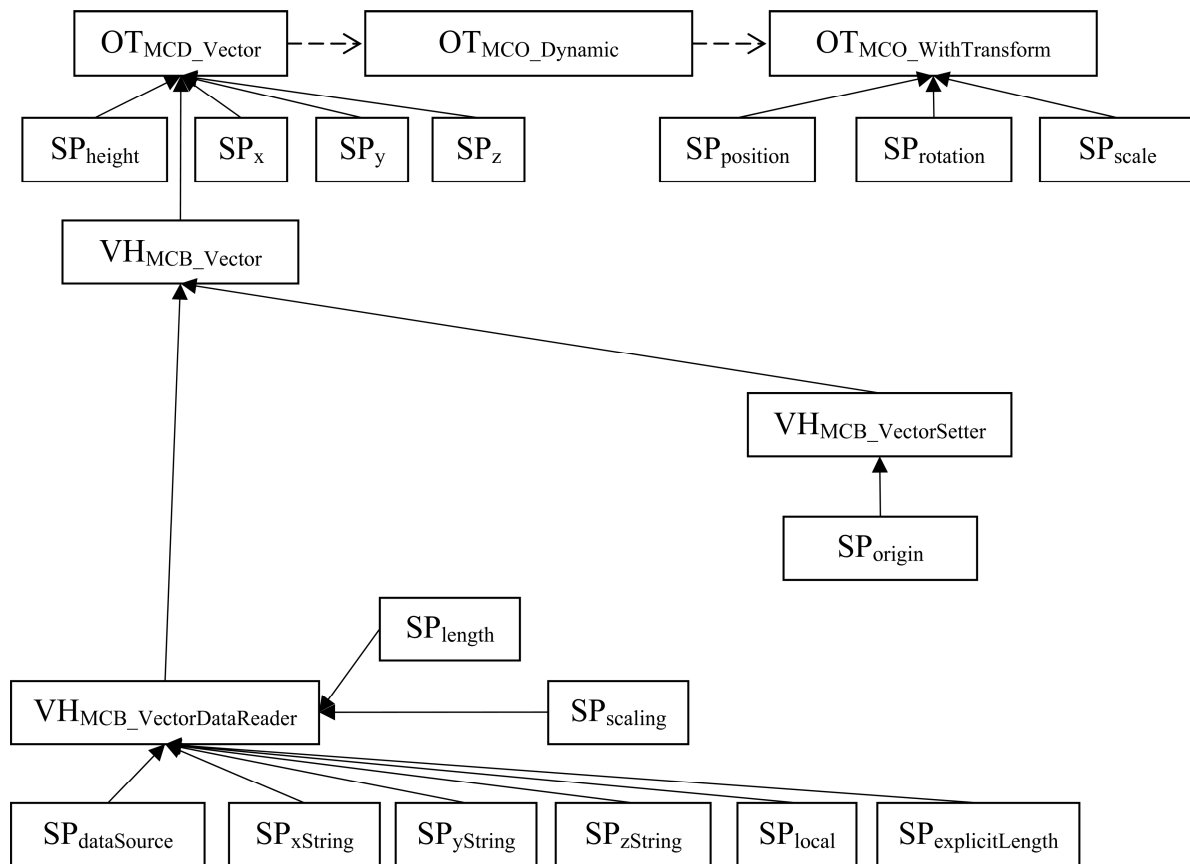


Abbildung 33: OT-Hierarchie zu „OT_{MCD_Vector}“

Der OT „MCD_Vector“ definiert das Verhalten eines dreidimensionalen Vektors, der dreidimensionale Daten in einer Szene illustriert. Er kann einem Zylinder-Objekt zugewiesen werden, das schon im 3D-Animationsprogramm die Form eines Zylinders und damit auch eine veränderbare Länge als Eigenschaft hat.

„MCD_Vector“ erbt zunächst indirekt von „MCO_WithTransform“ die Standard-SPs „position“, „rotation“ und „scale“, mit deren Hilfe ein Vektor seine Position, seine Lage im Raum bzw. seine Größe verändern kann.

Direkt dem OT zugeordnet sind die SPs „height“, „x“, „y“ und „z“. „height“ gibt die aktuelle Länge des Vektors an. Sie ist eine Standard-SP, so dass eine Änderung dieser SP direkt die Länge des Vektors ändert. Die SPs „x“, „y“ und „z“ geben den darzustellenden dreidimensionalen Wert des Vektors an, aus dem die Länge und die Ausrichtung „rotation“ des Vektors abgeleitet werden. Dieser Wert (x,y,z) wird in Weltkoordinaten und relativ zur Position des Vektors interpretiert, die in der Standard-SP „position“ gespeichert ist und den Ursprung des Vektors repräsentiert. Direkt dem OT sind also nur die SPs zugeordnet, welche unmittelbar die Eigenschaften des Vektors speichern, deren Werte von den VHs oder manuell bestimmt werden.

Die VH „MCB_VectorSetter“ bestimmt anhand von „x“, „y“ und „z“ in jedem Simulationszwischen schritt die Ausrichtung und die Länge des Vektors. Um auch die Position „position“ bestimmen zu können, kann in ihrer SP „origin“ eine Referenz auf ein 3D-Objekt gespeichert sein, dessen Mittelpunkt stets der Ursprung des Vektors sein soll. Wird ein

solches Objekt nicht angegeben, wird das Elternobjekt dieses Vektors in der Szenenhierarchie als Ursprung verwendet. Dies kann sinnvoll sein, um physikalische Daten wie die aktuelle Geschwindigkeit oder wirkende Kräfte für ein 3D-Objekt als Vektoren in der Animation zu zeigen, die dann üblicherweise ihren Ursprung im zugehörigen 3D-Objekt haben sollen. Die VH „MCB_VectorSetter“ verwendet also nur die SPs, die direkt dem OT zugeordnet sind zusammen mit ihrer eigenen SP „origin“, um den Vektor den Daten entsprechend zu positionieren, seine Länge zu bestimmen und ihn auszurichten.

Die VH „MCB_VectorDataReader“ ist dafür zuständig, die darzustellenden Daten aus einem anderen Objekt zu erhalten. Die hierfür zusätzlich erforderlichen Informationen werden daher in SPs dieser VH gespeichert. Die SP „dataSource“ enthält eine Referenz auf das Objekt, aus dem die Daten für den Vektor gelesen werden sollen. Wird ein solches Objekt nicht angegeben, wird wie bei der SP „origin“ stets das Elternobjekt dieses Vektors in der Szenenhierarchie als Datenquelle verwendet. Das Datenquellenobjekt wird im Folgenden als „D“ bezeichnet. Die SPs „xString“, „yString“ und „zString“ speichern Strings, welche die Bezeichner der Container-Felder der SPs im Datenquellenobjekt „D“ angeben, deren Werte auf die drei Wertekomponenten „x“, „y“ und „z“ dieses Vektors übertragen werden sollen. Wird in „xString“ beispielsweise der String „vx“ gespeichert, so wird der Wert für die X-Komponente dieses Vektors aus dem Datenquellenobjekt „D“ aus dessen SP „vx“ gelesen, wobei diese SP auch zu einer VH oder Unter-VH von „D“ gehören kann. Die boolesche SP „local“ gibt an, ob die so gelesenen Werte sich auf das Koordinatensystem des Datenquellenobjektes „D“ beziehen und deshalb in Weltkoordinaten umgerechnet werden müssen. Die VH „MCB_Physics“, welche 3D-Objekte besitzen müssen, wenn sie mechanisch simuliert werden sollen, speichert in ihren SPs „avX“, „avY“, „avZ“ und „avW“ z.B. die aktuelle Rotationsgeschwindigkeit und die Rotationsachse des zugehörigen 3D-Objektes, wobei diese Rotationsachse im lokalen Koordinatensystem des 3D-Objektes und nicht im Weltkoordinatensystem angegeben ist. Die boolesche SP „explicitLength“ gibt an, ob die Länge des Vektors explizit aus einer weiteren SP in „D“ gelesen werden soll, so dass die bisher gelesenen drei Komponenten nur die Ausrichtung des Vektors im Raum bestimmen, oder ob die Länge aus diesen drei Komponenten abgeleitet werden soll. Die SP „length“ speichert ggf. einen String, der den Bezeichner des Container-Feldes der SP aus „D“ angibt, aus der diese Länge explizit gelesen werden soll. Die Double-SP „scaling“ gibt nun noch einen Skalierungsfaktor für die Länge des Vektors an. So kann die Länge des Vektors auch für relativ kleine oder große Eingabewerte so skaliert werden, dass sie optisch sinnvolle Größenverhältnisse hat. Die VH „MCB_VectorDataReader“ berechnet anhand dieser Daten nun Werte für die SPs „x“, „y“ und „z“, die von „MCB_VectorSetter“ wie oben beschrieben auf die sichtbaren Eigenschaften des Vektors übertragen werden.

Durch manuelle Kontrolle seiner SPs können beliebige Eigenschaften des Vektors manuell bestimmt werden. So kann z.B. die in der SP „height“ gespeicherte Länge des Vektors in einigen Intervallen der Animation manuell bestimmt werden, auch wenn die gelesenen Werte aus „D“ eine andere Länge für diesen Vektor vorgeben. Die Ausrichtung des Vektors im Raum würde dabei weiterhin durch die VHs anhand der Werte aus „D“ bestimmt werden.

Die SPs von „MCB_VectorSetter“ und „MCB_VectorDataReader“ ändern sich in der Regel während der Simulation nicht. Sie können nur manuell oder durch VHs anderer 3D-Objekte, welche diesen Vektor von außen steuern, verändert werden. Sie haben dadurch eher den Charakter von Parametern, die einmal manuell bestimmt werden und dann in der Regel für den Rest der Simulation unverändert bleiben.

Wie SPs im Quellcode zu einem OT oder einer VH zugeordnet werden können, wird zu dem Konstruktor der Klasse `MC_Property` in Kapitel 8.2.1 detailliert erläutert.

7.3 Namenskonventionen für Klassenbezeichner und dabei relevante Basisklassen

In Anlehnung an den Namen `MaxControl`, sollen alle Klassen, die zentral an der Simulation beteiligt sind, durch das Präfix „MC_“ oder systematische Abwandlungen davon gekennzeichnet werden, so dass wichtige durch Vererbung entstehende Abhängigkeiten der Klassen schon aus dem Namen erkennbar werden. Im Folgenden werden die entsprechenden Namenskonventionen erläutert. Neue Klassen, die vom Benutzer entwickelt werden können, sollten sich möglichst an diese Konventionen halten, um eine übersichtliche Klassenstruktur zu gewährleisten.

Die Basisklasse aller SPs trägt den Namen „MC_Property“. Mit Ausnahme der direkten Erweiterung dieser Klasse „MC_PropertyCommOptimized“ sollen alle Klassen, die SPs repräsentieren, in Anlehnung an die Bezeichnung der Basisklasse „MC_Property“ durch das Namenspräfix „MCP_“ kenntlich gemacht werden. Dies gilt z.B. für die SP-Klassen „MCP_Double“ und „MCP_Boolean“. Aus ihren Bezeichnungen ist so bereits ableitbar, dass es sich dabei um Unterklassen der Klasse „MC_Property“ handelt, deren Datentypen ebenfalls aus ihren Bezeichnungen ableitbar sind. Ein Objekt des Typs „MCP_Double“ repräsentiert eine Double-SP, während ein Objekt des Typs „MCP_Boolean“ eine boolesche SP repräsentiert. Wenn eine solche Klasse eine Standard-SP repräsentieren soll, wird dem Namen dieser Klasse zusätzlich das Suffix „StandardProperty“ angehängt. So repräsentiert eine Instanz der Klasse „MCP_DoubleStandardProperty“ eine Standard-Double-SP. Ist eine SP nicht animierbar, so erhält sie das Suffix „NA“, ggf. hinter dem Suffix „StandardProperty“.

Sowohl Verhaltensweisen als auch Objekttypen können SPs und VHs besitzen, deshalb stammen ihre Klassen beide von der Basisklasse `MC_Owner` ab, deren Name das **Besitzen** von SPs und VHs bereits andeutet. Alle Verhaltensweisen erben von der Klasse „MC_Behaviour“, während alle Objekttypen Unterklassen von „MC_Object“ sind. Unterklassen dieser Basisklassen werden mit den Präfixen „MCB_“ bzw. „MCO_“ gekennzeichnet.

Eine weitere Basisklasse für viele Klassen ist „MCO_Dynamic“, die ihrerseits eine Unterklasse von „MC_Object“ ist. Ihre Instanzen repräsentieren 3D-Objekte in einer Szene, die vornehmlich ihren eigenen Zustand im Verlauf der Simulation ändern können, indem sie sich z.B. durch Änderungen ihrer Position, Rotation und Skalierung im Raum bewegen. Die Namen aller zugehörigen Unterklassen tragen das Präfix „MCD_“. Das „O“ aus „MCO_Dynamic“ wurde dabei entfernt, damit dieses Präfix nicht zu lang wird. Analog wurde auch bei anderen Präfixen vorgegangen, die eine Abstammung von Klassen anzeigen, deren Oberklassen selbst bereits durch ein eigenes Präfix kenntlich gemacht werden. Die Klasse „MCO_Universe“ repräsentiert Objekte in der Szene, die z.B. global gültige Parameter für die Simulation in ihren SPs speichern. Dies kann beispielsweise die Gravitation sein, die auf alle Objekte in der Szene wirkt, die mechanisch simulierte Körper repräsentieren. Sie können auch Verhaltensweisen haben, die global die Zustände vieler 3D-Objekte in der Szene ändern. Sie können zwar auch ihren eigenen Zustand ändern, dies ist in der Regel jedoch nicht die Hauptaufgabe eines Simulationsobjektes dieses Typs. Unterklassen dieser Klasse werden mit dem Präfix „MCU_“ kenntlich gemacht. Der Unterschied zwischen SOs des Typs „MCO_Dynamic“ und SOs des Typs „MCO_Universe“ liegt hauptsächlich darin, für welche SOs sie Daten speichern und welche SOs sie ggf. kontrollieren. SOs des Typs

„MCO_Dynamic“ sollten eher ein autonomes Verhalten zeigen und damit hauptsächlich ihren eigenen Zustand und weniger die Zustände anderer SOs ändern. Ebenso sollten ihre SPs eher Daten über das SO selbst und weniger über andere SOs speichern. SOs des Typs „MCO_Universe“ sollten dagegen eher globale Informationen über die Simulation (Windstärke, Gravitation) speichern und ggf. eher die Steuerung nicht autonomer SOs übernehmen, indem sie deren Zustände direkt ändern. Sie können auch das Verhalten autonomer SOs global beeinflussen, indem sie für diese Objekte zusätzlich zu deren eigenem Verhalten z.B. einen Rollwiderstand oder eine gegenseitige Anziehung durch Massengravitation erzeugen.

Wie bereits in Kapitel 7.2.1 erläutert wurde, hat jeder Objekttyp mit Verhaltensweisen genau eine Hauptverhaltensweise, die im Feld `mainBehaviour` referenziert wird. Die Typbezeichnung dieser Verhaltensweise soll aus der Typbezeichnung des Objekttyps abgeleitet sein. Unter Berücksichtigung der hier erläuterten Präfixkonventionen hat z.B. die Hauptverhaltensweise für den OT „MCD_Light“ den Typ „MCB_Light“. Das Präfix wird bei der Ableitung in der Regel also durch „MCB_“ ersetzt, während das Suffix unverändert bleibt. Da OTs meistens Subklassen von `MCO_Dynamic` sind, ist es vertretbar, in dem Präfix „MCB_“ der entsprechenden Haupt-VHs kein Hinweis mehr auf `MCO_Dynamic` zu belassen, da dieser OT den Normalfall darstellt. Dagegen sind OTs seltener Subklassen von „MCO_Universe“, so dass in den Präfixen der entsprechenden Haupt-VHs ein Hinweis auf „MCO_Universe“ verbleibt, indem hier das Präfix „MCBU_“ verwendet wird. So ist jede Hauptverhaltensweise leicht ihrem zugehörigen OT zuzuordnen, ebenso kann so zu jedem OT leicht seine Haupt-VH gefunden werden, wenn diese z.B. in einer Klassendokumentation aufgelistet werden.

Hilfsklassen, die von anderen Klassen verwendet werden, stammen von der Klasse „MC_Help“ ab, soweit dies technisch möglich und sinnvoll ist. In jedem Fall verwenden sie das Namenspräfix „MCH_“. Die Klasse „MCH_Transform3D“ (Kap. 10.1.3) kann beispielsweise Transformationen (siehe Kapitel 3.2) in Form einer Matrix speichern, so können Verhaltensweisen bequem mit Transformationen arbeiten und in „MCH_Transform3D“ definierte Operationen auf sie anwenden. Sie erbt jedoch nicht von „MC_Help“, sondern von der Klasse „`javax.media.j3d.Transform3D`“ aus dem Java-3D-API (siehe Kapitel 5.8). Dies ist sinnvoller, da diese Klasse aus dem Java-3D-API bereits viele nötige Datenstrukturen und Methoden zum Umgang mit Transformationen zur Verfügung stellt und durch „MCH_Transform3D“ nur um einige Felder und Methoden erweitert werden muss.

Um auch Materialeigenschaften, die nicht optischer Natur sind, in Verhaltensweisen berücksichtigen zu können, werden „Materialdeskriptoren“ verwendet. Diese erben von der Klasse „MC_MaterialDescriptor“ und werden durch das Präfix „MCMD_“ gekennzeichnet. Die Materialdeskriptoren verwenden Materialattribute, die von der Klasse „MC_MaterialAttribute“ abstammen und das Namenspräfix „MCMA_“ tragen. Um Materialien auf beliebig komplexe Weise auszuwerten und zu erkennen, können Unterklassen von „MC_MaterialEvaluator“ definiert und verwendet werden. Sie sind durch das Namenspräfix „MCME_“ gekennzeichnet.

Im folgenden Kapitel 8 zeigt Abbildung 34 die Hierarchie der wichtigsten Basisklassen in MaxControl, ein genauerer Überblick über die Klassenhierarchie von MaxControl befindet sich im Anhang, Kapitel 15. Dort ist auch die genaue Verwendung der hier angegebenen Präfixregeln bei den bisher implementierten Klassen erkennbar.

8 Grundaufbau wichtiger Klassen, Möglichkeiten für deren Nutzung und Erweiterung

In diesem Kapitel werden wichtige Methoden und Felder wichtiger Klassen erläutert. Dabei wird beschrieben, wie diese Klassen, ihre Felder und Methoden für die Erstellung neuer Klassen genutzt werden können. Insbesondere wird darauf hingewiesen, welche Methoden und Felder bei einer Erweiterung dieser Klassen eventuell durch angepasste Versionen überschrieben werden müssen.

Dabei wird nur auf konzeptionell notwendige Klassen, Felder und Methoden eingegangen. Weitere rein technisch notwendige Klassen, Felder und Methoden werden in diesem Zusammenhang in Kapitel 10.1 erläutert.

Zur Einführung ist es erforderlich, bereits vorher zusammenzustellen, welche Arten von Objekten Instanzen dieser Klassen repräsentieren sollen. Die Klasse `MC_Entity` ist eine gemeinsame Oberklasse vieler anderer in diesem Kapitel behandelte Klassen und stellt ihnen wichtige Felder und Methoden zur Verfügung. Instanzen von `MC_Property` repräsentieren SPs. `MC_Owner` ist eine Oberklasse für Verhaltensweisen und Objekttypen, da beide sowohl SPs als auch VHs „besitzen“ können. `MC_Object` ist die Basisklasse für alle Objekttypen, Instanzen dieser Klasse sind somit Simulationsobjekte. Instanzen von `MC_Behaviour` repräsentieren Verhaltensweisen. Die folgende Abbildung 34 gibt einen Überblick über die Hierarchie der wichtigsten Basisklassen in MaxControl, die zum großen Teil in den nächsten Kapiteln behandelt werden.

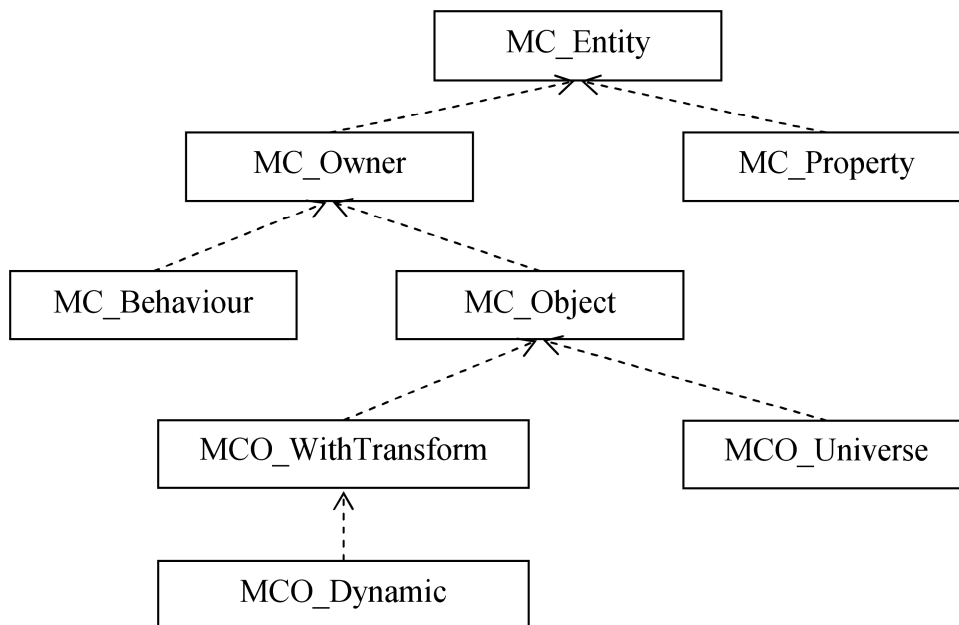


Abbildung 34: Überblick über die Hierarchie der wichtigsten Basisklassen in MaxControl

Die folgenden Kapitelüberschriften entsprechen jeweils dem Bezeichner der dort behandelte Klasse.

8.1 MC_Entity

Die Klasse `MC_Entity` ist eine gemeinsame Oberklasse für alle „nichttechnischen“ Klassen in MaxControl, die OTs, VHs und SPs repräsentieren. Direkte Unterklassen sind unter anderem `MC_Owner` und `MC_Property`, die in den folgenden Kapiteln näher erläutert

werden. Da die unten beschriebenen Klassen `MC_Behaviour` und `MC_Object` direkte Unterklassen von `MC_Owner` sind, sind auch diese Klassen Unterklassen von `MC_Entity`. Die Klasse `MC_Entity` stellt ihren Unterklassen wichtige Grundfunktionalitäten und Datenstrukturen zur Verfügung, die weitgehend von jeder dieser Klassen benötigt werden. Es gibt in `MaxControl` nur einige technische Klassen, die keine Unterklassen von `MC_Entity` sind, wie z.B. `MC_Help` und deren Unterklassen. Solche Klassen werden in Kapitel 10.1 erläutert. Einige wichtige Felder und Methoden der Klasse `MC_Entity` werden im Folgenden beschrieben:

8.1.1 Informationen über direkte und indirekte Besitzer

`MC_Owner owner:`

Dieses Feld referenziert für Instanzen der Klassen `MC_Behaviour` und `MC_Property` ihren jeweiligen „Besitzer“, eine Instanz der Klasse `MC_Owner`. Diese „Besitzer“ können sowohl Verhaltensweisen als auch Simulationsobjekte sein, also Instanzen der Klassen `MC_Behaviour` und `MC_Object`, da beide Klassen Unterklassen von `MC_Owner` sind. Verhaltensweisen können in beliebiger Tiefe ineinander verschachtelt werden.

`MC_Object rootOwner:`

Durch die oben genannte Verschachtelung von Verhaltensweisen ist es für VHs und SPs im Allgemeinen nicht mehr trivial, das Simulationsobjekt zu finden, zu dem sie gehören, da es möglicherweise nur indirekt ihr „Besitzer“ und dann nicht direkt in dem Feld „owner“ referenziert ist. Zu jedem Simulationsobjekt gibt es die schon vorher behandelte SO-Hierarchie, die aus VHs und SPs besteht und deren Wurzel das Simulationsobjekt ist. Dieses Simulationsobjekt, repräsentiert durch eine Instanz der Klasse `MC_Object`, ist für jede seiner VHs, Unter-VHs und SPs in dem Feld `rootOwner` referenziert.

8.1.2 Berücksichtigung und Steuerung des Zeitablaufs

`double getDeltaT():`

Diese Methode liefert das Zeitintervall in Sekunden, das in einem aktuellen Simulationszwischen schritt ablaufen soll. Dies ist ein global für alle Objekte gültiger Wert, der als Δt bezeichnet wird. Verhaltensweisen sollten diesen Wert als Basis für zeitliche Abläufe im spezifizierten Verhalten verwenden. Wird Δt dann im Verlauf der Simulation verändert, so können die Verhaltensweisen automatisch den neuen Δt -Wert berücksichtigen. Ein solcher globaler Wert sollte zu einer Instanz der Klasse `MCO_Universe` oder einer Unterklasse gehören, Δt ist indirekt in einer Instanz der Klasse `MCU_Time` in deren SP „timePerSecond“ gespeichert (siehe dazu auch Kap. 10.2.2). Ist in der Szene kein Objekt vorhanden, dem eine Instanz dieser Klasse zugewiesen wurde, so wird „timePerSecond“ intern in `MaxControl` gespeichert und die entsprechende Variable mit einem Standardwert belegt. Er kann dann im Verlauf der Simulation nicht mehr verändert werden. In beiden Fällen wird nicht direkt der Δt -Wert als Zeitintervalllänge pro Simulationszwischen schritt gespeichert, sondern es wird der Wert „timePerSecond“ als „simulierte Zeit pro Sekunde Film“ gespeichert. Dieser Wert gibt an, wie viel simulierte Zeit in jeder Sekunde der entstehenden Animation ablaufen soll. Ein Wert von 0,5 bedeutet also, dass in einer Sekunde der Animation nur eine halbe Sekunde der Simulation ablaufen soll. Durch Änderungen dieses Wertes sind leicht Zeitlupen- oder Zeitraffereffekte möglich.

Dieser Wert wird intern in Δt umgerechnet gemäß der Formel $\Delta t = (\text{timePerSecond} / \text{FrameRate}) / \text{simSubSteps}$. Die Variable `FrameRate` speichert die Anzahl von Frames pro Sekunde der Animation, die so genannte `Framerate`, und `simSubSteps` gibt die aktuelle Anzahl an Simulationszwischenritten je Simulationsschritt an. Der Wert `timePerSecond` ist unabhängig von der `Framerate` und der aktuellen Anzahl von Simulationszwischenritten je Simulationsschritt. Während der Wert von `FrameRate` konstant ist, können die Variablen `simSubSteps` und `timePerSecond` sowohl durch VHs als auch durch manuelle Kontrolle verändert werden und so auf Δt wirken (siehe dazu Kap. 10.2.2).

Ist der Wert `timePerSecond` durch ein Objekt A der Klasse `MCU_Time` in der Szene verfügbar, so kann er durch den Zugriff auf dieses Objekt A im Verlauf der Simulation verändert werden. Dies sollte in der Regel nur durch eine globale VH, die zu einer Unterklasse von `MCO_Universe` gehört, geschehen. Dabei wirkt eine durch VHs gesteuerte Änderung des Wertes `timePerSecond` nicht mehr im aktuellen Simulationsschritt, sondern der entsprechende Rückgabewert von `getDeltaT()` ändert sich erst direkt nach dem aktuellen Simulationsschritt und gilt für den nächsten Simulationsschritt, weil die oben angegebene Formel zur Berechnung von Δt durch `MaxControl` erst direkt nach dem aktuellen Simulationsschritt angewendet wird (siehe dazu Kap. 10.2.2). Alternativ kann der Wert im Verlauf der Simulation durch eine manuell festgelegte Animation für die SP „`timePerSecond`“ in Objekt A verändert werden. Gemäß der in Kapitel 6.6 definierten Behandlung von manuell kontrollierten SP-Werten gilt für einen in Frame n manuell kontrollierten Wert von „`timePerSecond`“ auch der daraus abgeleitete Wert Δt für das zugehörige Simulationsintervall zu Frame n.

Es kann nun durch Beeinflussung der SP „`timePerSecond`“ und damit durch indirekte Beeinflussung des Rückgabewertes Δt von `getDeltaT()` der Ablauf der Simulationszeit in der Animation verlangsamt oder beschleunigt werden.

Soll sich beispielsweise ein Objekt auf der X-Achse bei jedem Simulationszwischenritt mit einer bestimmten Geschwindigkeit bewegen, so könnte seine Verhaltensweise dies durch eine Programmzeile wie

```
x=x+v*getDeltaT();
```

erreichen. Dabei soll die Variable `x` für die aktuelle X-Koordinate stehen und `v` für die aktuelle Geschwindigkeit. Wie Verhaltensweisen genau zu implementieren sind und wie sie während der Simulation ausgeführt werden, wird in den Kapiteln 8.6 und 9 genauer behandelt. Hier kann man vereinfachend davon ausgehen, dass die oben angegebene Programmzeile bei jedem Simulationszwischenritt ausgeführt wird und die X-Koordinate des zugehörigen Objektes den Wert der Variablen `x` erhält. Wird nun das Zeitintervall Δt , das im aktuellen Simulationszwischenritt ablaufen soll, verändert, so ändert sich automatisch auch die Anzahl an Koordinateneinheiten, um die sich das Objekt im aktuellen Simulationszwischenritt weiterbewegt. Wird von `getDeltaT()` ein geringerer Wert zurückgeliefert, ergibt sich für alle SOs, deren VHs Δt entsprechend berücksichtigen, ein Zeitlupeneffekt, während eine Erhöhung zu einem Zeitraffer führt.

```
double getRealtimeDeltaT():
```

Diese Methode liefert einen Vergleichswert zum Rückgabewert von `getDeltaT()` zurück. Analog dazu ist sie indirekt in der SP „`realTimeTimePerSecond`“ einer Instanz der Klasse „`MCU_Time`“ gespeichert, sofern ein entsprechendes Objekt in der Szene existiert

(siehe dazu ebenfalls Kap. 10.2.2). Sonst ist auch dieser Wert als nicht veränderbarer Standardwert in MaxControl intern gespeichert. Er gibt den „Normalwert“ für die Zeit an, die in einer Sekunde der Animation abläuft, und soll im Verlauf einer Simulation konstant bleiben. Auch dieser Wert kann von 1 abweichen, wenn z.B. sehr schnelle Vorgänge wie atomare Reaktionen oder sehr langsame Vorgänge wie Planetenbewegungen in ständiger Zeitlupe bzw. ständigem Zeitraffer ablaufen sollen. Der Wert der SP „timePerSecond“ kann dann durch Abweichungen vom Wert „realTimeTimePerSecond“ diese „Normalzeit“ weiter verlangsamen oder beschleunigen.

Der Rückgabewert `simRealtimeDeltaT` von `getRealtimeDeltaT()` ergibt sich analog zum Rückgabewert von `getDeltaT()` aus der Formel
$$\text{simRealtimeDeltaT} = (\text{realTimeTimePerSecond} / \text{FrameRate}) / \text{simSubSteps}$$

Für diesen Wert gibt es verschiedene Anwendungsmöglichkeiten. Toneffekte verwenden diesen Wert zusammen mit dem Rückgabewert von `getDeltaT()`, um zu bestimmen, in welcher Tonhöhe ein Ton tatsächlich abgespielt werden muss. Die Tonhöhe wird für einen Toneffekt stets so angegeben, als würde die simulierte Zeit in der „Normalzeit“ ablaufen, die für den aktuellen Simulationszwischen schritt von `getRealtimeDeltaT()` zurückgegeben wird. Weicht nun der von `getDeltaT()` zurückgelieferte Wert davon ab, so kann die Tonhöhe dem sich ergebenden Zeitlupen- oder Zeitraffereffekt angepasst werden. Bei Zeitraffer werden damit automatisch alle Töne entsprechend höher und schneller abgespielt und bei Zeitlupe entsprechend tiefer und langsamer. Dies zeigt, wie weit reichend diese beiden globalen Werte die Simulation beeinflussen können. Eine weitere Anwendungsmöglichkeit für diesen Vergleichswert `simRealtimeDeltaT` ergibt sich für Objekte, die nicht unbedingt dem Zeitrahmen der Simulation folgen müssen. Eine simulierte Kamera soll sich eventuell gerade dann noch unverlangsamt weiterbewegen können, wenn die gesamte Simulation durch Zeitlupe verlangsamt oder sogar angehalten wurde. Sie kann sich dann weiterhin in Normalzeit bewegen, wenn sie statt des Rückgabewertes von `getDeltaT()` in ihren Verhaltensweisen den Rückgabewert von `getRealtimeDeltaT()` verwendet.

8.1.3 Zugriff auf die Simulationsobjekte einer Szene

Vector getScene():

Diese Methode liefert eine Instanz der Klasse „`java.util.Vector`“. Instanzen dieser Klasse repräsentieren eine beliebig lange Liste von Java-Objekten. Für eine gegebene 3D-Szene enthält die hier zurückgelieferte Liste stets alle Simulationsobjekte, die den 3D-Objekten aus der Szene zugeordnet wurden. Diese Liste ist die in Kapitel 9.2 erläuterte „Objektliste“. Sie ist geordnet, die Ordnung bestimmt die Reihenfolge, in der die Objekte in einem Simulationszwischen schritt nacheinander simuliert werden.

Der zurückgelieferte Objektliste kann von VHs verwendet werden, um bei der Simulation auf alle anderen Objekte einer Szene zuzugreifen. Beispielsweise kann ein simuliertes Fahrzeug so die Szene nach anderen Fahrzeugen durchsuchen, ihre Positionen auslesen und danach bestimmen, ob es einigen der anderen Fahrzeuge wegen zu geringen Abstandes ausweichen muss.

8.2 MC_Property

Objekte dieser Klasse speichern und verwalten SPs. Unterklassen von `MC_Property` werden implementiert, um diverse SP-Datentypen zu verwalten. In MaxControl sind bereits

mehrere solcher Unterklassen enthalten. So wurde `MCP_Double` für Double-SPs implementiert. Einige Unterklassen von `MC_Property` übernehmen aus technischen Gründen auch spezielle Aufgaben, auf das Beispiel `MCP_GeometryMessenger` wird in Kapitel 10.2.2 näher eingegangen. Weitere Unterklassen können bei Bedarf durch den Benutzer implementiert werden, um bisher nicht unterstützte SP-Datentypen zu ermöglichen.

8.2.1 Zuordnen einer Simulationseigenschaft zu ihrem Besitzer

`MC_Property(String _name, MC_Owner _owner) :`

Dies ist der Konstruktor von `MC_Property`. Ihm wird als erster Parameter der Name übergeben, mit dem die SP in der grafischen Benutzeroberfläche des verwendeten 3D-Animationsprogramms angezeigt werden soll. Bei einer Standard-SP muss dieser Name mit der entsprechenden Bezeichnung im ursprünglichen 3D-Objekt übereinstimmen, sonst würden bei einem Zugriffsversuch auf die Standard-SP Laufzeitfehler entstehen. Ebenso müssen die Standard-SP und die ursprüngliche Eigenschaft kompatible Datentypen haben. Bei einer nicht-standard SP kann dieser Name dagegen frei gewählt werden.

Der zweite Parameter gibt den „Besitzer“ an, zu dem die SP gehören soll. Eine Referenz auf diesen Besitzer wird in dem aus `MC_Entity` geerbten Feld `owner` gespeichert. Besitzer einer SP kann eine Instanz eines OTs oder einer VH sein.

Wie bereits in Kapitel 7.2.1 erläutert wurde, muss eine instanzierte SP X einem Container-Feld zugewiesen werden, das in der Klasse ihres Besitzers Y als `public`-Feld deklariert wurde. Dies ist erforderlich, damit die in Kapitel 8.3.1 beschriebene dynamische Bindung korrekt arbeitet. Damit die SP X noch vor der Initialisierung ihres „Besitzers“ Y instanziiert und ihrem zugehörigen Container-Feld in Y zugewiesen ist, ist es ebenfalls erforderlich, die Deklaration eines solchen referenzierenden Feldes, die Instanzierung der zugehörigen SP sowie deren Zuweisung an das Container-Feld in einer Anweisung durchzuführen.

Das folgende Codefragment demonstriert diese Vorgehensweise und ist der Beginn der Klassendefinition für den Objekttyp `MCD_Light` (siehe Abbildung 29 auf Seite 96), der die Standard-SP „`lightMultiplier`“ enthält:

```
public class MCD_Light extends MCO_Dynamic
{
    public final MCP_DoubleStandardProperty lightMultiplier=
        new MCP_DoubleStandardProperty("Multiplier",this);
    ...
}
```

Wie bereits in Kapitel 7.2.1 erläutert, werden SPs in diesem Text in der Regel mit dem **frei wählbaren** Namen ihres jeweiligen Container-Feldes bezeichnet. Aus der Definition der Klasse `MCD_Light` geht bereits klar hervor, dass jede Instanz dieser Klasse eine Standard-SP des Typs `MCP_DoubleStandardProperty` besitzt, die eine Verbindung zur ursprünglich im zugehörigen 3D-Objekt vorhandenen Eigenschaft „`Multiplier`“ aufbaut und im Feld `lightMultiplier` referenziert wird. Man kann also bereits bei der Definition von „`MCD_Light`“ sagen, dass dieser OT die Standard-SP „`lightMultiplier`“ besitzt.

Wird `MCD_Light` nun als Objekt A instanziiert, wird auch eine Instanz B der Klasse `MCP_DoubleStandardProperty` erzeugt. Dem Konstruktor dieser Instanz B wird als erster Parameter mit „`Multiplier`“ der Name übergeben, den diese Eigenschaft bereits vor der Zuordnung einer Java-Klasse im 3D-Animationsprogramm hat. Lichtquellen haben bereits

diese Eigenschaft, welche die Helligkeit der Lichtquelle bestimmt. Durch das Schlüsselwort „this“ wird dem Konstruktor als zweiter Parameter die Instanz A als ihr Besitzer übergeben. Die so instanzierte SP B wird in ihrem Besitzer A dem Feld lightMultiplier zugewiesen.

Wie bereits in Kapitel 7.2.3 erläutert wurde, ist es für MaxControl eine Konvention, jede SP nur einem public-Feld zuzuweisen, das auch genau den Typ der SP hat und keinen Obertyp, da dies mit Javadoc⁶¹ generierte Dokumentationen verbessert. Denn diese Dokumentationen geben nur den Deklarationstyp eines Feldes an aber nicht den Typ des Objektes, das diesem Feld zugewiesen wird. Der Typ der tatsächlich zugewiesenen SP wird in der generierten Dokumentation also nicht direkt angegeben sondern kann nur aus dem Typ des Feldes, dem sie zugewiesen wird, abgeleitet werden, weshalb hier kein Obertyp verwendet werden sollte.

Dem oben genannten Codefragment entspricht der folgende Teil der OT-Hierarchie zu MCD_Light:

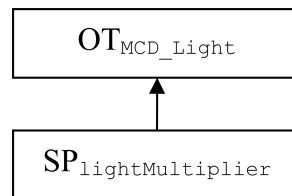


Abbildung 35: Teil-OT-Hierarchie zu MCD_Light

Das folgende Codefragment ist der Beginn der Klassendefinition für die Verhaltensweise „MCB_LightSwitch“ (siehe Abbildung 29 auf Seite 96), die drei Standard-SPs enthält und deren Diagramm in Abbildung 36 angegeben ist:

```
public class MCB_LightSwitch extends MC_Behaviour
{
    public final MCP_Double multOn =new MCP_Double ("Mult ON", this);
    public final MCP_Double multOff=new MCP_Double ("Mult OFF", this);
    public final MCP_Boolean lightOn=new MCP_Boolean("Light ON", this);
    ...
}
```

MCB_LightSwitch wird als Unterklasse von MC_Behaviour definiert. Sie ist als Steuerung der Lichthelligkeit für Lichtquellen des Objekttyps MCD_Light konzipiert. Sie enthält drei SPs. Die SPs „multOn“ und „multOff“ legen die gewünschte Helligkeit der Lichtquelle im eingeschalteten bzw. ausgeschalteten Zustand fest. Die SP „lightOn“ bestimmt, ob die Lichtquelle ein- oder ausgeschaltet sein soll. Je nach Zustand dieser SP überträgt die VH MCB_LightSwitch entweder den Wert von „multOn“ oder von „multOff“ auf die vorangehend erläuterte SP „lightMultiplier“ aus MCD_Light und ändert somit die Helligkeit der Lichtquelle entsprechend.

Wird die Klasse MCB_LightSwitch nun als Objekt A instanziiert, werden auch Instanzen für die drei SPs erzeugt. Ihren Konstruktoren wird mit dem ersten Parameter als String der Name übergeben, mit der diese hinzugefügten nicht-standard SPs in der grafischen Benutzeroberfläche des verwendeten 3D-Animationsprogramms bezeichnet werden sollen. Durch das Schlüsselwort „this“ wird den SPs wieder die Instanz A als ihr Besitzer übergeben.

⁶¹ <http://java.sun.com/j2se/javadoc/>

Die Reihenfolge, in der die SPs eines SOs „A“ in der grafischen Benutzeroberfläche des verwendeten 3D-Animationsprogramms für ein zugehöriges 3D-Objekt angezeigt werden, entspricht der Reihenfolge, in der die Container-Felder der SPs im Besitzer „A“ deklariert werden. So erscheint in der Benutzeroberfläche für ein 3D-Objekt, das in seiner zugehörigen SO-Hierarchie die VH „MCB_LightSwitch“ enthält, zuerst die SP „multOn“, dann die SP „multOff“ und danach die SP „lightOn“. Zusätzlich werden die SPs in der Benutzeroberfläche nach ihren zugehörigen Verhaltensweisen gruppiert.

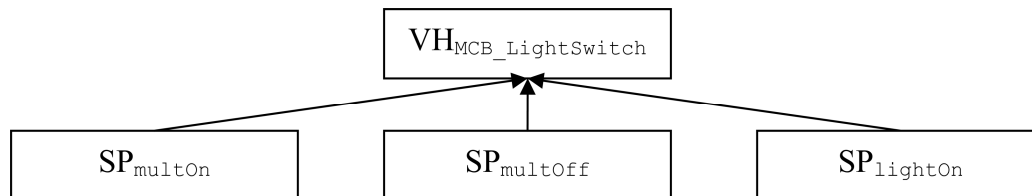


Abbildung 36: Ausschnitt einer OT-Hierarchie zu MCB_LightSwitch

8.2.2 Zugriff auf die Werte von Simulationseigenschaften

void setValue(Object v):

Über diese Methode kann im aktuellen MaxControl-Zustand ein Wert in eine SP gesetzt werden. Der Wert muss dabei als Java-Objekt (Parameter *v*) übergeben werden. Ein Wert des primitiven Typs `double` kann also nur eingeschlossen in ein Objekt übergeben werden, beispielsweise als Instanz der Klasse `Double`. Die Methode beachtet den Zustand der zugehörigen MP und entscheidet danach, ob der Wert wirklich in die SP übernommen wird. Ist dies der Fall, so wird der Wert weiter an die Methode `forceSetValue` übergeben. Da erst dort das Objekt, das den zu übergebenden Wert repräsentiert, weiter interpretiert wird, können alle Unterklassen von `MC_Property` die Methode `setValue(Object v)` in der Regel durch die Vererbung unverändert übernehmen.

void forceSetValue(Object v):

Erst hier muss das übergebene Objekt *v* wirklich interpretiert werden. Diese Methode sollte dann den durch *v* repräsentierten Wert in einer entsprechenden Variablen speichern. In der Klasse `MC_Property` enthält diese Methode keine Befehle, sie muss im Allgemeinen von jeder Unterklasse entsprechend implementiert werden. In der Unterklasse `MCP_Double` setzt `forceSetValue` voraus, dass *v* eine Instanz der Klasse `Double` ist und liest den in *v* enthaltenen Wert aus. Diesen Wert speichert die Methode dann im Feld `value`, das der Klasse `MCP_Double` hinzugefügt wurde und den primitiven Typ `double` hat. Analog sollten alle Klassen implementiert werden, die `MC_Property` erweitern.

Object getValue():

Diese Methode soll den im aktuellen MaxControl-Zustand in der SP gespeicherten Wert als Java-Objekt zurückliefern. Auch diese Methode muss im Allgemeinen von jeder Unterklasse entsprechend implementiert werden. In der Unterklasse `MCP_Double` liefert `getValue` den im Feld `value` gespeicherten Wert als Java-Objekt des Typs `Double` zurück. Analog sollten alle Klassen implementiert werden, die `MC_Property` erweitern.

8.3 MC_Owner

Diese Klasse repräsentiert „Besitzer“. Dies sind Objekte, die VHs und SPs besitzen können, also SOs und VHs. Die Klasse enthält Felder und Methoden, welche von den entsprechenden Unterklassen `MC_Object` bzw. `MC_Behaviour` verwendet werden können. Einige wichtige Felder und Methoden von `MC_Owner` werden hier erläutert.

8.3.1 Dynamische Bindung von SPs und VHs an ihren Besitzer

`Vector<MC_Property> properties:`

Dieses Feld referenziert in jeder Instanz A der Klasse `MC_Owner` eine Instanz der Klasse `Vector`. Diese enthält eine geordnete Liste aller SPs, die zu der betrachteten Instanz A der Klasse `MC_Owner` gehören. Über diese Liste können spezielle in Kapitel 8.3.3 erläuterte Methoden aus `MC_Owner` SPs finden und darauf zugreifen. Die Instanz A kann sowohl eine Instanz der Klasse `MC_Object` sein als auch eine Instanz der Klasse `MC_Behaviour`, da beide Klassen Unterklassen von `MC_Owner` sind. Es werden nur die SPs in diese Liste übernommen, die (wie in Kapitel 7.2.1 erläutert) in der betrachteten Instanz A in als `public` deklarierten Container-Feldern referenziert sind.

Diese Liste wird für jede Instanz A der Klasse `MC_Owner` so erstellt, dass dabei eine dynamische Bindung der Felder durchgeführt wird. Wenn ein Feld aus einer Oberklasse, das eine SP referenziert, in einer Unterklasse durch ein gleichnamiges Feld mit möglicherweise anderem SP-Datentyp überdeckt wird, so wird beim Erstellen der Liste für diese Unterklasse der Inhalt des überdeckenden Feldes in die Liste aufgenommen, der Inhalt des überdeckten Feldes wird dagegen nicht in die Liste eingefügt.

Jede SP speichert auch die Bezeichnung des Feldes, in dem sie referenziert wird. Wird nun anhand dieser Bezeichnung in der Liste eine SP gesucht, so werden bei gleichnamigen Feldern immer das in der Vererbungsfolge zuletzt definierte Feld und die darin referenzierte SP gefunden.

Zur Verdeutlichung seien hier zwei konstruierte Beispiele für VHs angegeben:

```
public class MCB_Special_A extends MC_Behaviour
{
    public final MCP_Double special=new MCP_Double ("special", this);
    ...
    public void test() {use(special)}
    public void listTest() {use(properties.firstElement();)}
}

public class MCB_Special_B extends MCB_Special_A
{
    public final MCP_Long special=new MCP_Long ("special", this);
    ...
    public void test2() {use(special)}
}
```

Es sei dabei angenommen, dass die Klasse `MCB_Special_B` keine weiteren Unterklassen hat. Außer „special“ seien keine weiteren SPs in den beiden Klassen oder deren Oberklassen definiert.

Hier erbt die Klasse `MCB_Special_B` von der Klasse `MCB_Special_A`, entsprechend wird das Feld `special` vom Typ `MCP_Double` aus der Klasse `MCB_Special_A` an die Klasse `MCB_Special_B` vererbt. Dort wird dieses Feld aber durch das gleichnamige Feld `special` überdeckt. Dieses neue Feld hat einen anderen Typ, nämlich `MCP_Long`, und referenziert gleichzeitig auch eine andere SP mit entsprechendem Typ, nämlich eine **neue** SP vom Typ `MCP_Long`. Ist ein Objekt eine Instanz der Klasse `MCB_Special_A` und dabei **keine** Unterklasse davon, so enthält seine im Feld `properties` referenzierte Liste von SPs nur die in der Klasse `MCB_Special_A` bei der Deklaration des Feldes `special` instanziierte SP des Typs `MCP_Double`. Ist ein Objekt dagegen eine Instanz der Unterklasse `MCB_Special_B`, so enthält seine im Feld `properties` referenzierte Liste von SPs nur die in der Klasse `MCB_Special_B` bei der Deklaration des Feldes `special` instanziierte SP des Typs `MCP_Long`.

Welches Feld bei Überdeckungen tatsächlich verwendet wird hängt auf diese Weise nicht mehr davon ab, wo in einer Klasse auf das Feld zugegriffen wird.

Es gelte folgende Zuweisung an ein Feld außerhalb der oben definierten Klassen:

```
MCB_Special_A a=new MCB_Special_B();
```

Das Feld `a` ist also vom Typ `MCB_Special_A`, referenziert aber eine Instanz der Klasse `MCB_Special_B`. Wird nun `a.test()` aufgerufen, so ist diese Methode in der Klasse `MCB_Special_A` zu finden. Sie übergibt der Methode `use` die SP, welche auch in der Klasse `MCB_Special_A` instanziiert wurde, da innerhalb des Methodenrumpfes von `test()` eine statische Bindung für das dort vorkommende Feld `special` gilt. Wird dagegen nun `a.test2()` aufgerufen, so ist diese Methode in der Klasse `MCB_Special_B` zu finden. Sie übergibt der Methode `use` die SP, welche auch in der Klasse `MCB_Special_B` instanziiert wurde. Auf welche SPs eine Methode bei Überdeckungen zugreift, hängt hier also davon ab, wo die Methode definiert wurde. Analoges gilt für externe Zugriffe. Wird auf `a.special` zugegriffen, so hängt es nicht vom Typ des in `a` referenzierten Objektes ab, auf welches Feld mit der Bezeichnung `special` zugegriffen wird, sondern allein vom deklarierten Typ der Variable `a`. Deshalb wird auf das Feld `special` aus der Klasse `MCB_Special_A` zugegriffen, obwohl das in `a` referenzierte Objekt den Typ `MCB_Special_B` hat.

Wird dagegen nun `a.listTest()` aufgerufen, so findet diese Methode durch die Verwendung der in `properties` referenzierten Liste von SPs auf jeden Fall die in der Klasse `MCB_Special_B` instanziierte SP, wobei angenommen wird, dass diese SP den Anfang der Liste bildet, so dass der Aufruf von `properties.firstElement()`, der das erste Element der Liste zurückliefert, genau diese SP zurückliefert. Diese Liste wird für jede Instanz von `MC_Owner` so erstellt, dass Inhalte **überschriebener** Felder aus allen **Oberklassen** dieser Instanz darin **nicht** enthalten sind.

Dadurch wird eine dynamische Bindung für Felder möglich, die SPs referenzieren. Dieselbe Technik wird bei der Erstellung der im Folgenden erläuterten Liste „behaviours“ auch für Felder verwendet, die auf VHs verweisen.

Meistens wird, sofern erwünscht, eine dynamische Bindung für Felder in Java dadurch erreicht, dass in der Klasse, die dieses Feld deklariert, spezielle Methoden für das Lesen und Schreiben des Inhaltes des Feldes hinzugefügt werden. Zu einer Felddeklaration „`double a;`“ werden dann die Methoden „`double getA()`“ und „`void setA(double a)`“ implementiert. Die Implementation dieser Methoden wird in jeder Unterklasse wiederholt, die

das Feld „a“ durch ein neues gleichnamiges Feld überdeckt. Da Java für Methoden automatisch eine dynamische Bindung durchführt, können die oben beschriebenen Zugriffsprobleme bei Feldern durch die ausschließliche Verwendung dieser Methoden bei jedem Zugriff auf das Feld a umgangen werden. Dies hat jedoch den Nachteil, dass diese Methoden in jeder Klasse wiederholt werden müssen, die ein entsprechendes Feld überschreibt. Bei vollständig manuell durchgeführter Programmierung, z.B. durch einen einfachen Texteditor, besteht die Gefahr, dass dem Programmierer dabei Fehler unterlaufen. Auch bei automatisierten Verfahren, wie sie in Java-Entwicklungsumgebungen zu finden sind, könnte eine dieser Methoden versehentlich manuell gelöscht werden, auch wenn sie vorher automatisch erzeugt wurde. Fehlt eine dieser Methoden, wird unter Umständen auf ein nicht gewünschtes Feld zugegriffen, was schwer zu analysierende Fehlersituationen hervorrufen könnte.

Ein weiterer Grund für die hier verwendete Lösung über Listen von Feldinhalten ist die Möglichkeit, eine SP oder auch eine VH innerhalb der hierarchischen Struktur von VHs und SPs in einem OT durch eine Baumsuche zu finden. So kann z.B. eine VH auch auf eine SP zugreifen, die **nicht** direkt durch ein Feld dieser VH referenziert wird, ohne dabei die genaue Position dieser SP innerhalb der Struktur des OTs kennen zu müssen. Diese Anwendungsmöglichkeit wird weiter unten in diesem Kapitel genauer beschrieben.

Vector<MC_Behaviour> behaviours:

Dieses Feld verweist analog zum Feld `properties` auf eine geordnete Liste aller VHs, die zu der betrachteten Instanz A der Klasse `MC_Owner` gehören. Über diese Liste können spezielle in Kapitel 8.3.3 erläuterte Methoden aus `MC_Owner` VHs finden und darauf zugreifen.

Analog zum Feld `properties` werden nur die VHs in diese Liste übernommen, die in so genannten Container-Feldern der Instanz A referenziert werden, die als `public` deklariert sind. Soll eine VH also direkt zu einem Simulationsobjekt oder als Unter-VH zu einer anderen VH gehören, so muss das sie referenzierende Container-Feld dort wie in Kapitel 7.2.1 erläutert entsprechend als `public` deklariert werden.

Wie schon bei der oben erläuterten Liste „`properties`“ wird für diese referenzierenden Felder bei der Erstellung der hier behandelten Liste „`behaviours`“ automatisch eine dynamische Bindung für die Felder durchgeführt.

In Kapitel 7.2.1 wurde bereits kurz zusammengefasst, wie VHs und SPs ihren Besitzern zugeordnet werden können. Um einer Unterklasse B von `MC_Owner` eine VH zuzuordnen, sollte eine Felddeklaration der folgenden Form verwendet werden:

```
public final MCB_CarAutoGear carAutoGear=new MCB_CarAutoGear(this);
```

Diese Deklaration legt fest, dass jede Instanz der Klasse B eine eigene VH des Typs `MCB_CarAutoGear` enthält, die im Feld `carAutoGear` referenziert wird.

Um einer Unterklasse B von `MC_Owner` analog dazu eine SP zuzuordnen, sollte eine Felddeklaration der folgenden Form verwendet werden:

```
public final MCP_Boolean forward=new MCP_Boolean ("Forward", this);
```

Diese Deklaration legt fest, dass jede Instanz der Klasse B eine eigene SP des Typs `MCP_Boolean` enthält, die im Feld `forward` referenziert wird. In der grafischen Benutzeroberfläche des verwendeten 3D-Animationsprogrammes wird diese SP mit dem

Namen „Forward“ bezeichnet, der dem Konstruktor der Klasse `MCP_Boolean` übergeben wird. Dieser Konstruktor wurde bereits in Kapitel 8.2.1 erläutert.

Dabei ist zu beachten, dass jede SP und auch jede VH jeweils nur genau einem public-Feld in genau einer Instanz von `MC_Owner` zugeordnet sein sollte. Denn eine **Wieder**verwendung von SP- und VH-**Instanzen** in eventuell mehreren als `public` deklarierten Feldern ist nicht sinnvoll und dies ist beim Design von Klassen für MaxControl nicht zulässig. In als `public` deklarierten Feldern dürfen nur VHs und SPs referenziert werden, die auch direkt zu dem entsprechenden OT oder der entsprechenden VH gehören sollten. Wie jedoch in Kapitel 7.2.1 bereits erläutert wurde, können OTs und VHs für einen vereinfachten Zugriff auch Referenzen auf VHs und SPs **anderer** VHs und OTs speichern, jedoch nur, wenn dafür **keine** als `public` deklarierten Felder verwendet werden.

8.3.2 Initialisierung von Verhaltensweisen und Festlegung von Standardwerten für Simulationseigenschaften

`void init():`

Diese Methode wird nach der Erstellung einer Instanz der Klasse `MC_Owner` und vor dem ersten Simulationsschritt ausgeführt. Diese Methode kann in Unterklassen von `MC_Object` und `MC_Behaviour` überschrieben werden, um dort nötige Initialisierungen festzulegen. Dies können Standardwerte für SPs sein, die verwendet werden, wenn einer SP bisher noch kein Wert zugewiesen wurde.

Im Folgenden werden einige Init-Methoden von Unterklassen von `MC_Owner` erläutert, um mögliche Anwendungen und auch erforderliche Designregeln anzugeben.

Dies ist die Init-Methode aus der Klasse `MCB_KITTSscannerLights`:

```
void init()
{
    super.init();
    setBooleanValue("scannerOn", true);
    setDoubleValue ("lightTimeOn", 0.3d);
    setDoubleValue ("timeFL", 1d);
}
```

Die Zeile `super.init();` ist bei jedem Überschreiben der Methode „`init()`“ als erster Befehl erforderlich, damit auch die Initialisierungen aus der überschriebenen Methode „`init()`“ aus `MC_Owner` durchgeführt werden. Bei Klassen, die nicht direkte Unterklassen von `MC_Owner` sind, garantiert diese Zeile auch, dass alle in den Überklassen definierten `init()`-Methoden vorher durchgeführt werden und so keine dort implementierte Initialisierung entfällt.

Der Befehl „`setBooleanValue("scannerOn", true);`“ setzt die boolesche SP „`scannerOn`“ auf den Wert `true`. Die genaue Funktionsweise dieses Befehls wird in Kapitel 8.3.5 genauer erklärt. Der Befehl „`setDoubleValue ("lightTimeOn", 0.3d);`“ setzt die Double-SP „`lightTimeOn`“ auf den Wert `0,3`. Der darauf folgende Befehl hat eine dazu analoge Semantik. Werden SPs in dieser Init-Methode Werte zugewiesen, so sind dies **nicht** notwendigerweise die Anfangswerte dieser SPs für die Simulation. Diese so genannten „Standardbelegungen“ werden nur dann als Werte

verwendet, wenn der betroffenen SP bisher noch kein Wert auf andere Weise zugewiesen wurde. Dies ist der Fall, wenn ein OT das erste Mal einem 3D-Objekt in der Szene zugewiesen wird. Die dabei dem 3D-Objekt neu hinzugefügten nicht-standard SPs werden dann mit den Werten belegt, die wie oben gezeigt in der `init`-Methode der zugehörigen VH bzw. des zugehörigen OTs definiert wurden. Auf diese Weise können also Standardbelegungen für alle SPs festgelegt werden.

Dies ist die `init`-Methode aus der Klasse `MCB_LightBlinking`:

```
void init()
{
    super.init();
    ((MCB_IntervalSwitcher)getBehaviour("intervalSwitcher")).setTarget("lightOn");
}
```

In der zweiten Befehlszeile der oben angegebenen `init`-Methode werden keine Standardbelegungen für SPs definiert, sondern es wird eine Verbindung zwischen einer VH und einer SP hergestellt. Dies ist ein weiteres Beispiel für die Anwendungsmöglichkeiten der `init`-Methode.

Wie bereits in Kapitel 7.1 beschrieben, ist die Verhaltensweise `MCB_IntervalSwitcher` universell einsetzbar für das periodische Ein- und Ausschalten von booleschen SPs. In der Klasse `MCB_LightBlinking` wird dem Feld `intervalSwitcher` eine Instanz dieser Verhaltensweise zugeordnet. Die weiter unten beschriebene Methode `getBehaviour` sucht im aktuellen Objekt ein Feld des Typs `MC_Behaviour` mit der als String angegebenen Bezeichnung und liefert ggf. dessen Inhalt zurück. In der zweiten Befehlszeile der oben angegebenen `init`-Methode wird so im aktuellen Objekt der Klasse `MCB_LightBlinking` das Feld `intervalSwitcher` gesucht und dessen Inhalt, eine Verhaltensweise des Typs `MCB_IntervalSwitcher`, zurückgeliefert. Da die Methode `getBehaviour` laut ihrer Signatur ein Objekt der Klasse `MC_Behaviour` zurückliefert, muss das Ergebnis hier zum Typ `MCB_IntervalSwitcher` gecastet werden, damit auf spezielle Methoden dieser Unterklasse von `MC_Behaviour` zugegriffen werden kann. Über die so verfügbare Methode `setTarget(String propertyName)` aus der Klasse `MCB_IntervalSwitcher` wird dieser VH der Name des Feldes „lightOn“ übergeben, in dem die SP referenziert wird, die zyklisch verändert werden soll. Die VH `MCB_IntervalSwitcher` erkennt dann selbst, wo sich eine SP mit der angegebenen Feldbezeichnung in dem OT befindet. Alternativ kann auch über die erweiterte Methode `setTarget(MC_Owner t, String propertyName)` neben der Feldbezeichnung der SP auch die Instanz der Klasse `MC_Owner` angegeben werden, in der die zu steuernde SP zu finden ist.

Auf diese Weise können Voreinstellungen an VHs vorgenommen werden, die vor der Simulation erforderlich sind.

8.3.3 Baumsuche in der zugehörigen Simulationsobjekt-Hierarchie nach Verhaltensweisen und Simulationseigenschaften

`MC_Behaviour getBehaviour(String name):`

Diese Methode sucht eine VH anhand der Bezeichnung des Container-Feldes, das in einem Besitzer auf sie verweist. Der Name des Feldes wird als String übergeben. Es wird dann eine Suchstrategie verfolgt, die auch in allen vergleichbaren Suchalgorithmen in der Klasse

MC_Owner analog angewendet wird. Wird eine entsprechende VH gefunden, so wird sie von dieser Methode zurückgegeben. Wird die gesuchte VH nicht gefunden, so wird ein Laufzeitfehler erzeugt.

Ein 3D-Objekt wird in MaxControl durch eine **Instanz** eines OTs, das zugehörige Simulationsobjekt (kurz SO), repräsentiert. Die zugehörigen VHs und SPs werden für dieses SO ebenfalls instanziiert, und bilden gemäß der zugrunde liegenden OT-Hierarchie die zugehörige SO-Hierarchie. Wenn im Folgenden die Suche nach einer VH oder einer SP in einem OT beschrieben wird, so bezieht sich dies stets auf die Ebene der Instanzen.

Die verwendete Suchstrategie ist eine Baumsuche, die in SO-Hierarchien den Zugriff auf Felder vereinfachen soll, die Instanzen von SPs oder VHs referenzieren. In einer SO-Hierarchie ist das SO selbst der Wurzelknoten, die Nicht-Wurzelknoten sind jeweils Instanzen von VHs oder SPs und die Kanten geben die „Besitzstruktur“ an. Das bedeutet, dass ein Elternknoten bei seinem Kindknoten im Feld `owner` eingetragen ist, das aus der Klasse `MC_Entity` auf SPs und VHs vererbt wird (siehe Kapitel 8.1.1). Ebenso sind Kindknoten in den in Kapitel 8.3.1 genannten Listen `properties` bzw. `behaviours` ihrer Elternknoten referenziert, abhängig davon, ob es sich bei dem Kindknoten um eine SP oder um eine VH handelt. SPs können nur Blattknoten sein, da sie selbst keine SPs oder VHs besitzen können.

Wird nun in einer VH oder in einem SO die Methode `getBehaviour` aufgerufen, so wird anhand des übergebenen Strings ein Feld gesucht, das auf eine VH verweist und dessen Bezeichnung mit diesem String übereinstimmt.

Die Suche findet in mehreren Einzelschritten statt, die für spätere Referenz nummeriert werden:

①:

Die Suche beginnt in der Instanz von `MC_Owner`, deren Methode `getBehaviour` aufgerufen wurde. Diese Instanz wird im Folgenden als „A“ bezeichnet. Zuerst wird der in „A“ referenzierte `Vector` „behaviours“ und damit die Liste der direkt zu „A“ gehörenden VHs nach der gewünschten VH durchsucht. Diese Liste wird im Folgenden als $[B_1, \dots, B_n]$ bezeichnet. Ob eine der VHs aus $[B_1, \dots, B_n]$ die gesuchte VH ist, kann leicht überprüft werden, da in jeder VH (und auch in jeder SP) die Bezeichnung des Feldes gespeichert ist, von dem diese VH (bzw. diese SP) seit ihrer Instanzierung referenziert wird. Dieses Feld wird sich immer in einer Instanz einer Unterklasse von `MC_Owner` befinden, also in einem SO oder in einer VH. Dabei ist zu beachten, dass im Allgemeinen jede SP und auch jede VH jeweils genau einem Container-Feld in genau einer Instanz von `MC_Owner` zugeordnet ist.

②:

Ist die gesuchte VH unter den in ① durchsuchten VHs nicht zu finden, wird die Suchstrategie rekursiv für fast alle VHs aus $[B_1, \dots, B_n]$ durchgeführt. Diese rekursive Suche ist vergleichbar mit einem Aufruf der Methode `getBehaviour` in den entsprechenden VHs aus $[B_1, \dots, B_n]$. Unter welchen Umständen eine dieser VHs nicht in die Rekursion einbezogen wird, ist weiter unten bei Schritt ③ erläutert.

Durch Schritt ① wird erreicht, dass nicht sofort rekursiv alle VHs aus $[B_1, \dots, B_n]$ mit ihrem gesamten jeweiligen Teilbaum durchsucht werden, sondern zunächst nur alle direkten VHs aus $[B_1, \dots, B_n]$ getestet werden. Eine bestimmte Bezeichnung für ein Feld kann zwar bis auf Überdeckung in jeder Klasse nur einmal auftreten, innerhalb der gesamten hier durchsuchten Baumstruktur kann sie jedoch mehrfach zu finden sein. Ist die gesuchte Feldbezeichnung

sowohl direkt in einer VH aus $[B_1, \dots, B_n]$ zu finden als auch indirekt in der Teilbaumstruktur einer VH aus $[B_1, \dots, B_n]$, so wird bei der hier implementierten Suchstrategie zuerst die entsprechende Feldbezeichnung in einer der direkten VHs aus $[B_1, \dots, B_n]$ gefunden, noch bevor diese VHs rekursiv tiefer durchsucht werden.

③:

Liefert keiner der rekursiven Aufrufe aus Schritt ② die gesuchte VH zurück und hat „A“ einen im Feld „owner“ referenzierten Besitzer, so wird die Methode `getBehaviour` in diesem Besitzer aufgerufen, sofern nicht der im nächsten Absatz beschriebene Fall vorliegt. Dieser Aufruf signalisiert dem Besitzer zusätzlich, dass er zunächst sich selbst daraufhin überprüfen soll, ob er die gesuchte VH ist, noch bevor er beginnend in Schritt ① seine eigenen Unterverhaltensweisen überprüft.

Bei diesem Schritt ③ wäre es möglich, dass die Methode `getBehaviour` jetzt in einem Besitzer aufgerufen wird, der selbst bereits bei einer rekursiven Suche in seinem Suchschritt ② die Methode `getBehaviour` in „A“ aufgerufen hat. Dann wurde diese Methode nicht ursprünglich in „A“ aufgerufen, sondern der Aufruf dieser Methode in „A“ findet nur im Rahmen einer Rekursion statt, die von einer anderen Instanz der Klasse `MC_Owner` ausging.

Damit in einem solchen Fall die Schleife terminiert, wird dieser Fall in dem oben genannten Schritt ② und auch bei diesem Schritt ③ erkannt und das Problem verhindert.

Das Objekt „A“ wendet in Schritt ② die Rekursion nur auf VHs an, die nicht selbst die Methode `getBehaviour` rekursiv in diesem Objekt „A“ aufgerufen haben. Die in Schritt ② rekursiv durchsuchten VHs können die Methode `getBehaviour` in diesem Objekt „A“ nur in ihrem Schritt ③ aufgerufen haben, da nur in diesem Schritt die Suche in ihrem jeweiligen „Besitzer“ „A“ fortgeführt werden kann. Solche VHs haben bereits in ihren Schritten ① und ② rekursiv ihre eigenen Unter-VHs überprüft, weshalb eine weitere rekursive Suche in diesen VHs zu einer nicht terminierenden Schleife führen würde.

Auch der letzte Schritt ③ führt die Rekursion nur beim „Besitzer“ von Objekt „A“ weiter, wenn dieser nicht bereits selbst eine Rekursion durchführt und dabei die Methode `getBehaviour` in diesem Objekt „A“ aufgerufen hat. In diesem Fall überprüft der „Besitzer“ bereits selbst seine Unter-VHs und darf kein weiteres Mal rekursiv durchlaufen werden. Der Besitzer „B“ von „A“ könnte allerdings selbst die gesuchte VH sein, wenn er nicht das SO und damit der Wurzelknoten der SO-Hierarchie ist. Ein solcher Besitzer „B“, der eine VH und **nicht** das SO ist, wird aber in jedem Fall entweder noch daraufhin überprüft oder er wurde bereits daraufhin überprüft, ob er selbst die gesuchte VH ist. Um dies zu zeigen sind drei mögliche Fälle zu betrachten. „B“ könnte erstens die VH sein, in der die Methode `getBehaviour` ursprünglich, also nicht im Rahmen einer Rekursion, aufgerufen wurde. Es sei nun angenommen, dass die gesuchte VH nicht zu den direkten und indirekten Unter-VHs von „B“ gehört, anderenfalls würde die bei „B“ beginnende Suche diese VH auch finden und dann terminieren⁶², damit wäre es dann auch nicht mehr nötig, „B“ selbst zu testen. Hat die VH „B“ nun ihre Unter-VHs erfolglos durchsucht, führt sie die Rekursion in ihrem eigenen Besitzer fort. Da „B“ eine VH ist, gibt es auf jeden Fall einen Besitzer für „B“, da jede VH entweder Unter-VH einer anderen VH ist oder das SO ihr direkter Besitzer ist. Dieser Besitzer „C“ von „B“ wird nun in seinem Schritt ① seine Unter-VHs überprüfen. Im folgenden Absatz wird beschrieben, welche Unter-VHs dabei unter Umständen nicht überprüft werden. „B“

⁶² Alle hier beschriebenen Schleifen werden abgebrochen, sobald die gesuchte VH gefunden ist.

wäre ein Kandidat dafür, nicht überprüft zu werden, weil `getBehaviour` von „B“ in ihrem Besitzer „C“ aufgerufen wurde. Da „B“ jedoch die VH ist, in der die Methode `getBehaviour` ursprünglich aufgerufen wurde, gilt ein im folgenden Absatz beschriebener Sonderfall und „B“ wird dennoch überprüft. Ein weiterer Fall wäre, dass „B“ von ihrem Besitzer „C“ ausgehend rekursiv durchsucht wird. Im vorangehenden Absatz wurde erläutert, dass diese Rekursion nur dann von „C“ ausgehend durchgeführt wird, wenn „B“ die Rekursion nicht vorher schon selbst in „C“ weitergeführt hat. Da hier aber angenommen wird, dass „B“ trotzdem von ihrem Besitzer „C“ ausgehend rekursiv durchsucht wird, hat „B“ die Rekursion **nicht** vorher schon selbst in „C“ weitergeführt. Nach den Ausführungen im folgenden Abschnitt bedeutet dies, dass „B“ dann im Schritt ① von „C“ bereits daraufhin überprüft wurde, ob sie die gesuchte VH ist, da sie nicht die Kriterien erfüllt, um bei dieser Überprüfung übersprungen zu werden. Der dritte mögliche Fall ist, dass die Rekursion in „B“ von einer ihrer Unter-VHs in deren Schritt ③ ausging. Wie zu Schritt ③ erläutert wurde, wird „B“ bei einem solchen Aufruf die Anweisung gegeben, zunächst sich selbst daraufhin zu testen, ob sie die gesuchte VH ist, noch bevor in „B“ der Schritt ① ausgeführt wird. Auch in diesem Fall wurde „B“ selbst also vorher bereits überprüft.

Um redundante Tests zu vermeiden, werden in Schritt ① nur Unter-VHs überprüft, die **nicht** selbst in ihrem eigenen Suchschritt ③ die Methode `getBehaviour` in diesem Objekt „A“ aufgerufen haben, es sei denn, es handelt sich bei einer solchen Unter-VH um die VH, in der die Methode `getBehaviour` ursprünglich, also nicht im Rahmen einer Rekursion, aufgerufen wurde.

Um zu zeigen, dass der Test in einem solchen Fall tatsächlich entfallen kann, sei nun eine Unter-VH „B“ betrachtet, die selbst in ihrem eigenen Suchschritt ③ die Methode `getBehaviour` in diesem Objekt „A“ aufgerufen hat und in der die Methode `getBehaviour` nicht ursprünglich aufgerufen wurde. Für solche VHs ist nun zu entscheiden, ob der Test dieser VH in Schritt ① entfallen kann, da er bereits an anderer Stelle durchgeführt wurde, oder ob er dennoch erforderlich ist. Wenn die Rekursion nicht ursprünglich in „B“ gestartet wurde, sind zwei Fälle zu betrachten. Die Rekursion könnte von „A“ ausgehend in „B“ weitergeführt worden sein, die Bedingungen in Schritt ③ von „B“ hätten dann aber verhindert, dass „B“ die Rekursion zurück zu „A“ führt, indem sie die Methode `getBehaviour` in diesem Objekt „A“ aufruft. Es kann also nur der zweite Fall zutreffen, dass `getBehaviour` in „B“ von einer ihrer eigenen Unter-VHs in deren Schritt ③ aufgerufen wurde. In diesem Fall wurde „B“ dabei die Anweisung gegeben, sich zunächst selbst daraufhin zu testen, ob „B“ die gesuchte VH ist. Ein weiterer entsprechender Test von „B“ wäre hier also redundant und kann entfallen. Handelt es sich bei „B“ dagegen um die VH, in der die Methode `getBehaviour` ursprünglich aufgerufen wurde, so ist diese VH selbst noch nicht daraufhin getestet worden, ob sie die gesuchte VH ist. **Dieser Sonderfall wird für spätere Referenzen als α bezeichnet.** Der Test, ob „B“ selbst die gesuchte VH ist, kann in diesem Fall nur in Schritt ① von „A“ gemacht werden. Es ist nicht möglich, dass eine Unter-VH „C“ von „B“ in ihrem Schritt ③ die Rekursion in „B“ weiterführt und dabei an „B“ die Anweisung gibt, sich zunächst selbst zu überprüfen. Da „B“ die VH ist, in der die Rekursion begonnen hat, könnte die Rekursion nur von „B“ ausgehend in „C“ weitergeführt werden. Nach den Erläuterungen zu Schritt ③ ist danach ein Weiterführen der Rekursion von „C“ zurück nach „B“ aber nicht mehr möglich, da dann eine nicht terminierende Schleife entstehen würde. „B“ kann also von keiner Unter-VH „C“ in deren Schritt ③ die Anweisung erhalten, sich selbst zu überprüfen. Daher kann nur in Schritt ① von „A“ überprüft werden, ob

„B“ die gesuchte VH ist. Dieser Prüfschritt kann also nicht entfallen, weshalb die entsprechende Bedingung für Schritt ① entsprechend erweitert wurde.

Insgesamt werden so redundante Suchvorgänge und nicht-terminierende Ketten von Methodenaufrufen vermieden.

Als weiteres Detail ist anzumerken, dass alle oben beschriebenen Schleifen abgebrochen werden, sobald die gesuchte VH gefunden ist.

Anhand eines Beispiels für eine SO-Hierarchie soll die Wirkungsweise dieses Suchalgorithmus verdeutlicht werden. Gegeben seien „SO₁“, die Haupt-Verhaltensweise „VH_{OT1}“, die Unter-VHs „VH₁“ bis „VH₁₂“ und die SPs „SP₁“ bis „SP₆“. Die Pfeile im Diagramm geben wieder die „Besitzhierarchie“ an. In der Verhaltensweise „VH₁₀“ werde nun die Methode `getBehaviour` aufgerufen, um die Verhaltensweise „VH₉“ anhand der Bezeichnung ihres Container-Feldes zu finden. Die einzelnen Knoten der Hierarchie sind in der Reihenfolge nummeriert, in der sie durch den Algorithmus daraufhin überprüft werden, ob sie die gesuchte VH repräsentieren. Diese Reihenfolge ist an den Knoten durch elliptisch umrahmte Nummern angegeben.

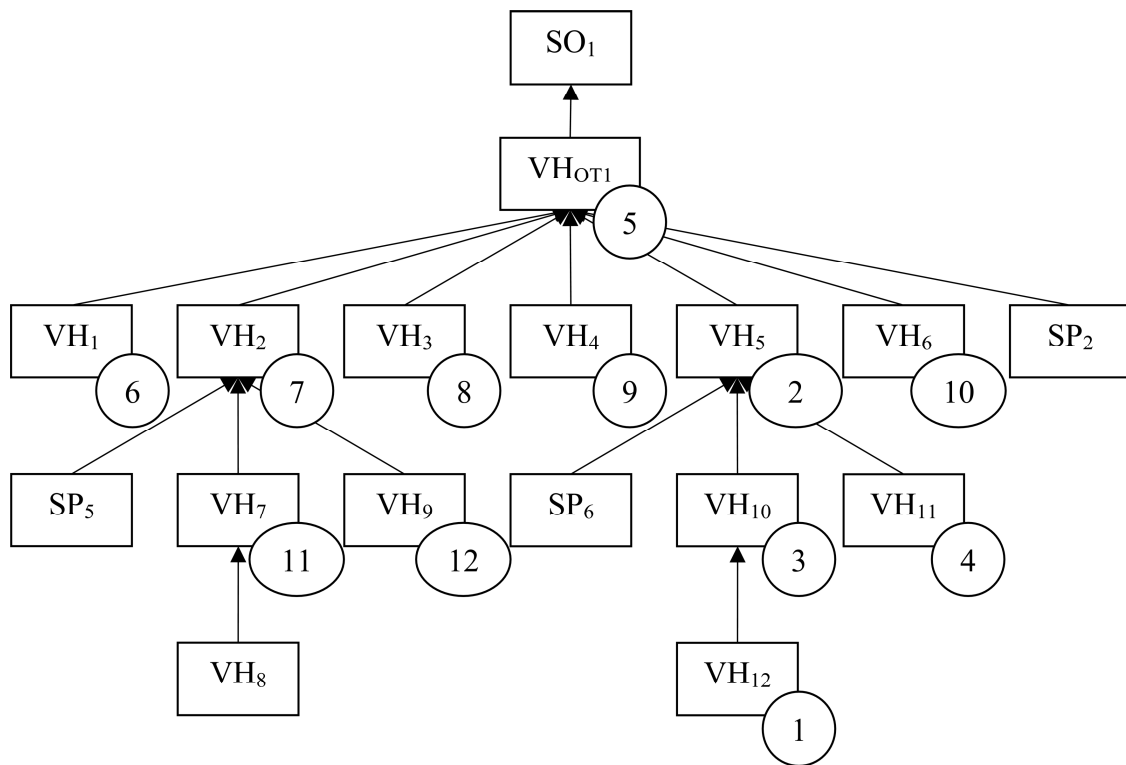


Abbildung 37: Beispiel einer Baumsuche nach einer VH

Die einzelnen Prüfschritte werden nun anhand der ihnen im obigen Diagramm zugeordneten Nummern genauer erläutert:

1:

Da die Methode `getBehaviour` in VH₁₀ aufgerufen wird, werden alle Unter-VHs von VH₁₀ überprüft. Dabei wird zunächst in Schritt ① nur die Liste der direkten Unter-VHs durchsucht, d.h. (VH₁₂). In dem folgenden Schritt ② startet VH₁₂ selbst einen Aufruf von `getBehaviour`. Es existieren jedoch keine Unter-VHs von VH₁₂, die überprüft werden könnten. Der Aufruf von `getBehaviour` in VH₁₂ ruft auch nicht im letzten Schritt ③ die

gleiche Methode im „Besitzer“ VH_{10} auf, da der Aufruf von `getBehaviour` in VH_{12} bereits von VH_{10} ausging und so eine nicht terminierende Schleife verhindert wird.

2:

Da in VH_{10} die gesuchte VH nicht gefunden werden konnte, wird nun im letzten Schritt ③, den `getBehaviour` in VH_{10} ausführt, die Rekursion weiter zu VH_5 geführt. Dabei wird VH_5 angewiesen, zunächst sich selbst zu überprüfen.

3-4:

Nun werden alle Unter- VH s von VH_5 in Schritt ① überprüft. Da diese Suche **ursprünglich** in VH_{10} gestartet wurde, trifft der oben erläuterte Sonderfall α zu. Statt die Überprüfung von VH_{10} entfallen zu lassen, wird daher neben VH_{11} auch VH_{10} direkt daraufhin überprüft, ob sie die gesuchte VH ist. In Schritt ② werden jetzt alle Unter- VH s von VH_5 rekursiv durchsucht, die nicht selbst die Methode `getBehaviour` in VH_5 aufgerufen haben. Deshalb wird nur VH_{11} rekursiv durchsucht. Da VH_{11} keine Unter- VH s besitzt, wird dort lediglich in Schritt ③ entschieden, ob die Rekursion in VH_5 fortgeführt wird. Da `getBehaviour` in VH_{11} von VH_5 aufgerufen wurde, wird die Rekursion nicht von VH_{11} zurück nach VH_5 geführt, was eine nicht terminierende Schleife verhindert.

5:

Da in VH_5 die gesuchte VH nicht gefunden werden konnte, wird nun im letzten Schritt ③, den `getBehaviour` in VH_5 ausführt, die Rekursion weiter zu VH_{OT1} geführt. Dabei wird VH_{OT1} angewiesen, zunächst sich selbst zu überprüfen. Dieser Test fällt negativ aus, da VH_{OT1} nicht die gesuchte VH ist.

6-10:

Nun werden alle Unter- VH s von VH_{OT1} in Schritt ① überprüft, die nicht selbst die Methode `getBehaviour` in VH_{OT1} aufgerufen haben. VH_5 wird daher nicht erneut überprüft, auch weil die Rekursion nicht ursprünglich in VH_5 gestartet wurde, so dass auch der oben genannte Sonderfall α **nicht** zutrifft. In Schritt ② werden danach alle Unter- VH s von VH_{OT1} rekursiv durchsucht, die nicht selbst die Methode `getBehaviour` in VH_{OT1} aufgerufen haben. Deshalb wird VH_5 auch nicht rekursiv durchsucht.

Da VH_1 , VH_3 , VH_4 und VH_6 keine Unter- VH s besitzen, wird dort jeweils lediglich in Schritt ③ entschieden, ob die Rekursion in VH_{OT1} fortgeführt wird. Da `getBehaviour` in VH_1 , VH_3 , VH_4 und VH_6 jeweils von VH_{OT1} aufgerufen wurde, wird die Rekursion nicht wieder zurück nach VH_{OT1} geführt, was eine nicht terminierende Schleife verhindert.

11-12:

Von den übrigen direkten Unter- VH s von VH_{OT1} besitzt nur VH_2 weitere Unter- VH s. Dort werden zuerst in Schritt ① nur alle direkten Unter- VH s von VH_2 überprüft. Dabei werden VH_7 und VH_9 getestet. Der Test von VH_9 fällt positiv aus, unter der Annahme, dass dies die gesuchte VH ist und die korrekte Bezeichnung des Feldes, das in VH_2 auf VH_9 verweist, dem ursprünglichen Aufruf von `getBehaviour` in VH_{10} als String übergeben wurde. Ferner darf die übergebene Bezeichnung des Feldes im Verlauf dieser Suche nicht schon vorher als Bezeichner eines anderen Feldes gefunden werden, das auf eine der übrigen getesteten VH s verweist. Auf diese erforderliche Eindeutigkeit wird weiter unten nach den Erläuterungen zu der Methode `getPropertyObjectOfType` näher eingegangen. Die aktuell in VH_2 ausgeführte Methode `getBehaviour` bricht daraufhin ab und VH_9 wird als Ergebnis dieser Methode an die aufrufende Methode zurückgegeben. Die aufrufende Methode bricht dann ebenfalls ab und übergibt das übernommene Ergebnis weiter nach oben in der rekursiven

Aufrufstruktur von `getBehaviour`. Schließlich gibt die in `VH10` aufgerufene Methode `getBehaviour` die gesuchte `VH` „`VH9`“ als Ergebnis zurück.

MC_Behaviour getPropertyObject(String name):

Diese Methode sucht analog zu `getBehaviour(String name)` eine **SP** anhand der Bezeichnung des `public`-Feldes, das in einer `VH` oder in einem `SO` auf sie verweist. Der Name des Feldes wird als `String` übergeben. Es wird nun eine Suchstrategie verfolgt, die analog zu dem in `getBehaviour` implementierten Suchalgorithmus verläuft. Wird eine entsprechende `SP` gefunden, so wird sie von dieser Methode zurückgegeben. Mit den Methoden dieser `SP` kann man dann beispielsweise auf deren Werte zugreifen. Wird die gesuchte `SP` nicht gefunden, so wird der Laufzeitfehler `MCE_MC_PropertyNotFoundException` erzeugt, der von einem `try-catch`-Konstrukt abgefangen werden kann.

Wird nun in einer `VH` oder in einem `SO` die Methode `getPropertyObject` aufgerufen, so wird anhand des übergebenen `String` ein Feld gesucht, das auf eine `SP` verweist und dessen Bezeichnung mit diesem `String` übereinstimmt.

Auch diese Suche findet in mehreren Einzelschritten statt, die für spätere Referenz nummeriert werden. Das Objekt, in dem `getPropertyObject` ursprünglich aufgerufen wird, erhält hier zur Vereinfachung wieder die Bezeichnung „`A`“.

①:

Zuerst wird der in „`A`“ referenzierte `Vector` „`properties`“ durchsucht. Dadurch wird die Liste der direkt zu „`A`“ gehörenden `SPs` nach der gewünschten `SP` durchsucht. Die geordnete Liste dieser dabei durchsuchten `SPs` wird im Folgenden als $[S_1, \dots, S_n]$ bezeichnet. Ob eine der `SPs` aus $[S_1, \dots, S_n]$ die gesuchte `SP` ist, kann leicht überprüft werden, da in jeder `SP` die Bezeichnung des Feldes gespeichert ist, von dem diese `SP` referenziert wird.

②:

Ist die gesuchte `SP` unter diesen `SPs` nicht zu finden, wird die hier beschriebene Suchstrategie rekursiv für fast alle direkt zu „`A`“ gehörenden `VHs` durchgeführt. Unter welchen Umständen eine dieser `VHs` nicht in die Rekursion einbezogen wird, wird weiter unten bei Schritt ③ erläutert. Die Liste dieser `VHs` wird im Folgenden als $[B_1, \dots, B_n]$ bezeichnet. Diese rekursive Suche ist vergleichbar mit einem Aufruf der Methode `getPropertyObject` in den entsprechenden `VHs` aus $[B_1, \dots, B_n]$.

③:

Liefert keiner der rekursiven Aufrufe aus Schritt ② die gesuchte `VH` zurück und hat „`A`“ einen im Feld „`owner`“ referenzierten „Besitzer“, so wird die Methode `getPropertyObject` rekursiv in diesem Besitzer aufgerufen, sofern nicht der im Folgenden erläuterte Fall vorliegt. Denn bei diesem Schritt ist es möglich, dass ein Besitzer aufgerufen wird, der selbst bereits bei einer rekursiven Suche in seinem Suchschritt ② die Methode `getPropertyObject` in „`A`“ aufgerufen hat. Dann wurde diese Methode nicht ursprünglich in „`A`“ aufgerufen, sondern der Aufruf dieser Methode in „`A`“ findet nur im Rahmen einer Rekursion statt, die von einer anderen Instanz der Klasse `MC_Owner` ausging.

Damit in einem solchen Fall keine nicht-terminierende Schleife entsteht, wird dieser Fall in dem oben genannten Schritt ② und auch bei diesem Schritt ③ erkannt und das Problem verhindert.

In Schritt ② wird die Rekursion nur für VHs durchgeführt, die **nicht** selbst in ihrem eigenen Suchschritt ③ die Methode `getPropertyObject` rekursiv in diesem Objekt „A“ aufgerufen haben. Solche VHs haben sich und ihre Unter-VHs bereits selbst daraufhin überprüft, ob sie die gesuchte SP besitzen.

Auch der letzte Schritt ③ führt die Rekursion nur beim „Besitzer“ von Objekt „A“ weiter, wenn dieser nicht bereits selbst eine Rekursion durchführt und dabei die Methode `getPropertyObject` in diesem Objekt „A“ aufgerufen hat. In diesem Fall durchsucht der „Besitzer“ bereits selbst sich und seine Unter-VHs nach der SP und darf kein weiteres Mal rekursiv durchlaufen werden.

Insgesamt werden so redundante Suchvorgänge und nicht-terminierende Ketten von Methodenaufrufen vermieden.

Als weiteres Detail ist anzumerken, dass auch hier alle oben beschriebenen Schleifen abgebrochen werden, sobald die gesuchte SP gefunden ist.

Anhand der konstruierten SO-Hierarchie aus dem vorangehenden Beispiel soll die Wirkungsweise dieses Suchalgorithmus verdeutlicht werden. In der Verhaltensweise „VH5“ werde nun die Methode `getPropertyObject` aufgerufen, um die SP „SP5“ anhand der Bezeichnung des Feldes, das in einer anderen VH als `public`-Feld auf sie verweist, zu finden. Die einzelnen Knoten der Hierarchie sind in der Reihenfolge nummeriert, in der sie durch den Algorithmus daraufhin überprüft werden, ob sie die gesuchte SP besitzen oder selbst diese SP repräsentieren. Diese Reihenfolge ist an den Knoten durch elliptisch umrahmte Nummern angegeben.

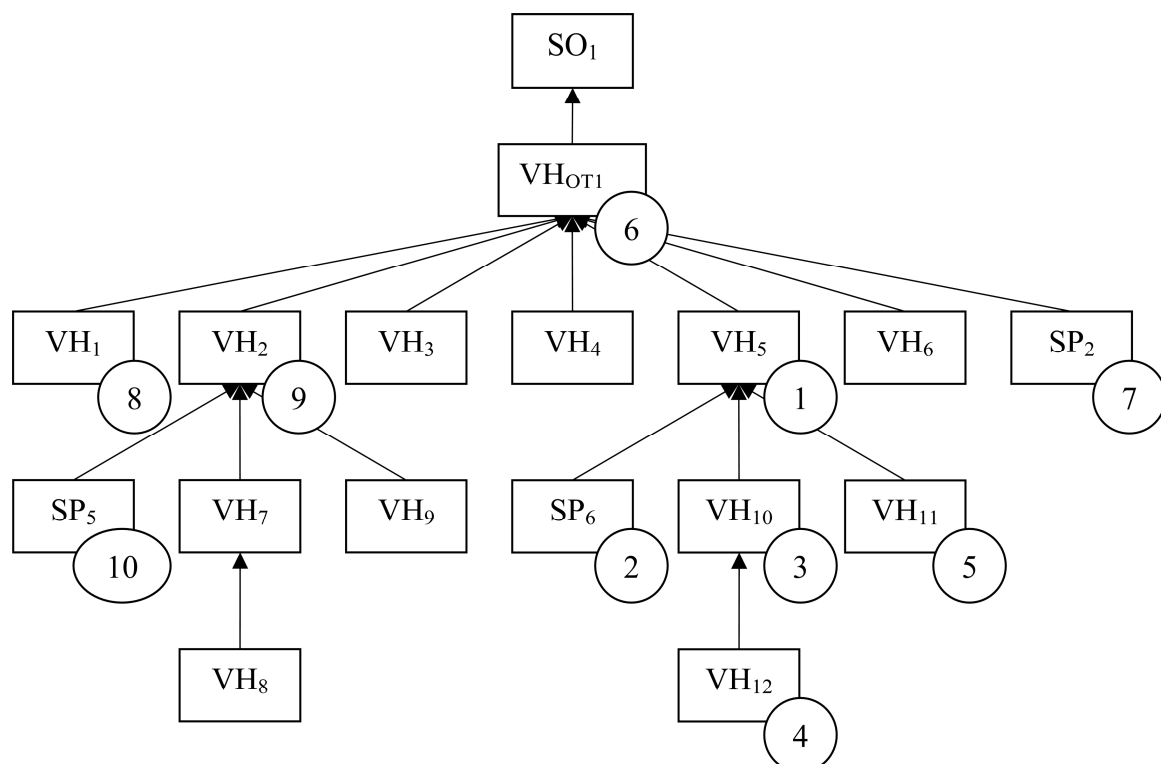


Abbildung 38: Beispiel einer Baumsuche nach einer SP

Die einzelnen Prüfschritte werden nun anhand der ihnen im obigen Diagramm zugeordneten Nummern genauer erläutert, dabei wird auch auf die oben erläuterten grundsätzlichen Schritte ①-③ aus `getPropertyObject` mit den entsprechenden kreisförmig umrahmten Bezeichnungen Bezug genommen:

1:

Da die Methode `getPropertyObject` in VH_5 aufgerufen wird, prüft diese Methode in ihrem Schritt ① zunächst, ob VH_5 die gesuchte SP direkt enthält.

2:

VH_5 enthält nur die SP SP_6 . Da dies nicht die gesuchte SP ist, folgen in Schritt ② nun die rekursiven Aufrufe von `getPropertyObject` in allen direkten Unter-VHs von VH_5 (siehe Schritte 3-5).

3:

Der erste rekursive Aufruf startet in VH_{10} . VH_{10} hat selbst keine SPs. Deshalb startet diese VH in Schritt ② direkt einen rekursiven Aufruf in ihrer Unter-VH VH_{12} .

4:

Auch VH_{12} hat keine SPs, ebenso keine weiteren Unter-VHs. Es wird auch nicht im letzten Schritt ③ die Methode `getPropertyObject` im „Besitzer“ VH_{10} aufgerufen, da der Aufruf von `getPropertyObject` in VH_{12} von VH_{10} ausging.

5:

Der zweite von VH_5 ausgehende rekursive Aufruf startet in VH_{11} . Diese VH hat keine SPs oder Unter-VHs und der entsprechende Aufruf endet sofort.

Analog zum Aufruf von `getPropertyObject` in VH_{12} rufen auch die in VH_{10} und VH_{11} gestarteten `getPropertyObject`-Methoden **nicht** im letzten Schritt ③ die gleiche Methode in VH_5 auf, da die Aufrufe von `getBehaviour` in VH_{10} und VH_{11} von VH_5 ausgingen.

6:

Nachdem alle Unter-VHs von VH_5 rekursiv durchsucht wurden, wird im letzten Schritt ③ rekursiv die Methode `getPropertyObject` im „Besitzer“ VH_{OT1} von VH_5 aufgerufen.

7:

Nun werden in Schritt ① alle direkten SPs von VH_{OT1} getestet, wobei dort jedoch nur die SP SP_2 zu finden ist und diese nicht die gesuchte SP ist.

8-9:

Dann werden in Schritt ② alle Unter-VHs von VH_{OT1} rekursiv durchsucht, wobei das rekursive Durchsuchen von VH_5 entfallen würde, da es selbst `getPropertyObject` in VH_{OT1} aufgerufen hat. Allerdings wird die gesuchte SP schon vor dem möglichen Aufruf von `getPropertyObject` in VH_5 gefunden. Zuerst wird `getPropertyObject` in VH_1 aufgerufen. VH_1 besitzt keine eigenen SPs, die überprüft werden könnten, ebenso keine Unter-VHs, die rekursiv durchsucht werden könnten. Da `getPropertyObject` in VH_1 von VH_{OT1} aufgerufen wurde, ruft VH_1 auch nicht mehr in seinem Schritt ③ `getPropertyObject` in VH_{OT1} auf. Von den übrigen direkten Unter-VHs von VH_{OT1} besitzt nur VH_2 weitere Unter-VHs.

10:

Der Aufruf von `getPropertyObject` in `VH2` durchsucht nun zunächst in Schritt ① die direkt zu `VH2` gehörenden SPs und findet dabei die gesuchte SP `SP5`. Dies gilt unter der Annahme, dass dies die gesuchte SP ist und die korrekte Bezeichnung des Feldes, das in `VH2` auf `SP5` verweist, dem ursprünglichen Aufruf von `getPropertyObject` in `VH5` als String übergeben wurde. Ferner darf die übergebene Bezeichnung des Feldes im Verlauf dieser Suche nicht schon vorher als Bezeichner eines anderen Feldes gefunden werden, das auf eine der übrigen getesteten SPs verweist. Auf diese erforderliche Eindeutigkeit wird weiter unten nach den Erläuterungen zu der Methode `getPropertyObjectOfType` näher eingegangen. Die aktuell in `VH2` ausgeführte Methode `getPropertyObject` bricht daraufhin ab und `SP5` wird als Ergebnis dieser Methode an die aufrufende Methode zurückgegeben. Die aufrufende Methode bricht dann ebenfalls ab und übergibt das übernommene Ergebnis weiter nach oben in der rekursiven Aufrufstruktur von `getPropertyObject`. Schließlich gibt die in `VH5` aufgerufene Methode `getPropertyObject` die gesuchte SP „`SP5`“ als Ergebnis zurück.

MC_Behaviour `getBehaviourOfType(String cn)` :

Diese Methode sucht eine VH analog zu `getBehaviour`, jedoch wird die VH hier anhand ihres Typs gesucht. Dies kann sinnvoll sein, da in den meisten Fällen eine Verhaltensweise eines bestimmten Typs höchstens einmal innerhalb eines SOs zu finden ist und ihr Typ über eine Verhaltensweise mehr aussagt als der Name des Feldes, das sie referenziert. Im Allgemeinen sollte man daher VHs auch über ihren Typ suchen und nicht über den Namen ihres Container-Feldes. Der Klassenbezeichner, der den Typ der gesuchten VH angibt, wird als String übergeben.

MC_Property `getPropertyObjectOfType(String cn)` :

Diese Methode sucht eine SP analog zu `getPropertyObject`, jedoch wird die SP hier anhand ihres Typs gesucht. Dies kann in einigen Fällen sinnvoll sein, wenn eine SP gesucht wird, deren Typ im Allgemeinen höchstens einmal in einem OT verwendet wird. Ein Beispiel dafür ist der SP-Typ `MCP_MaterialDescriptorContainer`. Eine SP dieses Typs speichert Informationen über die physikalischen Materialeigenschaften eines Objektes. Es ist daher nicht sinnvoll, einem Objekttyp mehrere SPs dieses Typs zuzuordnen.

Wenn SPs und VHs sinnvoll anhand ihres Typs gesucht werden können, ist es nicht mehr erforderlich, den Bezeichner des Feldes zu kennen, das die gesuchte SP bzw. die gesuchte VH speichert. Dies kann bei der Implementation eines Zugriffs auf eine VH oder eine SP eine Erleichterung sein, da ein solcher Feldbezeichner unabhängig von Typ der referenzierten SP bzw. der referenzierten VH frei gewählt werden kann und damit eher unbekannt ist als ihr Typ. Insbesondere, wenn man in einem SO eine bestimmte VH sucht, wird im Allgemeinen eher deren Typ bekannt sein als der Bezeichner des Feldes, in dem sie gespeichert ist.

Die oben beschriebenen Baumsuchen wurden implementiert, um den Zugriff auf SPs und VHs insgesamt zu vereinfachen. Um eine SP oder eine VH zu finden, muss so nicht explizit angegeben werden, wo sich diese innerhalb der SO-Hierarchie des zugehörigen SOs direkt befindet und unter Umständen muss sogar nur ihr Typ bekannt sein.

Es gibt verschiedene Beispiele, in denen die Verwendung dieser Baumsuche sinnvoll ist. Ein mechanisch simuliertes Objekt muss z.B. die Verhaltensweise `MCB_Physics` enthalten. Sie enthält unter anderem alle SPs des Objektes, die seinen mechanischen Zustand beschreiben.

Es ist jedoch nicht vorgeschrieben, wo sich diese VH innerhalb der SO-Hierarchie befinden muss und auch der Bezeichner des Feldes, in dem diese VH gespeichert ist, kann frei gewählt werden. Will nun ein Objekt a auf die mechanischen Eigenschaften eines anderen Objektes b zugreifen, so kann es die entsprechende VH in b durch den Methodenaufruf `b.getBehaviourOfType("MCB_Physics")` erhalten und darüber dann auf die entsprechenden SPs zugreifen. Es wurde auch ein OT implementiert, der die grafische Darstellung eines dreidimensionalen Vektors repräsentiert, der Daten aus wählbaren SPs eines anderen Objektes in der Szene liest und diese dann darstellt. So ist es beispielsweise möglich, die aktuelle Geschwindigkeit und Richtung eines mechanisch simulierten Objektes in einer 3D-Animation als Vektor darzustellen. Für eine Instanz „A“ dieses OTs werden in der grafischen Benutzeroberfläche des 3D-Animationswerkzeugs nur die Bezeichnungen der Felder festgelegt, mit denen die darzustellenden SPs referenziert werden sowie das 3D-Objekt „B“, aus dem die Werte für diese SPs gelesen werden sollen. Durch die Baumsuche ist es für den Vektor „A“ nun möglich, die gesuchten SPs innerhalb des Objektes „B“ zu finden, ohne deren exakte Position in der SO-Hierarchie von Objekt „B“ zu kennen.

Da die vier oben angegebenen Methoden zur Suche von VHs bzw. SPs entsprechende Laufzeitfehler erzeugen, wenn die gesuchte VH oder SP nicht gefunden werden konnte, kann mit entsprechenden `try-catch`-Konstrukten auch überprüft werden, ob ein SO oder eine VH überhaupt die gesuchte SP oder VH besitzt. Auf diese Weise testet das Hauptprogramm MaxControl, ob ein SO eine VH des Typs `MCB_Physics` besitzt und infolgedessen mechanisch simuliert werden muss.

8.3.4 Zugriffspfade für Simulationseigenschaften und Verhaltensweisen

Bezeichner von Feldern müssen nur innerhalb einer Klasse eindeutig sein, und verschiedene Klassen von SPs und VHs können sogar innerhalb einer Klasse mehrfach verwendet werden. Die Ergebnisse der vorgestellten Suchmethoden können also von der SO-Hierarchie abhängen, wenn der übergebene Feldbezeichner bzw. der übergebene Typ mehrfach in der durchsuchten SO-Hierarchie vorkommen. Diese Ergebnisse können sich auch ändern, sobald Änderungen an der SO-Hierarchie vorgenommen werden. Es ist daher sicherer, den Zugriff auf eine SP oder eine VH so genau wie möglich zu implementieren.

Es sei in Abbildung 39 zur Verdeutlichung noch einmal das vorangehende Beispiel einer SO-Hierarchie herangezogen. In der Verhaltensweise „VH₅“ werde wieder SP „SP₅“ gesucht. In den Ellipsoiden sei jeweils der Bezeichner des Feldes, das die zugeordnete VH bzw. die zugeordnete SP referenziert, angegeben.

Die Verhaltensweise „VH₅“ kann nun auf verschiedene Arten auf die SP „SP₅“ zugreifen, hier seien drei Beispiele verglichen:

```
getPropertyObject("FSP5") :
```

Ob dieser Zugriff das gewünschte Ergebnis liefert, ist davon abhängig, dass der Feldbezeichner „FSP5“ entweder nur einmal in der SO-Hierarchie von SO₁ verwendet wird oder die Baumsuche das gewünschte Feld zuerst findet, auch wenn es noch weitere Felder mit derselben Bezeichnung gibt. Welches Feld die Baumsuche zuerst findet, hängt von der Struktur der Hierarchie ab, die deshalb entsprechend gewählt werden sollte.

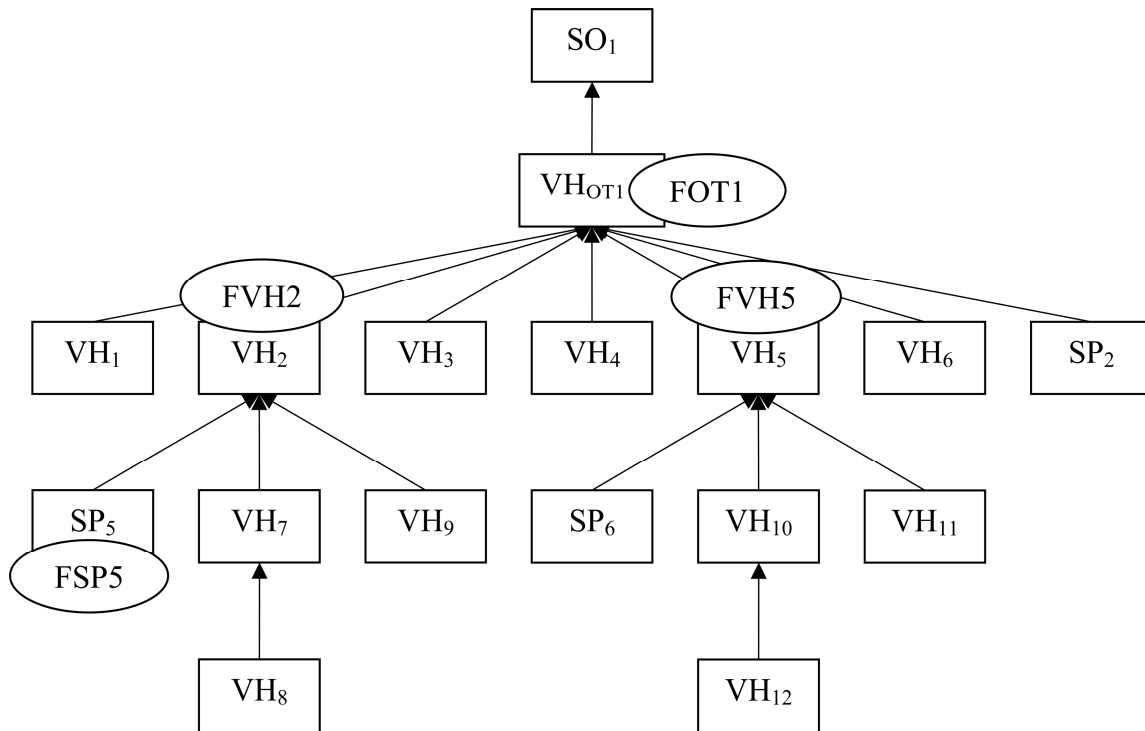


Abbildung 39: Beispiel einer SO-Hierarchie zur Illustration von Zugriffspfaden

Zur näheren Erläuterung dieser Empfehlung sei hier in Abbildung 40 noch einmal das erste Beispiel einer OT-Hierarchie aus Abbildung 30 in Kapitel 7.2.1 eingeschoben. OT_{Robot} sei nun instanziiert, so dass die entsprechende SO-Hierarchie SO_{Robot} entsteht, wobei hier, wie bei der vorangehenden SO-Hierarchie auch, zusätzlich jeweils in Ellipsoiden der Bezeichner des Feldes, das die zugeordnete VH oder SP referenziert, angegeben ist:

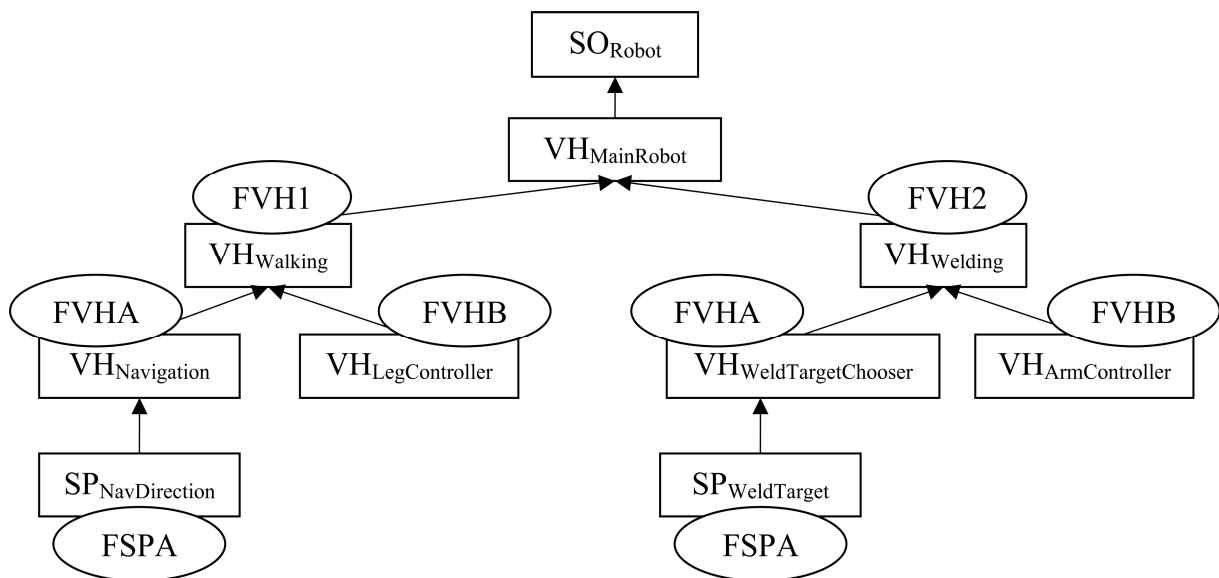


Abbildung 40: Beispiel zur Nutzung semantischer Zusammenhänge für Zugriffspfade

Wie man sieht, tragen die Felder, welche die VHs „ $VH_{Navigation}$ “ und „ $VH_{WeldTargetChooser}$ “ referenzieren, jeweils den gleichen Bezeichner, dies gilt ebenso für „ $VH_{LegController}$ “ und „ $VH_{ArmController}$ “. Gleiches gilt für die SPs „ $SP_{NavDirection}$ “ und „ $SP_{WeldTarget}$ “ in „ $VH_{Navigation}$ “ bzw. in „ $VH_{WeldTargetChooser}$ “.

Wie schon in den Kapiteln 7.2.1 und 7.2.4 gesagt wurde, sollte die hierarchische Struktur der VHs und SPs eines OTs semantische Zusammenhänge zwischen den VHs und SPs widerspiegeln.

Die Einhaltung dieser Designvorschrift unterstützt das gewünschte Verhalten eines Zugriffs wie `getBehaviour("FVHA")`. Nach der Definition der Methode `getBehaviour` wird ein Aufruf von `getBehaviour("FVHA")` in „VH_{Walking}“ und in „VH_{LegController}“ die VH „VH_{Navigation}“ zurückliefern. Dagegen wird ein Aufruf von `getBehaviour("FVHA")` in „VH_{Welding}“ und in „VH_{ArmController}“ die VH „VH_{WeldTargetChooser}“ zurückliefern. Es wird also in beiden Fällen die in der Hierarchie nächstliegende und damit auch jeweils die semantisch nächstliegende VH gefunden, wenn dieses Designkriterium eingehalten wurde.

Analog dazu wird ein Aufruf von `getPropertyObject("FSPA")` in „VH_{Walking}“ oder in einer ihrer Unter-VHs die SP „SP_{NavDirection}“ zurückliefern. Dagegen wird ein Aufruf von `getPropertyObject("FSPA")` in „VH_{Welding}“ oder in einer ihrer Unter-VHs die SP „SP_{WeldTarget}“ zurückliefern. Es wird also auch hier in beiden Fällen die in der Hierarchie nächstliegende und damit auch jeweils die semantisch nächstliegende SP gefunden, wenn dieses Designkriterium eingehalten wurde.

Es wird nun wieder zur ursprünglichen Beispiel-SO-Hierarchie SO₁ zurückgekehrt. Die folgenden Zugriffe sind weitere Möglichkeiten für die Verhaltensweise „VH₅“, auf die SP „SP₅“ zuzugreifen:

```
rootOwner.getBehaviour("FOT1").getBehaviour("FVH2").getPropertyObject("FSP5");
```

Dieser Zugriff ist sicherer, erfordert jedoch auch genauere Kenntnis über die SO-Hierarchie SO₁. Durch „rootOwner“ erhält die Verhaltensweise VH₅ zunächst die Wurzel der SO-Hierarchie zu SO₁, also SO₁ selbst. Von dort aus wird mit `getBehaviour("FOT1")` direkt auf die Verhaltensweise „VH_{OT1}“ zugegriffen. Der folgende Aufruf `getBehaviour("FVH2")` liefert die direkt in VH_{OT1} zu findende Verhaltensweise „VH₂“ zurück. Durch `getPropertyObject("FSP5")` wird dann deren direkt zugehörige SP „SP₅“ gefunden. Der erforderliche Zugriffspfad wurde hier also vollständig angegeben, so dass das Ergebnis nicht vom rekursiven Vorgehen der Baumsuche abhängt, da bei jedem der entsprechenden Methodenaufrufe nur direkte Kindknoten in Schritt ① durchsucht werden müssen, um die gesuchte SP zu finden. Da diese Kindknoten jeweils durch Felder der jeweiligen „Besitzer“ referenziert werden, kann bei der Suche nach einem Kindknoten keiner der zugehörigen Feldbezeichner doppelt vorkommen, so dass die Suchergebnisse auch unabhängig von der Reihenfolge sind, in der diese Kindknoten in einem Aufruf von `getPropertyObject` in Schritt ① überprüft werden. Da die Angabe eines vollständigen Zugriffspfades unnötige Baumsuchen vermeidet, ist diese Art des Zugriffs auch effizienter als eine Suche, die den Zugriffspfad weniger genau angibt und damit die oben beschriebene automatische Baumsuche erforderlich macht.

Ein solcher Zugriffspfad kann auch relativ zum Ausgangspunkt angegeben werden, in diesem Beispiel durch:

```
owner.getBehaviour("FVH2").getPropertyObject("FSP5");
```

Durch „owner“ erhält die Verhaltensweise „VH₅“ zunächst ihren direkten Besitzer „VH_{OT1}“. Der folgende Aufruf `getBehaviour("FVH2")` liefert die direkt in „VH_{OT1}“ zu findende Verhaltensweise „VH₂“ zurück. Durch `getPropertyObject("FSP5")` wird dann deren direkt zugehörige SP „SP₅“ gefunden. Der erforderliche Zugriffspfad wurde also auch hier vollständig angegeben. Der Unterschied zum vorangehenden Beispiel besteht in der relativen Festlegung des Zugriffspfades, wodurch dieser Zugriff unabhängiger von Änderungen an der

SO-Hierarchie ist. Wird der Typ der Hauptverhaltensweise „VH_{OT1}“ beispielsweise in einem anderen OT als Unter-VH wieder verwendet, so könnte die in Abbildung 41 folgende neue SO-Hierarchie entstehen.

Der Zugriff

`rootOwner.getBehaviour("FOT1").getBehaviour("FVH2").getPropertyObject("FSP5")`
würde hier nicht mehr wie gewünscht funktionieren, da in dem Feld `rootOwner` nun SO₂ referenziert wird. Der Aufruf `getBehaviour("FOT1")` würde hier nun von SO₂ ausgehend VH_{1B} zurückliefern, wenn „FOT1“ auch der **frei wählbare** Name des Feldes ist, das in VH_{OT2} die VH „VH_{1B}“ referenziert. Der folgende Aufruf `getBehaviour("FVH2")` würde hier aufgrund der Vorgehensweise der Baumsuche von VH_{1B} ausgehend VH_{2B} zurückliefern, wenn „FVH2“ auch der Name des Feldes ist, das in VH_{OT2} die VH „VH_{2B}“ referenziert und ein Feld mit dieser Bezeichnung nicht in SO₂ zur Referenzierung von VH_{OT2} verwendet wird. Der darauf folgende Aufruf `getPropertyObject("FSP5")` würde hier von VH_{2B} ausgehend SP_{1B} zurückliefern, wenn „FSP5“ auch der Name des Feldes ist, das in VH_{OT2} die SP „SP_{1B}“ referenziert. Somit würde nicht die gewünschte SP zurückgeliefert werden.

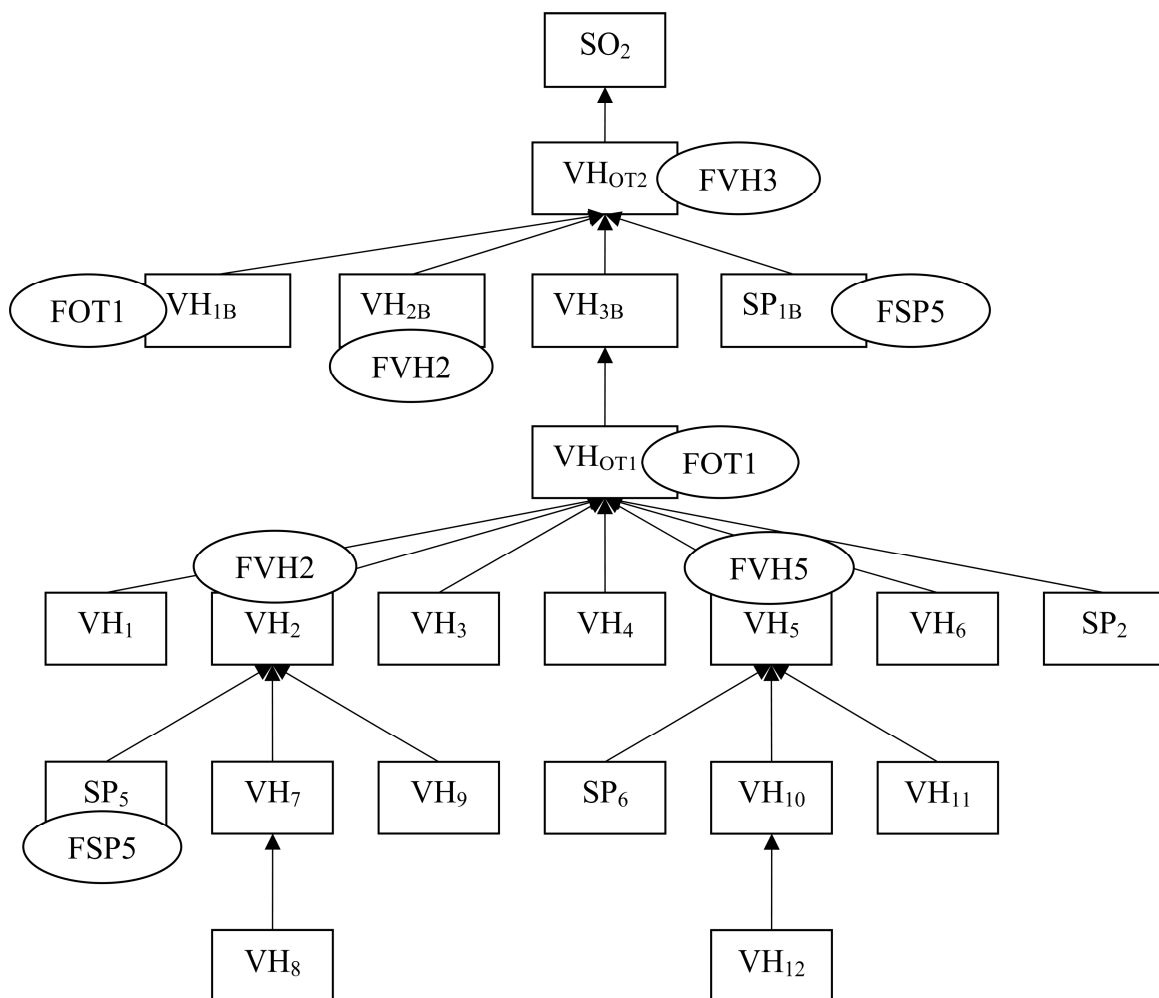


Abbildung 41: Unempfindlichkeit relativer Zugriffspfade gegen Änderungen an der SO-Hierarchie

Dagegen würde der Zugriff

`owner.getBehaviour("FVH2").getPropertyObject("FSP5")` immer noch das gleiche Ergebnis wie für die vorangehende SO-Hierarchie liefern, da im Feld `owner` der VH „VH₅“, in der dieser Aufruf gestartet wird, auch hier die VH „VH_{OT1}“ referenziert ist.

Somit würden die folgenden Aufrufe `getBehaviour("FVH2")` und `getPropertyObject("FSP5")` die gleichen Ergebnisse liefern wie für die vorangehende SO-Hierarchie.

Bei den hier beschriebenen Methoden für die Suche nach SPs und VHs werden die in den Feldern `properties` bzw. `behaviours` gespeicherten Listen verwendet. Zu diesen Listen wurde oben in Kapitel 8.3.1 bereits erläutert, dass sie nur die SPs bzw. VHs enthalten, die nicht durch andere Felder in der betrachteten Instanz von `MC_Owner` überdeckt werden. Dies ist ein weiterer Vorteil beim Zugriff auf VHs und SPs über die hier vorgestellten Methoden im Vergleich zu einem direkten Zugriff auf die Felder, die auf diese VHs und SPs verweisen.

Ein Nachteil bei der Verwendung dieser Methoden ist allerdings, dass Fehler in den übergebenen Feldnamen oder Typbezeichnungen erst zur Laufzeit erkannt werden, da diese als Strings übergeben werden, die vom Java-Compiler nicht mit den Feld- und Typnamen verglichen und so auf Korrektheit überprüft werden. Insgesamt überwiegen jedoch die Vorteile dieser Methoden. Zusätzlich bleibt es dem Benutzer freigestellt, auch direkte Zugriffe auf Felder zu implementieren, die auf SPs oder VHs verweisen. Um dabei die oben genannten Probleme mit überdeckten Feldern zu vermeiden, kann der Benutzer auch für jedes Feld „get“ und „set“ Zugriffsmethoden implementieren, wie es allgemein in Java üblich ist. Die Konzepte von `MaxControl` wurden stets so entwickelt, dass sie den Benutzer unterstützen und ihn möglichst nicht in seinen Möglichkeiten einschränken, deshalb sind auch hier solche Abweichungen vom eigentlichen Konzept möglich.

8.3.5 Bequemer Zugriff auf die Werte von Simulationseigenschaften

Um mit Hilfe einer Methode wie `getPropertyObject` den aktuellen Wert einer SP zu lesen, muss in dem gefundenen Objekt der Klasse `MC_Property` die Methode `getValue` aufgerufen werden. Diese liefert nach ihrer Signatur auch in Unterklassen von `MC_Property` ein Objekt der Klasse `Object` zurück. Ohne die Übergabeparameter von `getValue` zu verändern, kann man in Java eine geerbte Methode nicht so überschreiben, dass ihre Signatur einen anderen Rückgabetyt erhält. Bei einer `Double-SP` muss diese Methode also auch so implementiert werden, dass sie nach ihrer Signatur immer noch ein Objekt der Klasse `Object` zurückliefert. Durch die Polymorphie ist es aber dennoch möglich, die Methode `getValue` so zu implementieren, dass sie tatsächlich eine Instanz einer Unterklasse von `Object` zurückliefert, z.B. ein Objekt der Klasse `Double`. Um in einem so zurückgelieferten Objekt auf die Methoden der Klasse `Double` zugreifen zu können, die in der Oberklasse `Object` fehlen, müsste dieses Objekt zunächst durch eine `Cast-Anweisung` in den Typ `Double` umdeklariert werden. Durch die Methode `doubleValue` dieser Klasse kann man dann den aktuellen Wert der `Double-SP` als primitiven `double-Wert` erhalten.

Ein vergleichbares Vorgehen ist nötig, wenn man den aktuellen Wert einer SP verändern will. Dazu muss in dem gefundenen Objekt der Klasse `MC_Property` die Methode `setValue` aufgerufen werden. Es ist für `MaxControl` eine festgelegte Konvention, dass auch in Unterklassen von `MC_Property` die Signatur von `setValue` nicht verändert wird. Dieser Methode muss also eine Instanz der Klasse `Object` übergeben werden. Bei einer `Double-SP` kann man den gewünschten neuen `double-Wert` zunächst in einem `Double-Objekt` speichern, das dann wegen der Polymorphie an `setValue` übergeben werden kann.

Um solche Cast-Anweisungen und Umwandlungen von primitiven Werten in Objekte zu vermeiden und die Suche nach einer SP zusammen mit dem Lesen oder Schreiben ihres Wertes bequem in einem Befehl bündeln zu können, enthält die Klasse `MC_Owner` verschiedene Hilfsmethoden, welche das erforderliche Vorgehen automatisieren und den Zugriff auf SPs somit erleichtern. Hier werden nur einige wichtige dieser Methoden vorgestellt:

`double getDoubleValue(String name) :`

Diese Methode sucht eine Double-SP, die in einem Feld referenziert wird, dessen Bezeichnung dem übergebenen `String` „name“ entspricht. Der Wert dieser SP wird dann ausgelesen und als primitiver `double`-Wert zurückgegeben. Wie auch in allen anderen Methoden aus dieser Gruppe wird zum Suchen der SP die in Kapitel 8.3.3 erläuterte Methode `getPropertyObject` verwendet. Die Suche beginnt in der Instanz der Klasse `MC_Owner`, in der die Methode `getDoubleValue` aufgerufen wurde.

`boolean getBooleanValue(String name) :`

Diese Methode liest den aktuellen Wert einer booleschen SP und gibt ihn als primitiven `boolean`-Wert zurück.

`long getLongValue(String name) :`

Diese Methode liest den aktuellen Wert einer Integer-SP und gibt ihn als primitiven `long`-Wert zurück.

`String getStringValue(String name) :`

Diese Methode liest den aktuellen Wert einer String-SP und gibt ihn als Objekt des Typs `String` zurück. Eine String-SP speichert eine Zeichenkette und ist nicht animierbar.

`MC_Object getMC_ObjectValue(String name) :`

Diese Methode liest den aktuellen Wert einer SP, die eine Referenz auf ein Simulationsobjekt in der Szene speichert, und gibt ihn als Objekt der Klasse `MC_Object` zurück. Wie in Kapitel 8.4 erläutert wird, werden die 3D-Objekte in der Szene durch Instanzen der Klasse `MC_Object` repräsentiert, die hier auch als Simulationsobjekte bezeichnet werden. Eine SP dieses Typs ist nicht animierbar.

`boolean getManualValue(String name) :`

Diese Methode liest den aktuellen Wert der MP, die zu einer SP gehört. Die MP bestimmt, ob die zugehörige SP im aktuellen Frame `n` der Animation als manuell kontrolliert gilt und dann von der Simulation nicht verändert werden darf (siehe Kapitel 6.6). Ist der „aktuelle Simulationsschritt“ der Simulationsschritt zu einem Frame `n`, so ist dieser Frame `n` auch der „aktuelle Frame“. Wie in den vorangehen beschriebenen Methoden wird die zugehörige SP zunächst anhand des Übergabeparameters „name“ gesucht.

`double getXValue(String name) :`

Diese Methode kann zum Lesen von Werten aus SPs verwendet werden, deren Werte aus jeweils drei Komponenten bestehen und die Form `(x, y, z)` haben. Dies trifft auf SPs des Typs

MCP_XYZ zu. Direkte Unterklassen dieses Typs sind MCP_Position, MCP_Rotation und MCP_Scale. SPs dieser Typen können die Position, die Rotation bzw. die Skalierung eines SOs speichern. Auf diese Arten von SPs wird in Kapitel 8.5.1 näher eingegangen.

Die Methode `getXValue` kann nun verwendet werden, um die X-Komponente des Wertes einer solchen SP zu lesen. Aus einer SP des Typs MCP_Position kann so z.B. die aktuelle X-Koordinate des zugehörigen SOs als Fließkommawert doppelter Genauigkeit gelesen werden.

Analog zu dieser Methode existieren auch die Methoden `getYValue` und `getZValue`, mit denen die Y- bzw. die Z-Komponente eines solchen Wertes gelesen werden kann.

```
double setDoubleValue(String name, double v) :
```

Diese Methode setzt den Wert einer Double-SP auf den Wert des übergebenen `double`-Parameters „v“. Der Rückgabewert ist der Wert, den diese SP nach der Zuweisung trägt. Dieser Wert kann von „v“ abweichen, wenn die SP im aktuellen Frame als manuell kontrolliert gilt, weil ihre MP den Wert „true“ hat. Dann wird der übergebene Wert nicht angenommen und die SP bleibt unverändert.

Alle weiteren Methoden dieser Art zum Setzen der Werte von SPs arbeiten analog, weshalb diese hier nicht näher behandelt werden. Zu jeder der hier beschriebenen Methoden für das vereinfachte Lesen von Werten aus SPs gibt es auch eine entsprechende Methode für das Setzen dieser Werte, die jeweils auch eine analoge Bezeichnung hat, bei der das Präfix „get“ durch das Präfix „set“ ersetzt ist.

8.3.6 Suche nach Kindobjekten und deren Verhaltensweisen

```
Vector<MC_Object> getChildrenOfType(String cn) :
```

Diese Methode durchsucht die direkten und indirekten Kindobjekte des zugehörigen 3D-Objektes „A“. In einer Szene können jedem 3D-Objekt andere 3D-Objekte als Kindobjekte zugeordnet sein (siehe Kap. 3.3). Die Methode `getChildrenOfType` kann direkt in dem zu „A“ gehörigen Java-Objekt des Typs MC_Object aufgerufen worden sein oder in einer seiner VHs. Der übergebene `String` gibt den Bezeichner der Klasse an, der die gesuchten Kindobjekte angehören sollen. So ist es möglich, zu einem 3D-Objekte direkte oder indirekte Kindobjekte eines bestimmten Typs zu suchen. Ein Fahrzeug, das neben seinen Rädern auch Lichtquellen wie Blinker als Kindobjekte hat, kann so gezielt nach seinen Rädern suchen, wenn diese Instanzen einer bekannten Klasse sind. Die gefundenen Kindobjekte werden als Liste der Klasse `Vector` zurückgegeben.

```
Vector<MC_Behaviour> getChildrensBehavioursOfType(String cn) :
```

Diese Methode ist der vorangehend beschriebenen Methode `getChildrenOfType` sehr ähnlich. Die direkten und indirekten Kindobjekte des zugehörigen 3D-Objektes werden hier jedoch nach VHs des als Parameter übergebenen Typs durchsucht. Dabei wird jeweils die gesamte SO-Hierarchie überprüft, es werden also nicht nur direkte sondern auch indirekte Unter-VHs gesucht. Statt der gefundenen Kindobjekte werden dann ggf. deren VHs des gesuchten Typs als Liste der Klasse `Vector` zurückgegeben. Dies kann sinnvoll sein, wenn ein Objekt direkt auf bestimmte Verhaltensweisen seiner Kindobjekte zugreifen möchte. Die Verhaltensweise `MCB_CarControlBasic`, die für `MaxControl` entwickelt wurde, um die

grundsätzliche Steuerung eines Fahrzeugs der Klasse `MCD_Car` zu übernehmen, führt als ihren ersten Schritt in der Simulation folgende Anweisung aus:

```
tyres=getChildrensBehavioursOfType("MCB_CarTyreControl");
```

So wird nicht wie im vorangehenden zu `getChildrenOfType` genannten Beispiel eine Liste der Räder des Fahrzeugs erhalten, sondern es wird eine Liste der VHs des Typs `MCB_CarTyreControl` erzeugt, die in den SOs, welche diese Räder repräsentieren, zu finden sind. Die VH `MCB_CarControlBasic` muss nämlich in den meisten Anweisungen nicht auf die direkten Eigenschaften der Räder wie deren Position zugreifen, sondern hauptsächlich auf deren VHs des Typs `MCB_CarTyreControl`, über die das Verhalten der Räder gesteuert werden kann, indem man z.B. die Werte ihrer SPs für Drehkraft und Steuerungswinkel verändert. Als Beispiel dafür wäre die folgende Anweisung möglich, die das Drehmoment des ersten Rades auf 20Nm einstellt:

```
tyres.elementAt(0).setDoubleValue("torque", 20d);
```

Die SP „torque“ gehört nicht direkt zu SOs der Klasse `MCD_CarTyre` für Räder, sondern ist eine SP ihrer Unter-VH „`MCB_CarTyreControl`“. Die oben genannte Baumsuche für SPs würde die SP „torque“ zwar auch von einer Instanz der Klasse `MCD_CarTyre` ausgehend finden. Jedoch kann sich das Ergebnis einer direkt bei einer Instanz der Klasse `MCB_CarTyreControl` beginnenden Suche nach dieser SP auch dann nicht unerwünscht ändern, wenn die Bezeichnung „torque“ noch für eine weitere SP verwendet wird, die später der OT-Hierarchie zu `MCD_CarTyre` hinzugefügt wird. Denn eine weitere SP, deren sie referenzierendes Feld die Bezeichnung „torque“ trägt, kann nur einer anderen VH als `MCB_CarTyreControl` hinzugefügt werden, da sonst zwei Felder mit derselben Bezeichnung in einer Klasse definiert würden, was in Java nicht erlaubt ist. Es würde also zuerst immer die gewünschte SP in der Instanz der Klasse `MCB_CarTyreControl` gefunden werden, da die Suche direkt dort beginnt. Bei einer Suche, die bei einer Instanz von `MCD_CarTyre` beginnt, könnte sich das Suchergebnis dagegen in einem solchen Fall ändern. Wenn die hinzugefügte SP mit der Bezeichnung „torque“ z.B. direkt dem OT `MCD_CarTyre` hinzugefügt wird, würde diese neue SP gefunden werden. Weiterhin ist die Suche effizienter, wenn sie in der VH beginnt, welche die gesuchte SP direkt enthält, weil dann eine komplexe Baumsuche vermieden wird.

8.3.7 Beeinflussung der Mechaniksimulation und Lesen von Kollisionsereignissen

Simulationsobjekte, die nach den Gesetzen der Mechanik simuliert werden, können den Verlauf der Mechaniksimulation beeinflussen, indem ihre VHs für sich oder andere mechanisch simulierte SOs Kräfte erzeugen oder Federn zwischen Objekten erzeugen. Ebenso können Kollisionsereignisse ausgelesen werden.

Einige hierzu verwendbare Methoden werden im Folgenden erläutert.

```
void addCenterForce(double x, double y, double z):
```

Diese Methode fügt eine Kraft zu den Kräften hinzu, die auf das 3D-Objekt wirken, in dem diese Methode aufgerufen wurde. Die Kraft F wird als dreidimensionaler Vektor übergeben, dessen x-, y- und z-Komponenten in der Einheit Newton angegeben werden. Der Vektor wird in so genannten Weltkoordinaten interpretiert, er ist also relativ zum

Ursprungskoordinatensystem der Szene ausgerichtet. Die Kraft wirkt im Massenmittelpunkt des 3D-Objektes. Die so hinzugefügte Kraft wirkt für das Zeitintervall eines Simulationszwischen schrittes auf das Objekt und muss, wenn sie danach weiterhin wirken soll, erst im folgenden Simulationszwischen schritt erneut hinzugefügt werden. Dies ist insbesondere für die technische Umsetzung relevant, in der die Simulation von Mechanik in **einem** Simulations**zwischen**schrift eventuell **mehrere** interne Simulationsschritte durchführt, für die eine so gesetzte Kraft kontinuierlich wirkt, bis der Simulations**zwischen**schrift beendet ist (siehe dazu Kapitel 10.2.3.1).

```
void addForce(double px, double py, double pz, double x, double y, double z):
```

Diese Methode funktioniert analog zu `addCenterForce`, jedoch wird über die Parameter `px`, `py` und `pz` ein 3D-Punkt P angegeben, der bestimmt, an welchem Punkt des 3D-Objektes die neu hinzugefügte Kraft F angreift. Der Punkt P wird dabei im Koordinatensystem des 3D-Objektes angegeben, so dass er im Ursprungs koordinatensystem zusammen mit dem Objekt transformiert wird. So kann z.B. bei einer Rakete angegeben werden, dass ihr Schub immer am Triebwerk angreifen soll, das im Koordinatensystem der Rakete immer dieselbe Position behält.

```
void addLocalCenterForce(double x, double y, double z):
```

Diese Methode funktioniert analog zu `addCenterForce`, jedoch wird der Vektor, der die neue Kraft F angibt, dabei im Koordinatensystem des 3D-Objektes interpretiert, so dass er im Ursprungs koordinatensystem zusammen mit dem Objekt transformiert wird. So kann z.B. bei einer Rakete angegeben werden, dass ihr Schub immer in der Richtung des Triebwerks wirken soll, das im Koordinatensystem der Rakete immer dieselbe Ausrichtung behält.

```
void addLocalForce(double px, double py, double pz, double x, double y, double z):
```

Diese Methode funktioniert analog zu `addLocalCenterForce` und `addForce`. Sowohl der Punkt P als auch der Vektor F werden dabei im Koordinatensystem des 3D-Objektes angegeben.

```
void addTorque(double x, double y, double z, double w):
```

Diese Methode gibt ein Drehmoment an, das analog zu den `addForce`-Methoden zu den Kräften und Drehmomenten hinzugefügt wird, die im aktuellen Simulationszwischen schritt bisher auf das 3D-Objekt wirken. Die Drehachse für das Drehmoment wird über die Parameter `x`, `y` und `z` als dreidimensionaler Vektor angegeben, der im lokalen Koordinatensystem des 3D-Objektes interpretiert wird. So kann beispielsweise bei einem Autorad angegeben werden, dass ein Drehmoment das Rad immer um seine lokale Radachse drehen soll, unabhängig von der aktuellen Ausrichtung des Rades im Weltkoordinatensystem. Die Stärke des Drehmomentes wird in der Einheit Newtonmeter über den Parameter `w` angegeben. An welchem Punkt ein reines Drehmoment an einem Körper angreift ist physikalisch nicht relevant, daher wird auch kein solcher Angriffspunkt angegeben.

```
void addWorldTorque(double x, double y, double z, double w):
```

Diese Methode funktioniert analog zu `addTorque`, jedoch wird die Drehachse hier in Weltkoordinaten angegeben.

MCH_Spring createSpring(MC_Object o1, MC_Object o2):

Diese Methode erzeugt eine Feder zwischen Objekt o1 und Objekt o2. Die erzeugte Feder wird als Objekt des Typs MCH_Spring zurückgegeben, das Methoden enthält, mit denen die näheren Eigenschaften der Feder wie Stärke und Dämpfung angegeben werden können (siehe Kap. 10.1.4). Ein SO des Typs MCD_CarTyre, das ein Autorrad repräsentiert, verwendet die Methode createSpring, um vor Beginn der Simulation automatisch Federn zu erzeugen, die das Rad mit der Karosserie des Fahrzeugs verbinden. Diese Federn bilden eine Radachse nach, die möglichst nur nach oben und nach unten federn kann und kaum in andere Richtungen federt. Das ist nur näherungsweise möglich, diese Problematik wird in Kapitel 10.2.3.3 genauer behandelt.

Vector<MCH_CollisionInfo> getCollisions():

Diese Methode liefert alle Kollisionen zurück, die bisher im aktuellen Simulationszwischen schritt durch die Simulation von Mechanik für das 3D-Objekt erkannt wurden. Dabei werden nur Kollisionen zwischen 3D-Objekten erkannt, die als Festkörper nach den Gesetzen der Mechanik simuliert werden. Die bisher erkannten Kollisionen werden als Liste des Typs Vector zurückgegeben. Jedes Element der Liste ist eine Instanz der Klasse MCH_CollisionInfo und enthält nähere Informationen über die Kollision zwischen genau zwei 3D-Objekten. Die von dieser Methode zurückgelieferte Liste enthält nur Informationen über Kollisionen, an denen das 3D-Objekt direkt beteiligt ist, in dem die Methode aufgerufen wird. Die Klasse MCH_CollisionInfo wird in Kapitel 10.1.5 näher erläutert.

Über solche Kollisionsdaten können Simulationsobjekte mit ihren VHs auf Kollisionen reagieren, indem z.B. passende Töne erzeugt werden. Aus den Daten ist beispielsweise ersichtlich, mit welcher Geschwindigkeit zwei Objekte aufeinander stoßen, aber auch, mit welcher Geschwindigkeit sie ggf. aneinander reiben oder übereinander rollen. So können Reifen, die über Asphalt rutschen, entsprechende Quietschgeräusche erzeugen. Ebenso können für die Reifen Rollgeräusche erzeugt werden oder ein Aufprallgeräusch, wenn eine Autokarosserie auf einen anderen Gegenstand prallt. Das genaue Vorgehen beim Erzeugen von Tönen durch die Steuerung spezieller Simulationsobjekte, die virtuelle Tonquellen im Raum repräsentieren, wird in Kapitel 10.2.4 näher erläutert.

Vector<MCH_CollisionInfo> getGlobalCollisions():

Diese Methode ist analog zu getCollisions, jedoch liefert sie alle im aktuellen Simulationszwischen schritt durch die Simulation von Mechanik bisher erkannten Kollisionen zurück, auch wenn das Simulationsobjekt, in dem die Methode aufgerufen wird, nicht an der Kollision beteiligt ist. Diese Methode kann z.B. verwendet werden, wenn das Simulationsobjekt auf die Kollisionen aller 3D-Objekte reagieren soll, auch wenn es selbst nicht daran beteiligt ist. Denkbar wäre dies für ein Objekt des Typs MCO_Universe, das global auf alle Objekte wirkt und z.B. anhand Ihrer Kollisionen untereinander einen entsprechenden Rollwiderstand für Objekte erzeugt, die über andere Objekte hinwegrollen.

8.4 MC_Object

Instanzen der Klasse MC_Object sind genau die Simulationsobjekte. Diese Klasse und ihre Unterklassen repräsentieren damit Objekttypen (OTs). Sie enthält entsprechende Felder und Methoden zur Steuerung der 3D-Objekte und für den Zugriff auf wichtige ihrer Daten. Ein Simulationsobjekt kann SPs und VHs besitzen, daher ist die Klasse MC_Object eine direkte

Unterklasse von `MC_Owner`. Sie verfügt deshalb zunächst über dieselben Felder und Methoden wie `MC_Owner`. Einige weitere wichtige Felder und Methoden von `MC_Object` werden hier erläutert.

8.4.1 Informationen über die Szenenhierarchie

`Vector<MC_Object> children:`

Die in diesem Feld referenzierte Instanz der Klasse `Vector` enthält eine Liste aller Kindobjekte des betrachteten 3D-Objektes. Wie in Kapitel 3.3 beschrieben wurde, können 3D-Objekte einer Szene in eine Hierarchie von Ober- und Unterobjekten gebracht werden, um so zusammenhängende Objekte wie ein Auto mit 4 Rädern aus Einzelobjekten zusammzusetzen. In diesem Beispiel kann eine Autokarosserie mit Hilfe des Feldes `children` feststellen, welche Räder in der Szene als Kindobjekte zu ihr gehören, so dass sie diese Räder dann zur Kontrolle des Fahrzeugs ansteuern kann.

`MC_Object parent:`

Analog zum Feld `children` referenziert dieses Feld ggf. das Elternobjekt des betreffenden 3D-Objektes. Simulationsobjekte der Klasse `MCD_ContactSFX` repräsentieren z.B. tonerzeugende 3D-Objekte, die automatisch Toneffekte erzeugen können, wenn ein 3D-Objekt eines bestimmten Materials mit einem anderen 3D-Objekt eines gegebenen Materials auf eine bestimmte Weise kollidiert, z.B. darüber rollt, daran reibt oder dagegen stößt. Dies kann verwendet werden, um z.B. ein Quietschen zu erzeugen, wenn Autoreifen mit einer bestimmten Geschwindigkeit über Asphalt rutschen. Solche tonerzeugenden 3D-Objekte werden als Kindobjekte den 3D-Objekten zugeordnet, auf deren Kollisionen sie reagieren sollen. Um nun festzustellen, auf welche Kollisionsdaten sie beim Erzeugen von Tönen zugreifen sollen, müssen die tonerzeugenden Simulationsobjekte wissen, welche Objekte jeweils ihre Elternobjekte sind. Dies können sie über das Feld `parent` feststellen.

`String sceneName:`

Dieser `String` speichert den Namen, den das 3D-Objekt in der 3D-Szene hat. Der Name muss nicht eindeutig sein, es können durchaus mehrere Objekte in der 3D-Szene denselben Namen haben.

Namen können bei Bedarf auch mit Semantik versehen werden. Ein Fahrzeug kann mehrere Lichtquellen als Kindobjekte haben, die sich in ihrem Typ kaum unterscheiden, z.B. Blinklichter und Lichter, die von außen ein- und ausgeschaltet werden können. Einige dieser Lichter sollen nun Bremsleuchten darstellen, andere sind z.B. Scheinwerfer oder Blinker. Um bequem festlegen zu können, welche Lichter welche Aufgaben übernehmen sollen, könnten z.B. bestimmte Namenspräfixe verwendet werden. Anhand dieser frei wählbaren Präfixe kann die VH `MCB_CarLightController` bestimmen, welche Lichter sie anhand des aktuellen Zustandes eines zugehörigen Fahrzeugs auf welche Weise ansteuern soll. Um vorher festzustellen, welche Lichter überhaupt dem Fahrzeug als Kindobjekte untergeordnet sind, verwendet sie die Java-Anweisung

```
„lights=getChildrenOfType("MCD_Light");“
```

wobei `MCD_Light` die Basisklasse jeder Lichtquelle ist. Die dabei verwendete Methode `getChildrenOfType` aus `MC_Owner` wurde oben in Kapitel 8.3.6 erläutert.

8.4.2 Aufruf der Verhaltensweisen eines Simulationsobjektes

void startup(long f) :

Diese Methode wird vor dem Beginn der eigentlichen Simulation in jedem Simulationsobjekt aufgerufen und ruft ihrerseits die in Kapitel 8.6.3 beschriebenen `startupAll`-Methoden und damit indirekt die `startup`-Methoden aller zugehörigen VHs und Unter-VHs auf. Hier haben zu einem Objekt gehörige VHs die Möglichkeit, spezielle Vorbereitungen vor dem Beginn der Simulation zu treffen. Zum Beispiel stellt die oben genannte VH `MCB_CarLightController` die Liste der von ihr zu steuernden Lichtquellen in ihrer `startup`-Methode zusammen und teilt den Lichtquellen dort auch ihre Aufgaben wie „Blinker“ oder „Bremslicht“ zu.

Der erste Simulationsschritt in einem Simulationsintervall von Frame a bis Frame b ist der Simulationsschritt zu Frame a+1. Da die `startup`-Methoden aller VHs nur vor der Ausführung des ersten Simulationsschrittes ausgeführt werden, wird der `startup`-Methode die Nummer a+1 als aktueller Parameter übergeben.

Auf den genauen Ablauf dieser Methode wird anhand ihres Quellcodes in Kapitel 9.2 ab Seite 167 näher eingegangen.

void update(long f) :

Aktueller Parameter ist n mit $(a \leq n \leq b)$. Diese Methode wird in jedem Simulationszwischen Schritt in jedem Simulationsobjekt aufgerufen. Die in Kapitel 8.6.3 beschriebenen `update`-Methoden aller VHs und Unter-VHs, die zu einem OT gehören, werden durch diese Methode aufgerufen. Dieser Mechanismus bildet den Kern des Simulationsalgorithmus, der in den Kapiteln 8.6 und 9 beschrieben wird. Im Simulationsschritt zu einem Frame n wird in jedem zugehörigen Simulationszwischen Schritt die Nummer n als aktueller Parameter der `update`-Methode übergeben.

Auf den genauen Ablauf dieser Methode wird anhand ihres Quellcodes in Kapitel 9.2 ab Seite 165 näher eingegangen.

8.4.3 Zugriff auf die erstellten Federn

Vector<MCH_Spring> springs :

Die Federn, die mit der in Kap. 8.3.7 erläuterten Methode `createSpring` durch eine VH eines SOs „A“ für die Mechaniksimulation erstellt wurden, werden in diesem Feld des SOs „A“ als Liste des Typs `Vector` gespeichert und können so im Verlauf der Simulation durch die Methoden in `MCH_Spring` (siehe Kap. 10.1.4) verändert werden, z.B. in ihrer Federstärke oder Ruhelänge.

8.5 MCO_Dynamic

Die Klasse `MCO_Dynamic` ist eine direkte Unterklasse von `MCO_WithTransform` und damit eine indirekte Unterklasse von `MC_Object`. Diese Klasse ist wichtig, weil in den meisten Szenen die OTs der simulierten SOs überwiegend Unterklassen von `MCO_Dynamic` sind. Denn wie in Kapitel 9.2 erläutert wird, wird nur das Verhalten von SOs simuliert, deren OTs entweder die Klasse `MCO_Dynamic` selbst oder Unterklassen von `MCO_Dynamic` oder von `MCO_Universe` sind. In Kapitel 7.3 wurde erläutert, dass SOs des Typs

MCO_Universe für die Steuerung globaler Vorgänge konzipiert sind, z.B. für die Berechnung der Massengravitation, die zwischen allen mechanisch simulierten SOs wirkt. Den Hauptteil der Simulation wird dagegen in der Regel die Simulation des individuellen Verhaltens vieler SOs bilden, die vornehmlich ihren eigenen Zustand im Verlauf der Simulation ändern und deren OTs dann Subklassen von MCO_Dynamic sein sollten.

8.5.1 Zugriff auf die Transformationseigenschaften eines Simulationsobjektes

Die wichtigste Eigenschaft der Klasse MCO_Dynamic sind die von MCO_WithTransform geerbten SPs für den Zugriff auf die Transformationseigenschaften. Diese werden im Folgenden erläutert.

```
public final MCP_Position position:
```

Das Feld position referenziert in jedem SO des Typs MCO_Dynamic eine Standard-SP des Typs MCP_Position. Diese speichert einen dreidimensionalen Fließkommawert der Form (x,y,z), der die Position des SOs im Raum als X-, Y- und Z-Koordinaten angibt. Diese Position wird ggf. relativ zum Koordinatensystem eines direkten Elternobjektes gespeichert (siehe Kap. 3.3 und dazu Kap. 3.2). Hat das SO kein Elternobjekt, so wird die Position in Weltkoordinaten angegeben.

```
public final MCP_Rotation rotation:
```

Dieses Feld referenziert analog zu position eine Standard-SP des Typs MCP_Rotation. Diese speichert die Ausrichtung des Objektes im Raum, indem seine Rotation um die X-, Y- und Z-Achse angegeben wird. Diese Art der Repräsentation einer Rotation wird auch als Euler-Winkel bezeichnet, siehe dazu [Jack06]. Auch diese Rotation wird ggf. relativ zum Koordinatensystem eines direkten Elternobjektes gespeichert.

```
public final MCP_Rotation scale:
```

Analog zu position wird eine Standard-SP des Typs MCP_Scale referenziert, die einen dreidimensionalen Fließkommawert der Form (x,y,z), die Skalierung des zugehörigen SOs im Raum, speichert. Auch die Skalierung wird ggf. relativ zum Koordinatensystem eines direkten Elternobjektes angegeben.

8.6 MC_Behaviour

Instanzen der Klasse MC_Behaviour definieren Verhaltensweisen von Simulationsobjekten über die Implementation spezieller Methoden. Eine Verhaltensweise kann SPs und VHs besitzen, daher ist MC_Behaviour eine direkte Unterklasse von MC_Owner und verfügt zunächst über dieselben Felder und Methoden wie MC_Owner. Einige weitere wichtige Felder und Methoden von MC_Behaviour werden hier erläutert.

8.6.1 Aktivieren und Deaktivieren von Verhaltensweisen

```
public final MCP_Boolean behaviourOn:
```

Jede Instanz der Klasse MC_Behaviour referenziert in diesem Feld eine boolesche SP „behaviourOn“, die bestimmt, ob die zugehörige VH bei der Simulation ausgeführt werden soll oder nicht. Ist eine VH abgeschaltet, weil ihre SP „behaviourOn“ den Wert false trägt, so werden auch ihre Unter-VHs nicht ausgeführt. Der Wert von

„behaviourOn“ kann sich im Verlauf der Simulation ändern, entweder durch manuelle Kontrolle (siehe Kap.6.6) oder durch den Einfluss einer VH während der Simulation.

Das Abschalten einer VH bewirkt keine strukturelle Änderung der SO-Hierarchie. Eine abgeschaltete VH bleibt in der SO-Hierarchie erhalten, ebenso wie ihre SPs und ihre Unter-VHs mit deren SPs. Die Werte dieser SPs können weiterhin von anderen aktiven VHs gelesen und geändert werden. Lediglich die unten erläuterte update-Methode der abgeschalteten VH wird in den Simulationszwischenritten nicht mehr ausgeführt, solange die VH abgeschaltet ist.

Das Abschalten einer VH ist sinnvoll, wenn bestimmte Funktionen eines SOs in speziellen Fällen nicht benötigt werden oder nicht erwünscht sind. Ein Fahrzeug habe beispielsweise neben VHs zur grundsätzlichen Steuerung des Fahrzeugs über Lenkrad, Gas- und Bremspedal auch eine VH, welche diese VHs auf höherem Niveau steuert und das Auto so auf einer komplexen Fahrstrecke navigiert. Soll nun in einer ganzen Simulation oder auch nur in Zeitintervallen der Simulation das Fahrzeug nicht automatisch der Strecke folgen sondern sich nur entsprechend der Zustände von Lenkrad, Gas- und Bremspedal verhalten, so kann die höhere VH über die SP „behaviourOn“ ganz oder zeitweise abgeschaltet werden

8.6.2 Zuordnen einer Verhaltensweise zu ihrem Besitzer

MC_Behaviour (MC_Owner _owner) :

Dies ist der Konstruktor von MC_Behaviour. Ihm wird als Parameter der „Besitzer“ übergeben, zu dem die VH gehören soll. Eine Referenz auf diesen Besitzer wird in dem aus MC_Entity geerbten Feld owner gespeichert. Besitzer einer VH kann eine Instanz eines OTs oder eine VH sein.

Für das Zuordnen einer VH zu ihrem Besitzer gelten die gleichen Designvorschriften wie für die SPs (siehe Kap.8.2.1).

Das folgende Codefragment demonstriert diese Vorschriften und ist der Beginn einer Klassendefinition für den Objekttyp MCD_Car aus Abbildung 42, der eine Haupt-VH enthält:

```
public class MCD_Car extends MCD_PhysicalBody
{
    public final MCB_Car mainBehaviour=new MCB_Car(this);
    ...
}
```

Die Klasse MCD_Car wird als Unterklasse von MCD_PhysicalBody definiert. Sie ist als Objekttyp für Fahrzeuge konzipiert und erbt von MCD_PhysicalBody, da Fahrzeuge als Festkörper nach den Gesetzen der Mechanik simuliert werden sollen, was für Objekte des Typs MCD_PhysicalBody zutrifft. Ihr wird die Haupt-VH „MCB_Car“ zugeordnet, die weitere Unter-VHs enthält. Die Haupt-VH eines OTs wird stets einem Container-Feld mit der Bezeichnung „mainBehaviour“ zugewiesen. Wie in Kapitel 7.2.1 erläutert wurde, kann dagegen die Bezeichnung eines public-Feldes, das eine Unter-VH in einer VH referenziert, wie bei SPs frei gewählt werden.

SPs und VHs können alternierend in einer Klassendefinition vorkommen, es kann also z.B. nach einer SP eine VH definiert werden und dann wieder eine weitere SP. Wie in Kapitel 7.2.1 erläutert, werden VHs in diesem Text in der Regel mit dem Namen ihrer zugehörigen Klasse bezeichnet.

Aus der Definition der Klasse `MCD_Car` geht bereits hervor, dass jede Instanz dieser Klasse eine VH des Typs `MCB_Car` besitzen wird, die im Feld `mainBehaviour` referenziert sein wird. Man kann also bereits bei der Definition dieser Klasse sagen, dass der OT „`MCD_Car`“ die VH „`MCB_Car`“ besitzt, obwohl diese Strukturen erst bei einer Instanzierung von „`MCD_Car`“ wirklich entstehen. Wird die Klasse `MCD_Car` nun als Objekt A instanziiert, wird auch eine Instanz B der Klasse `MCB_Car` erzeugt. Dem Konstruktor dieser Instanz B wird als Parameter die Instanz A als ihr Besitzer übergeben. Die so instanziierte VH B wird in ihrem Besitzer A dem Feld `mainBehaviour` zugewiesen.

Wie bereits in Kapitel 7.2.3 erläutert wurde, ist es in MaxControl wie für SPs auch für VHs eine Konvention, jede VH nur einem Container-Feld zuzuweisen, das genau den Typ der VH hat und keinen Obertyp.

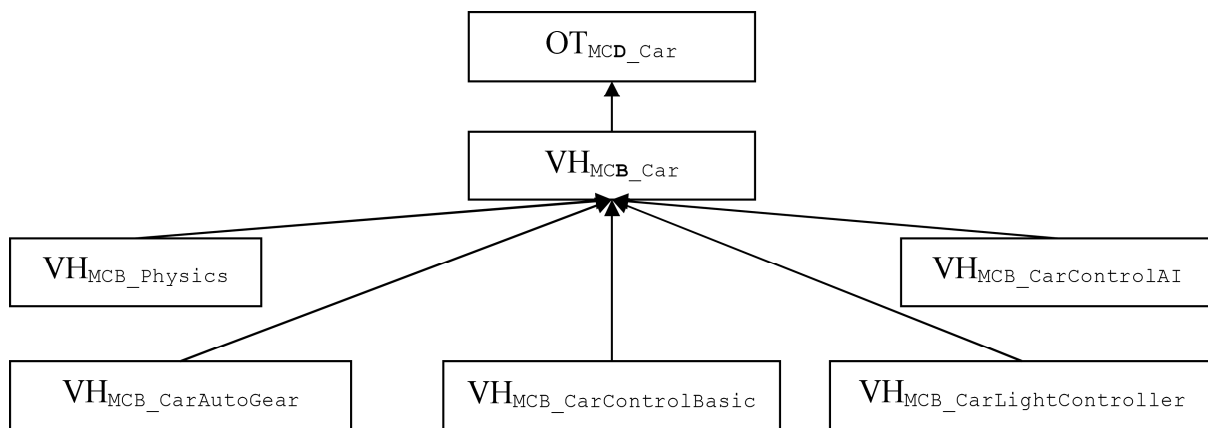


Abbildung 42: OT-Hierarchie zu `MCD_Car` ohne SPs

Das folgende Codefragment ist der Beginn der Klassendefinition für die Verhaltensweise `MCB_Car`, die vier Unter-VHs enthält:

```

public class MCB_Car extends MCB_PhysicalBody
{
    public final MCB_CarControlAI carControlAI=
        new MCB_CarControlAI(this);
    public final MCB_CarAutoGear carAutoGear=
        new MCB_CarAutoGear(this);
    public final MCB_CarControlBasic carControlBasic=
        new MCB_CarControlBasic(this);
    public final MCB_CarLightController carLightController=
        new MCB_CarLightController(this);
    ...
}
  
```

Die Klasse `MCB_Car` wird als Unterklasse von `MCB_PhysicalBody` definiert. In Kap. 7.2.3 wurde bereits erläutert, dass nach den Designvorschriften für MaxControl ein OT A, der von einem OT B erbt, auch eine neue Haupt-VH MA haben soll, die von der Haupt-VH MB des OTs B erbt. Da der OT `MCD_Car` von `MCD_PhysicalBody` erbt, `MCB_Car` die Haupt-VH von `MCD_Car` ist und `MCB_PhysicalBody` die Haupt-VH von `MCD_PhysicalBody`, erbt auch `MCB_Car` von `MCB_PhysicalBody`. `MCB_Car` erbt somit auch alle Unter-VHs von `MCB_PhysicalBody`. Es folgt der dabei relevante Beginn der Klassendefinition zu `MCB_PhysicalBody`:

```
public class MCB_PhysicalBody extends MC_Behaviour
{
    public final MCB_Physics physics=new MCB_Physics(this);
    ...
}
```

Hier ist erkennbar, dass `MCB_PhysicalBody` die Unter-VH `MCB_Physics` besitzt. Jedes 3D-Objekt, dessen OT diese VH direkt oder indirekt (als Unter-VH) besitzt, wird nach den Gesetzen der Mechanik simuliert, wobei andere VHs die Simulation von Mechanik z.B. durch das Setzen von Kräften beeinflussen können (siehe Kap. 9.5 und 8.3.7).

Die VH `MCB_Car` erbt aus `MCB_PhysicalBody` also zunächst die Unter-VH `MCB_Physics` und erhält dann selbst die Unter-VHs `MCB_CarControlAI`, `MCB_CarAutoGear`, `MCB_CarControlBasic` und `MCB_CarLightController`.

`MCB_CarControlAI` übernimmt die höheren Funktionen des Fahrzeugs wie die Navigation auf einer Strecke oder die Entscheidung, mit welcher Geschwindigkeit um eine Kurve gefahren werden soll. Die VH `MCB_CarAutoGear` steuert anhand der Motordrehzahl die Gangschaltung, `MCB_CarControlBasic` setzt die Basissteuerung des Fahrzeugs um, wie z.B. die Erzeugung von Drehmomenten an den Rädern auf der Basis des Zustandes von Gaspedal, Gangschaltung und Kupplung oder das Schwenken der Räder auf der Basis der eingestellten Lenkrichtung, die von `MCB_CarControlAI` bestimmt wird. Die VH `MCB_CarLightController` steuert die Lichtanlage des Fahrzeugs, indem sie z.B. beim Abbiegen den entsprechenden Blinker oder beim Bremsen die Bremsleuchten einschaltet.

Die Reihenfolge, in der diese VHs bei der Simulation ausgeführt werden, hängt von der Reihenfolge ab, in der ihre Container-Felder im Besitzer deklariert werden, welche dort diese VHs referenzieren. Im obigen Beispiel wird bei der Simulation der VH „`MCB_Car`“ zuerst die Unter-VH „`MCB_Physics`“, dann die VH „`MCB_CarControlAI`“, danach die VH „`MCB_CarAutoGear`“, dann die VH „`MCB_CarControlBasic`“ und schließlich die VH „`MCB_CarLightController`“ simuliert. Auf die genaue Arbeitsweise der Simulation in Bezug auf VHs und Unter-VHs wird in Kapitel 9.2 näher eingegangen.

Die VHs werden auch als Gruppierungen ihrer SPs in der grafischen Benutzeroberfläche des verwendeten 3D-Animationsprogramms angezeigt, daher bestimmt die oben genannte Deklarationsreihenfolge der `public`-Felder, die im Besitzer auf diese VHs verweisen, auch die Reihenfolge, in der diese VHs als Gruppierungen in der Benutzeroberfläche angezeigt werden. Die Hierarchie der VHs und Unter-VHs wird dabei jedoch in der konkreten technischen Umsetzung nicht hierarchisch angezeigt, sondern als **eine** Gesamtreihenfolge von VHs und Unter-VHs.

8.6.3 Definition einer Verhaltensweise

Die im Folgenden erläuterten Methoden sind `startup`, `update`, `updateBeforeSubBehaviours` und `updateAfterSubBehaviours`. Bis auf `update` tragen sie in der Klasse `MC_Behaviour` noch keinen eigenen Programmcode. Sie sind vielmehr die Rahmen für eine Verhaltensweise, die ein Benutzer von `MaxControl` als Unterklasse von `MC_Behaviour` implementieren kann.

void startup(long f) :

Diese Methode wird, wie in Kapitel 8.4.2 zur gleichnamigen Methode beschrieben, vor dem Beginn einer Simulation **indirekt** in jeder Verhaltensweise jedes Simulationsobjekts mit dem aktuellen Parameter $a+1$ aufgerufen. Dieser Aufruf erfolgt direkt über die unten erläuterte Methode `startupAll`. In `startup` hat die VH die Möglichkeit, spezielle Vorbereitungen **vor** dem Beginn einer Simulation zu treffen. In der Klasse `MC_Behaviour` führt diese Methode keine Befehle aus, sie wird im Bedarfsfall von Unterklassen so implementiert, dass sie die gewünschten Vorbereitungen trifft.

void startupAll (long f) :

Diese Methode wird vor dem Beginn einer Simulation **direkt** in jeder Verhaltensweise jedes Simulationsobjekts mit dem aktuellen Parameter $a+1$ aufgerufen. Sie ruft selbst die oben beschriebene Methode `startup` in der zugehörigen VH und die Methode `startupAll` in allen zugehörigen direkten Unter-VHs auf.

Auf den genauen Ablauf dieser Methode wird anhand ihres Quellcodes in Kapitel 9.2 ab Seite 167 näher eingegangen.

void update(long f) :

Aktueller Parameter ist n mit $(a \leq n \leq b)$. Diese Methode wird in jedem Simulationszwischen schritt zu Frame n für jedes Simulationsobjekt in jeder seiner VHs ausgeführt, sofern diese jeweils eingeschaltet ist. Bei der Ausführung ihrer `update`-Methode bewirkt jede VH die Zustandsänderungen für das Zeitintervall $\Delta t(n)$, die für diese VH spezifiziert wurden. Die genauen Mechanismen dieser Simulation werden in Kapitel 9 näher beschrieben.

Die Methode `update` startet zunächst eine weitere Methode der betrachteten VH durch den Aufruf „`updateBeforeSubBehaviours(f) ;`“. Dann wird eine Rekursion in die Unter-VHs dieser VH ausgeführt, indem die `update`-Methoden dieser Unter-VHs der Reihe nach aufgerufen werden. Danach wird eine weitere Methode der VH durch den Aufruf „`updateAfterSubBehaviours(f) ;`“ gestartet. Dadurch ist es für eine VH möglich, bestimmte Zustandsänderungen sowohl **vor** ihren Unter-VHs durchzuführen als auch **danach**. Dies kann sinnvoll sein, wenn eine VH zunächst Daten wie z.B. Werte von SPs für ihre Unter-VHs aufbereiten soll, was sie in der Methode `updateBeforeSubBehaviours` tun kann, oder wenn sie auf Zustandsänderungen, die von ihren Unter-VHs durchgeführt wurden, noch in demselben Simulationszwischen schritt reagieren soll, was dann in der Methode `updateAfterSubBehaviours` möglich ist.

So wird definiert, welche Anweisungen die VH in jedem Simulationszwischen schritt ausführt. Ist die VH Teil der OT-Hierarchie zu einem OT „X“, so bestimmt sie zusammen mit den übrigen VHs von „X“ das Verhalten aller 3D-Objekte, denen dieser OT „X“ zugeordnet wurde.

Auf den genauen Ablauf der Methode `update` wird anhand ihres Quellcodes in Kapitel 9.2 ab Seite 166 näher eingegangen.

void updateBeforeSubBehaviours(long f):

Die Methode `updateBeforeSubBehaviours` wird am häufigsten zur Definition von Verhaltensweisen verwendet und stellt damit die wichtigste Methode von `MaxControl` dar. In der Klasse `MC_Behaviour` führt diese Methode keine Befehle aus, sie kann in Unterklassen von `MC_Behaviour` und damit in neu entwickelten Verhaltensweisen passend implementiert werden.

In der Regel kann man eine VH aus drei Hauptaufgaben aufbauen, die der Reihe nach ausgeführt werden.

Zuerst sollte eine VH den aktuell vorliegenden `MaxControl`-Zustand betrachten und benötigte Daten nach Möglichkeit in lokale Variable übernehmen. Dabei hat die VH im Allgemeinen Zugriff auf Daten, die als Werte von SPs, über Methoden oder als Felder verfügbar sind. Nicht über SPs sondern über Methoden und Felder sind beispielsweise spezielle Daten verfügbar wie die Namen der SOs in der Szene oder die Liste aller SOs sowie deren hierarchische Einteilung durch Eltern-Kind-Beziehungen (siehe Kap. 3.3) oder Daten über Kollisionsereignisse (siehe Kap. 8.3.7 und 10.1.5). In diesem ersten Schritt sollte die VH also in der Regel Daten lesen.

Für den zweiten Schritt wird empfohlen, dass die VH dann auf der Basis dieser Daten die im aktuellen Simulationszwischen Schritt erforderlichen Zustandsänderungen berechnet. Dabei sollte sich eine VH eines OTs, der von der Klasse `MCO_Dynamic` abstammt, auf Änderungen des Zustandes des zugehörigen SOs beschränken und nicht den gesamten `MaxControl`-Zustand ändern, obwohl dies möglich ist. OTs, die von der Klasse `MCO_Universe` (siehe Kap. 7.3 und 8.5) abstammen, sind dagegen dafür konzipiert, mit ihren VHs auf den gesamten `MaxControl`-Zustand zu wirken, da sie das globale Verhalten der Szene steuern sollen, z.B. durch die Simulation von Massengravitation oder Wind, die auf alle mechanischen Objekte in der Szene wirken. Dieser zweite Schritt sollte die Änderungen zunächst nur berechnen, diese aber noch nicht durchführen.

Erst im dritten Schritt sollten diese Zustandsänderungen dann ausgeführt werden, indem die Werte der entsprechenden SPs geändert werden.

Ein Beispiel einer einfachen VH soll diese drei Schritte verdeutlichen:

```
public class MCB_SimpleMotionMover extends MC_Behaviour
{
    public final MCP_Double xUnitsPerSecond=new MCP_Double ("X-UPS", this);
    public final MCP_Double yUnitsPerSecond=new MCP_Double ("Y-UPS", this);
    public final MCP_Double zUnitsPerSecond=new MCP_Double ("Z-UPS", this);

    public MCB_SimpleMotionMover(MC_Owner _owner)
    {
        super(_owner);
    }
}
```

```

public void updateBeforeSubBehaviours(long f)
{
    double x=getXValue("position");
    double y=getYValue("position");
    double z=getZValue("position");
    double xUnitsPerSecond=getDoubleValue("xUnitsPerSecond");
    double yUnitsPerSecond=getDoubleValue("yUnitsPerSecond");
    double zUnitsPerSecond=getDoubleValue("zUnitsPerSecond");

    x+=xUnitsPerSecond*getDeltaT();
    y+=yUnitsPerSecond*getDeltaT();
    z+=zUnitsPerSecond*getDeltaT();

    setXValue("position",x);
    setYValue("position",y);
    setZValue("position",z);
}
}

```

MCB_SimpleMotionMover ist eine Unter-VH von MCB_SimpleMotion, welche die Haupt-VH von MCD_SimpleMotion ist, einer Unterklasse von MCO_Dynamic. Von MCO_Dynamic werden die SPs „position“, „rotation“ und „scale“ (siehe Kap. 8.5.1) geerbt. Über die SP „position“ kann MCB_SimpleMotionMover nun in einem Simulationszwischen schritt die aktuelle Position des zugehörigen SOs verändern.

Eine Instanz von MCB_SimpleMotionMover soll das zugehörige 3D-Objekt in der Richtung und Geschwindigkeit eines einstellbaren Geschwindigkeitsvektors bewegen. Um diesen Geschwindigkeitsvektor von außen einstellbar zu machen, hat diese VH die SPs „xUnitsPerSecond“, „yUnitsPerSecond“ und „zUnitsPerSecond“, welche in den drei Dimensionen des übergeordneten Koordinatensystems die gewünschte Geschwindigkeit speichern. Da diese SPs durch die VH MCB_SimpleMotionMover nicht verändert werden, tragen sie immer den Wert, der im ersten Frame des Animationsintervalls eingestellt wurde, sofern sie nicht im Simulationsverlauf durch andere VHs verändert werden oder stellenweise mit anderen vorgegebenen Werten als manuell kontrolliert eingestellt werden. Wird z.B. die Möglichkeit der manuellen Kontrolle genutzt, so wird das SO seine Bewegungsgeschwindigkeit an den entsprechenden Stellen des Simulationsintervalls wie vorgegeben ändern.

In einem Simulationszwischen schritt wird nun für jedes SO des Typs „MCD_SimpleMotion“, das sich in der simulierten Szene befindet, die Methode updateBeforeSubBehaviours seiner VH MCB_SimpleMotionMover aufgerufen. Diese führt die drei oben empfohlenen Hauptschritte aus, welche im Programmcode mit „Schritt ①“ bis „Schritt ③“ bezeichnet sind.

In Schritt ① liest die VH alle für Schritt ② relevanten Werte aus SPs und speichert sie in lokale einfache Variablen, um bequemer mit diesen Werten arbeiten zu können. Die SP „position“ gehört direkt zum OT und nicht zu der VH MCB_SimpleMotionMover, welche die Anweisungen in Schritt ① ausführt. Die hierbei verwendeten Methoden zum Lesen von Werten aus SPs führen automatisch die in Kapitel 8.3.3 erläuterte Baumsuche nach SPs durch. Daher ist es für den Zugriff auf die SP „position“ nicht erforderlich, ihre genaue Lage in der zugehörigen OT-Hierarchie anzugeben. Es wird empfohlen, zur besseren Wiedererkennung die Werte der SPs in lokalen Variablen zu speichern, welche exakt die gleiche Bezeichnung erhalten, wie das public-Feld, das diese SP in ihrem Besitzer referenziert. Für SPs wie „position“, die mehrere Komponenten speichern, ist dies jedoch nicht möglich, wenn die Komponenten in einfachen Variablen gespeichert werden sollen. Für

die Speicherung der X-, Y- und Z-Komponente dieser SP wurden hier deshalb drei lokale Variablen des Typs `double` eingeführt, die als `x`, `y` bzw. `z` bezeichnet wurden. Für die verwendeten SP-Zugriffsmethoden wie `getDoubleValue` stellt es **kein** Problem dar, wenn ein Feld wie `xUnitsPerSecond` durch die gleichnamige lokale Variable `xUnitsPerSecond` überdeckt wird.

Für den Schritt ② stehen nun die Werte aller relevanten SPs in einfachen Variablen zur Verfügung, so dass für die Verarbeitung dieser Werte nun übersichtlicher und effizienter Programmcode geschrieben werden kann. In den meisten Fällen kann der komplexere Zugriff auf SPs über Methoden, wie er in Schritt ① und ③ verwendet wird, in Schritt ② vermieden werden. Wie hier noch erläutert wird, ist dies jedoch nicht immer möglich.

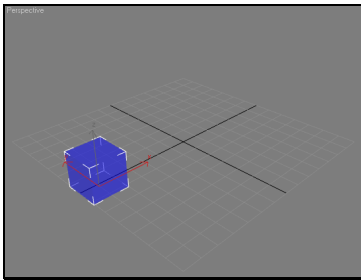
Der Schritt ② führt nun für die X-, Y- und Z-Komponenten der Position des SOs jeweils einen Befehl wie `x+=xUnitsPerSecond*getDeltaT()`; durch. Der Befehl ist eine Zuweisung an die aktuelle X-Position des SOs, die lokal in der Variablen `x` gespeichert wurde. Die Zuweisung bewirkt die Erhöhung des Wertes von `x` um den Wert des rechts stehenden Ausdrucks. Wird der Wert der X-Position in aufeinanderfolgenden Simulationszwischenritten auf diese Weise inkrementiert, so wird sich das 3D-Objekt in der Szene entsprechend entlang der X-Achse bewegen. Dabei ist zu beachten, dass der Wert des hinzuaddierten Ausdrucks auch negativ sein kann und sich das 3D-Objekt dann in die entgegengesetzte Richtung bewegt. Die gewünschte Geschwindigkeit in X-Richtung ist in der lokalen Variablen `xUnitsPerSecond` gespeichert. Sie gibt an, um wie viele Einheiten sich das SO in jeder Sekunde in der X-Richtung bewegen soll. Ein Simulationszwischenritt bezieht sich auf das zugeordnete Zeitintervall Δt . Wie in Kapitel 8.1.2 erläutert wurde, liefert die Methode `getDeltaT` aus der Klasse `MC_Entity` stets die Länge dieses Zeitintervalls in Sekunden, die sich im Laufe der Simulation auch ändern kann. Deshalb sollte man sie nicht lokal speichern, sondern sie in jedem Simulationszwischenritt erneut aus dieser Methode beziehen. Multipliziert man nun die aktuell gewünschte Geschwindigkeit `xUnitsPerSecond` mit dem aktuellen Wert von `getDeltaT()`, ergibt sich die Wegstrecke, die das SO in X-Richtung in diesem Simulationszwischenritt zurücklegen soll, so dass diese Wegstrecke zur aktuellen X-Position des SOs hinzuaddiert werden kann. Dieser Schritt wird analog für die Y- und Z-Position des SOs durchgeführt.

Die neue Position des SOs ist zunächst in den lokalen Variablen `x`, `y` und `z` gespeichert. Dadurch ändert sich die Position des SOs noch nicht. Die Werte der entsprechenden SPs müssen dazu in Schritt ③ geändert werden.

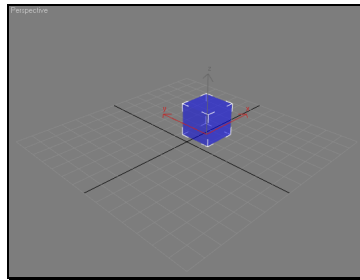
Dabei kann man sich auf die SPs beschränken, deren Werte wirklich geändert wurden. Im Beispiel ist nur die SP „position“ betroffen. Die Anweisungen `„setXValue(„position“,x);“` bis `„setZValue(„position“,z);“` übertragen die neuen Werte für die X-, Y- und Z-Position des SOs auf die entsprechende Standard-SP „position“. Eine Änderung dieser SP hat Einfluss auf das Objekt in der Szene, so dass es sich wie gewünscht bewegt.

So kann mit entsprechenden Einstellungen z.B. die in Abbildung 43 gezeigte Animation erzeugt werden:

Frame 0:



Frame 50:



Frame 100:

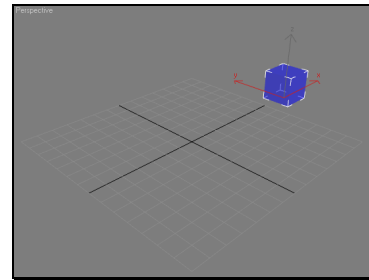


Abbildung 43: Beispielanimation zu „MCD_SimpleMotion“

Die Abbildung 44 zeigt für dieses Beispiel die GUI-Repräsentation eines SOs vom Typ „MCD_SimpleMotion“ mit passenden SP-Einstellungen. Das SO wurde so eingestellt, dass es sich in jeder Sekunde um 45 Einheiten in Richtung der X-Achse bewegt. Weil für das SO kein Elternobjekt in der Szene festgelegt wurde, führt es diese Bewegung im Weltkoordinatensystem durch, es bewegt sich also in Richtung der X-Achse des Weltkoordinatensystems. Diese Animation verwendet eine Framerate von 30 Bildern pro Sekunde. Bei dieser Framerate entsprechen 100 Filmbilder einer Simulationszeit von $100/30=3\frac{1}{3}$ Sekunden. Bei Bild 100 hat das Objekt also in X-Richtung eine Strecke von $45 \cdot 3\frac{1}{3}=150$ Einheiten zurückgelegt. Dabei ist zu beachten, dass die Zählung der Frames meistens bei 0 beginnt.

Bei Schritt ② muss grundsätzlich beachtet werden, dass das direkte Arbeiten mit einfachen Variablen zu Konflikten bezüglich der manuellen Kontrolle der ursprünglichen SPs führen kann. Denn während eine SP, deren MP momentan den Wert „true“ hat, keinen neuen Wert annimmt, kann die für sie in Schritt ① eingeführte einfache Variable in diesem Fall trotzdem geändert werden. Wird der Wert dieser Variablen in Schritt ③ wieder der zugrunde liegenden SP zugewiesen, so wird diese SP den neuen Wert zwar wie gewünscht nicht annehmen, jedoch können unerwünschte Seiteneffekte auftreten.

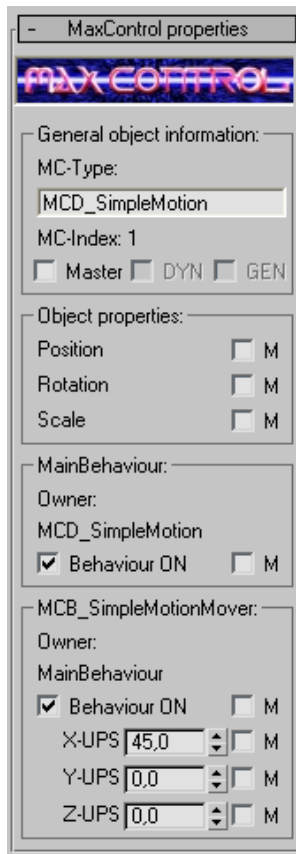


Abbildung 44: Beispieleinstellungen zu „MCD_SimpleMotion“

Es seien drei Double-SPs „a“, „b“ und „c“ einer VH in einem aktuellen Frame n betrachtet, von denen allein b im aktuellen Frame manuell kontrolliert sei. In Schritt ① seien die Werte der SPs bereits den lokalen double-Variablen a , b bzw. c zugewiesen worden.

Schritt ② führe nun die folgenden Anweisungen aus:

```
a+=1d;
b=a+1d;
c=b+1d;
```

Die Zuweisung $b=a+1d$; hätte eigentlich nicht durchgeführt werden dürfen, weil die SP „b“ im das Simulationsintervall zu Frame n manuell kontrolliert ist und ihr Wert daher nicht geändert werden darf. Dies wäre jedoch nicht von Bedeutung, wenn der Wert der lokalen Variable b nach dieser Zuweisung in Schritt ② nicht mehr gelesen werden würde. In Schritt ③ würde dann zwar der fälschlich geänderte Wert der Variablen b an die SP „b“ übergeben werden, diese würde den neuen Wert jedoch nicht annehmen, sondern den Wert behalten, den sie bereits in Schritt ① hatte.

In dem hier gezeigten Beispiel hat die Zuweisung $b=a+1d$; jedoch einen Seiteneffekt auf die Zuweisung $c=b+1d$; . Der Wert, welcher der Variablen c zugewiesen wird, ist abhängig vom aktuellen Wert der Variablen b , deren Wert fälschlich geändert wurde. Die SP „c“ ist im aktuellen Simulationsschritt nicht manuell kontrolliert und kann daher den neuen Wert der Variablen c annehmen. Dieser Wert beruht auf der fehlerhaften Zuweisung $b=a+1d$; und muss damit als nicht korrekt angesehen werden.

Um solche Fälle zu vermeiden, muss jeweils die letzte Zuweisung an eine lokale Hilfsvariable, die vor einem lesenden Zugriff auf diese Variable ausgeführt wird, die Zuweisungsmethoden für die entsprechende SP verwenden. Das heißt für unser Beispiel:

```
a+=1d;  
b=setDoubleValue("b", a+1d);  
c=b+1d;
```

Auf diese Weise wird die Zuweisung an die lokale Variable `b` mit einer Zuweisung an die SP „`b`“ verbunden. Zunächst wird der Wert des Ausdrucks `a+1d` der SP „`b`“ übergeben. Sie nimmt diesen Wert jedoch nicht an sondern behält ihren bisherigen Wert, der in dem aktuellen Frame `n` manuell festgelegt wurde. Dieser unveränderte Wert ist, wie in Kapitel 8.3.5 erläutert, auch der Rückgabewert der Methode `setDoubleValue`. Der unveränderte Wert wird daher auch der lokalen Hilfsvariablen `b` zugewiesen, weshalb die folgende Anweisung `c=b+1d` den korrekten unveränderten Wert für `b` verwendet und die Variable `c` daher auch den korrekten darauf basierenden Wert erhält.

Möchte man den vorhandenen Programmcode möglichst geringfügig anpassen, kann man sich auch darauf beschränken, nur entsprechende Absicherungsbefehle in den Code einzufügen und dabei keine bereits vorhandenen Anweisungen zu verändern, wie die folgende Version unseres Beispiels zeigt:

```
a+=1d;  
b=a+1d;  
b=setDoubleValue("b", b);  
c=b+1d;
```

Der vorher vorhandene Code wurde hier nur durch das Einfügen einer neuen Befehlszeile „`b=setDoubleValue("b", b);`“ modifiziert. Sie weist der SP „`b`“ den neuen Wert der Variablen `b` zu, den die SP jedoch nicht akzeptiert, so dass die Variable `b` schließlich wieder den unveränderten Wert der SP „`b`“ erhält, bevor in der folgenden Anweisung „`c=b+1d;`“ auf diese Variable `b` zugegriffen wird.

Um bei Codeausführungen die manuelle Kontrolle korrekt berücksichtigen zu können, kann die in Kapitel 8.3.5 erläuterte Methode `getManualValue` verwendet werden, wie die folgende Abwandlung des Beispiels zeigt:

```
a+=1d;  
if (getManualValue("b"))  
{  
    b=a+1d+einZeitaufwändigerAlgorithmus();  
}  
c=b+1d;
```

Sei „`einZeitaufwändigerAlgorithmus()`“ eine Methode, deren Ausführung unter Umständen viel Rechenzeit benötigt. Ihr Aufruf kann wie oben gezeigt in dem Fall umgangen werden, dass die SP „`b`“ manuell kontrolliert ist und den Rückgabewert dieser Methode ohnehin nicht speichern würde. Der in Schritt ① gelesene manuell festgelegte Wert für `b` bleibt dann erhalten und wird für `c=b+1d;` verwendet.

Möchte man die hier erläuterten möglichen Fehlerstellen in Bezug auf die Berücksichtigung von MPs sicher ausschließen, ohne Abhängigkeiten im Programmcode suchen zu müssen,

kann man alternativ natürlich auch ohne lokale Variablen arbeiten und jeden Zugriff auf SPs direkt über die dafür vorgesehenen Methoden vornehmen. Der Programmcode wird dadurch sehr viel unübersichtlicher.

Dabei ist ebenfalls zu beachten, dass die verwendeten Zugriffsmethoden für SPs wie in Kapitel 8.3.5 erläutert ggf. jeweils eine Baumsuche nach der SP durchführen, weshalb diese Art des Umgangs mit SPs meistens ineffizienter ausgeführt wird als die Herangehensweise mit lokalen Hilfsvariablen.

void updateAfterSubBehaviours(long f):

Diese Methode wird im Vergleich zu `updateBeforeSubBehaviours` nur selten implementiert. Auch sie führt in der Klasse `MC_Behaviour` keine Befehle aus. Wird sie in einer VH implementiert, so führt sie ihre Anweisungen nach den Unter-VHs dieser VH aus und kann noch in demselben Simulationszwischenstadium auf Zustandsänderungen reagieren, welche die Unter-VHs durchgeführt haben. Auch für die Implementierung dieser Methode wird die Einhaltung der oben genannten Schritte ① bis ③ empfohlen.

Als Beispiel für eine Nutzung dieser Möglichkeit sei hier in Teilen die Verhaltensweise `MCB_2DNavigation` erläutert, die zur Steuerung der Bewegungsrichtung eines Roboters verwendet wird. Der VH-Typ `MCB_2DNavigation_with_Avoider2D` ist ein Subtyp von `MCB_2DNavigation` und erbt von diesem unverändert sowohl die Methode `updateBeforeSubBehaviours` als auch die Methode `updateAfterSubBehaviours`. Zusätzlich hat er die Unter-VH `MCB_Avoider2D`. Diese vermeidet die Kollision des gesteuerten Roboters mit anderen Robotern.

Es folgt ein Auszug aus der Klassendefinition von `MCB_2DNavigation`, aus der die zu erläuternden Klasse `MCB_2DNavigation_with_Avoider2D` die hier relevanten Teile erbt:

```
public class MCB_2DNavigation extends MC_Behaviour
{
    ...
    public final MCP_SceneObjectNA nGoal=...
    ...
    public final MCP_Double t2Angle=...
    public final MCP_Double nAngle=...
    public final MCP_Double angleSpeed=...

    ...

    public void updateBeforeSubBehaviours(long f)
    {
        ...
        MC_Object nGoal=getMC_ObjectValue("nGoal");
        ...
        double t2Angle=getDoubleValue("t2Angle");
        double nAngle=getDoubleValue("nAngle");

        ... //Berechnung des neuen Ziel-Navigationswinkels

        setDoubleValue("t2Angle", t2Angle);
    }
}
```

```

    ...
}

public void updateAfterSubBehaviours(long f)
{
    double t2Angle=getDoubleValue("t2Angle");
    double nAngle=getDoubleValue("nAngle");
    double angleSpeed=getDoubleValue("angleSpeed");

    nAngle=MC_C.rotateAngleTowards(nAngle,t2Angle,
                                   angleSpeed*getDeltaT());
    setDoubleValue("nAngle",nAngle);
}
}

```

Abbildung 45: Auszug aus der Klasse MCB_2DNavigation

Die VH MCB_2DNavigation hat eine SP „nGoal“ des Typs MCP_SceneObjectNA. SPs dieses Typs sind nicht animierbar und speichern eine Referenz auf ein SO. Die SP „nGoal“ wird hier verwendet, um ein Navigationsziel für den Roboter einstellen zu können. Das Navigationsziel kann ein beliebiges 3D-Objekt in der Szene sein, von dem allerdings die Position als SP zugänglich sein muss, es kann z.B. ein SO des Typs „MCO_Dynamic“ verwendet werden. Die Navigations-VH bestimmt nun einen „Navigationswinkel“, der sich um die Z-Achse des Weltkoordinatensystems dreht, die vom Boden weg zeigt, und der bei der X-Achse des Weltkoordinatensystems beginnend entgegen dem Uhrzeigersinn gemessen wird (siehe Abbildung 46). Dieser Winkel gibt die Richtung an, in die der Roboter laufen muss, um das Navigationsziel zu erreichen.

Dieser „Ziel-Navigationswinkel“ wird in der SP „t2Angle“ gespeichert. Die SP „nAngle“ enthält den momentan verwendeten Navigationswinkel, in dessen Richtung der Roboter läuft. Damit der Roboter keine zu abrupten Richtungswechsel ausführt, wird der in nAngle gespeicherte Winkel in jedem Simulationszwischen schritt mit einer bestimmten Geschwindigkeit dem Ziel-Navigationswinkel t2Angle angenähert. Die dabei verwendete Annäherungsgeschwindigkeit wird in Grad pro Sekunde in der SP „angleSpeed“ gespeichert.

Die Methode updateBeforeSubBehaviours führt die folgenden Abläufe durch: Entsprechend dem Schritt ① (siehe Methode updateBeforeSubBehaviours weiter oben) werden zunächst alle oben erläuterten und auch die hier nicht erwähnten SPs von MCB_2DNavigation in lokale Variablen zwischengespeichert. Der Schritt ② wird in der obigen Abbildung 45 nicht gezeigt. Er berechnet anhand der in Schritt ① gelesenen SPs den erforderlichen Ziel-Navigationswinkel, zunächst ohne Berücksichtigung von Hindernissen, und schreibt diesen Winkel in Schritt ③ in die zugehörige SP „t2Angle“. In diesem Schritt werden auch Werte anderer lokaler Variablen zurück in ihre SPs geschrieben, was hier jedoch nicht gezeigt wird.

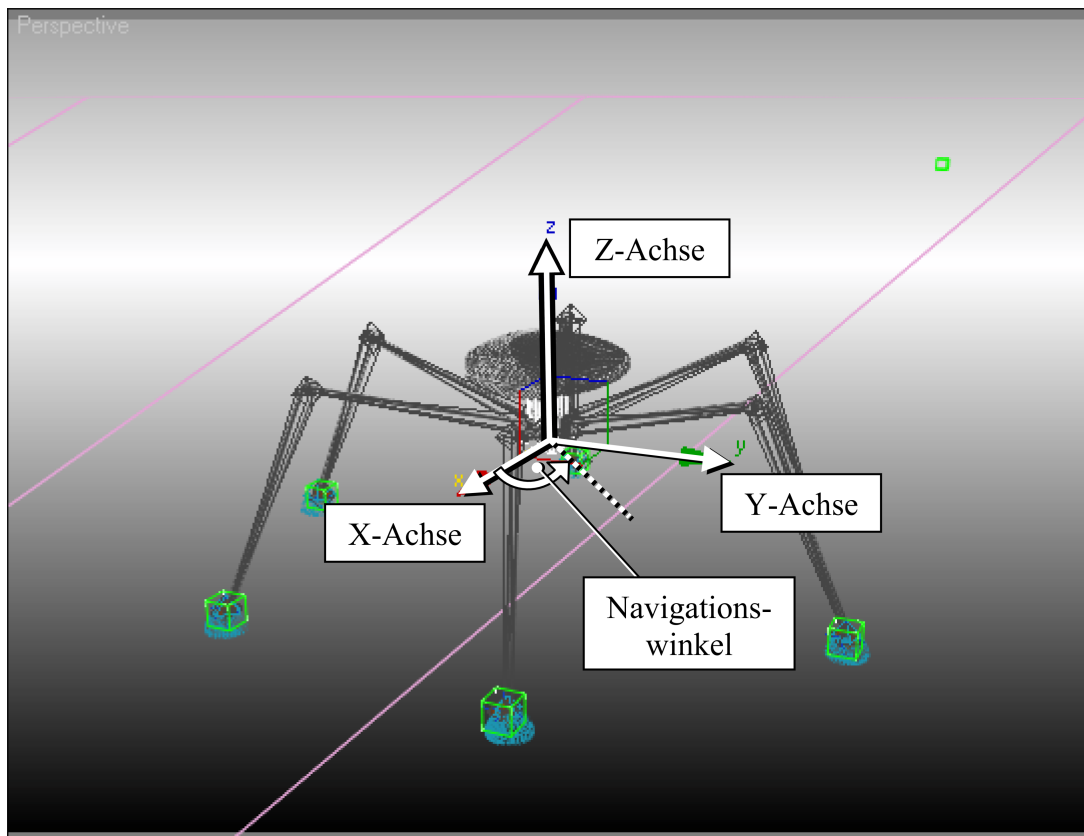


Abbildung 46: Illustration des Navigationswinkels für „MCB_2DNavigation“

Wir betrachten nun die Unterklasse `MCB_2DNavigation_with_Avoider2D`. Sie besitzt wie oben erwähnt zusätzlich die Unter-VH `MCB_Avoider2D`. Diese Verhaltensweise wird ausgeführt, nachdem die unverändert aus `MCB_2DNavigation` geerbte im vorangehenden Absatz erläuterte Methode `updateBeforeSubBehaviours` aufgerufen wurde.

Die VH `MCB_Avoider2D` liest in Schritt ① unter anderem den Inhalt von `t2Angle` in eine lokale Variable. Der hier nicht erläuterte Schritt ② stellt mögliche Kollisionen mit anderen Robotern fest, die vermieden werden müssen und berechnet ggf. einen neuen Navigationswinkel für das korrekte Ausweichen. Ist der Roboter eingeschlossen, wird er über die hier nicht dokumentierte SP `„nStop“` angehalten. In Schritt ③ werden die SPs auf die neuen Werte gesetzt, also erhält `„t2Angle“` einen neuen Wert, wenn ein Ausweichmanöver erforderlich ist und die SP `„nStop“` erhält den Wert `„true“`, wenn der Roboter warten muss.

Nun ist die VH `„MCB_2DNavigation_with_Avoider2D“` mit ihrer aus `„MCB_2DNavigation“` geerbten Methode `updateAfterSubBehaviours` in der Lage, noch in dem aktuellen Simulationszwischen Schritt auf diesen vielleicht neuen Navigationswinkel zu reagieren. Sie liest in Schritt ① die SP `„t2Angle“` für den Ziel Navigationswinkel, die SP `„nAngle“` für den momentanen Navigationswinkel und die SP `„angleSpeed“` für die Geschwindigkeit, mit der `nAngle` an `t2Angle` angenähert werden soll in lokale Variablen. Durch die entsprechende Anweisung aus Abbildung 45 wird in Schritt ② diese Annäherung für den aktuellen Simulationszwischen Schritt durchgeführt. Der so neu bestimmte Navigationswinkel wird dann in Schritt ③ in die zugehörige SP `nAngle` geschrieben.

Dies ist in der Methode `updateAfterSubBehaviours` und nicht erst im nächsten Simulationszwischenstschritt zu tun ist empfehlenswert, weil die VH `MCB_6LegWalk`, eine Geschwister-VH von `MCB_2DNavigation_with_Avoider2D`, zur Steuerung des Gehens in jedem Simulationszwischenstschritt nach der hier beschriebenen VH `MCB_2DNavigation_with_Avoider2D` ausgeführt wird und den in `nAngle` gespeicherten Wert zur Richtungsbestimmung verwendet, der möglichst aktuell sein sollte.

9 Festlegung und Diskussion der Simulationsvorschrift

9.1 Prinzipien

Wie in den Kapiteln 6.5 und 6.7 erläutert wurde, erzeugt bei gegebener vorläufiger Szenenzustandsfolge jeder von MaxControl durchgeführte Simulationsschritt zu einem Frame n den jeweils nächsten neuen Szenenzustand δ_n' in der endgültigen Szenenzustandsfolge. So entsteht automatisch eine Animation, die durch die entstehende Szenenzustandsfolge $(\delta_a', \dots, \delta_b')$ repräsentiert wird. Jeder Simulationsschritt kann dabei in mehreren Simulationszwischenritten durchgeführt werden, wodurch Zwischenzustände entstehen. Eine exakte Definition der zugehörigen Simulationsfunktion sim steht noch aus und wird im Folgenden nachgeholt.

Wir beziehen uns dabei auf die Betrachtung der iterativen Lösung von linearen Gleichungssystemen mit n Variablen y_1, \dots, y_n . Dort stehen Gesamtschrittverfahren (Jacobi) und Einzelschrittverfahren (Gauß-Seidel) zur Auswahl [Stoer73]. Im ersten Fall lässt sich die Iterationsfunktion in der Form

$$y_i^{j+1} = \Phi_i(y_1^j, \dots, y_n^j), i = 1, \dots, n, j = 1, 2, 3 \dots$$

angeben. Hier werden die im aktuellen Iterationsschritt bereits bestimmten Werte $y_1^{j+1}, \dots, y_{i-1}^{j+1}$ ignoriert und stets die Werte y_1^j, \dots, y_{i-1}^j aus dem vorangehenden Iterationsschritt verwendet. Die Reihenfolge, in der die $y_i^{j+1}, i=1, \dots, n$, berechnet werden, ist beliebig.

Im zweiten Fall nutzt man sukzessiv die bisher neu bestimmten Werte $y_1^{j+1}, \dots, y_{i-1}^{j+1}$ für die Bestimmung von y_i^{j+1} aus, so dass man eine Iterationsfunktion der folgenden Form erhält:

$$y_i^{j+1} = \Phi_i(y_1^{j+1}, \dots, y_{i-1}^{j+1}, y_i^j, \dots, y_n^j), i = 1, \dots, n, j = 1, 2, 3 \dots$$

Die Simulationsfunktion von MaxControl stützt sich auf die Idee des Einzelschrittverfahrens. Bei jedem Iterationsschritt, also einem Simulationszwischenritt in der Sprechweise für MaxControl, werden die VHs aller SOs einer gegebenen Szene in einer festen Reihenfolge mit dem zugehörigen zu simulierenden Zeitintervall Δt aufgerufen. Damit werden die Werte aller SPs in einer für diesen Zwischenritt festen Reihenfolge neu bestimmt, wobei die im betrachteten Iterationsschritt schon geänderten Werte berücksichtigt werden. Als Erweiterung gegenüber den linearen Gleichungssystemen kann eine SP in einem Iterationsschritt auch mehrfach mit verschiedenen Funktionen Φ_i neu bestimmt werden. Es sei darauf hingewiesen, dass Einzelschrittverfahren in Simulationssystemen für 3D-Animationen durchaus üblich sind.

In Kapitel 9.4.6 wird die Idee eines entsprechenden Gesamtschrittverfahrens diskutiert. Bei diesem Verfahren muss vorausgesetzt werden, dass in einem Iterationsschritt jede SP höchstens einmal neu bestimmt wird.

9.2 Ausführungsreihenfolge der Verhaltensweisen

Jeder Simulationszwischenritt führt in einer bestimmten Reihenfolge die update-Methoden der Verhaltensweisen aller SOs genau einmal aus. Die dadurch insgesamt hervorgerufene Zustandsänderung überführt einen Zustand $\sigma_{n-1,j-1}$ in den Nachfolger-Zustand $\sigma_{n-1,j}$. Die Reihenfolge, in der die SOs in einem Simulationszwischenritt behandelt werden,

indem die `update`-Methoden ihrer Verhaltensweisen ausgeführt werden, hängt davon ab, wie die gegebene 3D-Szene als Hierarchie im 3D-Animationsprogramm gespeichert ist.

Bisher sind in MaxControl die für SOs verfügbaren Objekttypen Unterklassen von `MCO_Dynamic` oder von `MCO_Universe`. `MCO_Dynamic` kann auch selbst als SO instanziiert werden, die Klasse `MCO_Universe` ist dagegen „abstract“, so dass nur ihre Unterklassen für SOs verfügbar sind. Wie in den Kapiteln 7.3 und 8.5 erläutert wurde, ändern SOs des Typs `MCO_Dynamic` eher ihren eigenen Zustand, während SOs des Typs `MCO_Universe` im Allgemeinen global auf viele SOs in der Szene wirken. MaxControl führt seine Simulation nur für SOs dieser beiden Klassen (oder Unterklassen davon) aus. Dies ermöglicht die Existenz weiterer OTs, die erst in anderen geplanten Betriebsarten von MaxControl zum Einsatz kommen sollen und beim normalen Simulationsvorgang nicht berücksichtigt werden, auch wenn SOs dieser Klassen in der Szene vorhanden sind. So soll durch SOs des Typs `MCO_Rendering` eine automatische Steuerung komplexer Rendering-Vorgänge wie die Erzeugung mehrerer Filmsequenz-Dateien aus unterschiedlichen Kameraperspektiven möglich werden. SOs des Typs `MCO_Generating` sollen hingegen in einem rekursiven Ansatz automatisch komplexe Szenen erzeugen können. Diese beiden Klassen haben noch keine Unterklassen und sind selbst als „abstract“ definiert, so dass sie selbst noch nicht als SOs instanziiert werden können. Auf die mit beiden Klassen geplanten Erweiterungen von MaxControl wird in Kapitel 10.5 näher eingegangen. Im Folgenden wird angenommen, dass die OTs aller vorkommenden SOs jeweils entweder direkt die Klasse `MCO_Dynamic` oder Unterklassen von `MCO_Dynamic` oder von `MCO_Universe` sind und somit auch simuliert werden.

Vor der Simulation wird die gegebene 3D-Szene analysiert und für alle 3D-Objekte werden entsprechende SOs in MaxControl erzeugt. Diese SOs werden in einer festen Reihenfolge in einer Liste der Klasse `Vector` gespeichert (siehe Kap. 8.1.3, „`getScene()`“). Diese Liste wird im Folgenden auch als Objektliste bezeichnet. Die Reihenfolge der SOs in dieser Liste wird im Laufe der Simulation nicht mehr verändert und entspricht der Reihenfolge, in der die SOs in einem Simulationszwischen-schritt simuliert werden, indem ihre Verhaltensweisen ausgeführt werden.

Es folgt ein Auszug aus dem **Szenen-Analysealgorithmus**, der die Szenenstruktur aus 3D-Studio-MAX an MaxControl übermittelt. Die Reihenfolge, in der die 3D-Objekte dabei gefunden werden, orientiert sich an der Tiefensuche in Bäumen und legt die Reihenfolge fest, in der die zugehörigen SOs in die Objektliste eingetragen werden.

Die in Abbildung 47 folgende Deklaration der rekursiven Szenenanalysefunktion `MC_getNames` ist in der Skriptsprache MAXScript verfasst. Der Programmcode wird im Folgenden erläutert.

Als ersten Parameter `MC_objects` erhält die Funktion ein Array von strukturierten 3D-Objekten. Diese 3D-Objekte sollen in eine Objektliste aufgenommen werden, wo sie durch ihre zugehörigen SOs repräsentiert sind. Die Funktion durchsucht die 3D-Objekte aus dem übergebenen Array rekursiv nach direkten und indirekten Kindobjekten, die ebenfalls in die Objektliste übernommen werden sollen. Der zweite Parameter `MC_level` gibt die aktuelle Tiefe an, in der im Szenengraphen gesucht wird.

Der beim Start der Analyse eines gegebenen Szenengraphen ausgeführte Aufruf `MC_getNames rootNode.children 0` übergibt das Array der Kindobjekte (`children`) des Wurzelknotens (`rootNode`) als ersten Parameter. Die aktuelle Tiefe im

Szenengraphen ist dabei noch 0, deshalb wird dieser Wert als zweiter Parameter übergeben. Dem Wurzelknoten sind alle anderen 3D-Objekte der Szene direkt oder indirekt als Kindobjekte zugeordnet. Durch die folgenden rekursiven Aufrufe von MC_getNames werden so alle 3D-Objekte in der Szene erreicht.

```
function MC_getNames MC_objects MC_level=
(
  MC_currentObject=undefined
  MC_type=undefined
  MC_controllerType
  if MC_objects.count>0 then
  (
    ...
    for MC_i=1 to MC_objects.count do
    (
      MC_currentObject=MC_objects[MC_i]
      MC_type=getUserProp MC_currentObject "MC_type"
      ...
      MC_getNames MC_currentObject.children (MC_level+1)
    )
    ...
  )
)
```

Abbildung 47: Szenen-Analysealgorithmus

Die Schleife for MC_i=1 to MC_objects.count do (...) durchläuft die 3D-Objekte im übergebenen Array rootNode.children, die jeweils der Variablen MC_currentObject zugewiesen werden. Durch verschiedene Anweisungen wie MC_type=getUserProp MC_currentObject "MC_type" werden Daten aus dem jeweiligen 3D-Objekt ausgelesen und an MaxControl übermittelt, wie hier z.B. der ihm zugewiesene Objekttyp. MaxControl erzeugt dann ein entsprechendes SO für dieses 3D-Objekt und trägt es als nächstes Objekt in die Objektliste ein. Neben dem zugeordneten Objekttyp werden weitere Daten aus den 3D-Objekten gelesen und an die SOs übertragen, wie der Name, den das 3D-Objekt in der Szene hat, ggf. eine Referenz auf sein evtl. vorhandenes Elternobjekt sowie die Liste seiner Kindobjekte, wobei hier jeweils Referenzen auf die entsprechenden SOs verwendet werden. Abschließend wird durch den Aufruf MC_getNames MC_currentObject.children (MC_level+1) die Rekursion mit der Liste der Kinder des betrachteten 3D-Objekts weitergeführt.

Die SOs von Kindobjekten aus der Szene werden also nach dem SO ihres jeweiligen Elternobjekts in die Objektliste eingetragen.

Die Reihenfolge, in der Geschwisterobjekte aus dem jeweils übergebenen Array MC_objects in die Objektliste eingetragen werden, hängt von ihrer Reihenfolge in diesem Array ab. Diese Reihenfolge hängt technisch von dem verwendeten 3D-Animationsprogramm ab, das die Arrays zur Verfügung stellt. Bei 3D-Studio-MAX bestimmt die Reihenfolge, in der die 3D-Objekte erstellt wurden, auch die Reihenfolge, in der Geschwisterobjekte in einem „children“-Array sowohl von rootNode als auch von anderen 3D-Objekten gespeichert sind.

Abbildung 48 erläutert das Entstehen der Objektliste zu einer Szene:

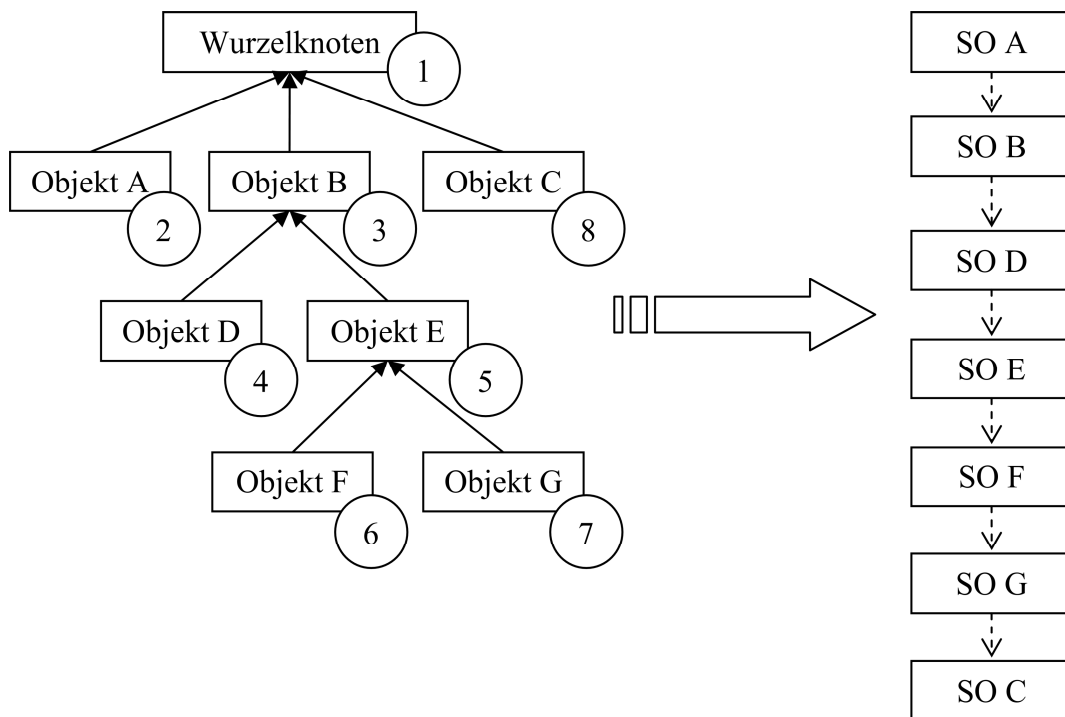


Abbildung 48: Herstellung der Objektliste

Im obigen Strukturdiagramm des Szenengraphen zeigen die Pfeile jeweils von einem Kindobjekt auf sein Elternobjekt. Bei Geschwisterobjekten ist die Reihenfolge von links nach rechts jeweils die Reihenfolge, in der die 3D-Objekte im 3D-Animationsprogramm gespeichert sind. Jedem 3D-Objekt „Objekt X“ aus dem Strukturdiagramm ist in der Objektliste ein entsprechendes SO mit der Bezeichnung „SO X“ zugeordnet. Die Reihenfolge, in der die 3D-Objekte in die Objektliste eingetragen werden, ist im Strukturdiagramm in kreisförmig umrahmten Nummern angegeben.

Diese Analysefunktion wird nicht nur als Vorbereitung für die Simulation verwendet, sondern in der technischen Umsetzung auch für einen besonderen Schritt, der einmal vor der ersten Simulation einer gegebenen Szene durchgeführt werden muss. Auch in diesem Schritt wird zunächst die Szene wie oben angegeben analysiert. Allen 3D-Objekten im 3D-Animationsprogramm werden dann gemäß ihrem Objekttyp die nicht-standard SPs und alle MPs hinzugefügt. Außerdem werden Repräsentationen dieser nicht-standard SPs und MPs in der grafischen Benutzeroberfläche des 3D-Animationsprogramms erzeugt.

Nach strukturellen Änderungen an der Szene oder an relevanten OTs muss dieser Schritt erneut durchgeführt werden. Er wird technisch genauer in Kap. 10.2.1 erläutert.

Bei der hier zugrunde liegenden vereinfachten Konzeption wird jedoch angenommen, dass dieser Schritt automatisch für jedes 3D-Objekt direkt bei der Zuordnung eines OTs und bei jeder Änderung an seinem OT vorgenommen wird.

Wir kommen zur **Definition der Simulationsfunktion „sim“** für die Zustandsübergänge $\sigma_{n-1,j-1}$ nach $\sigma_{n-1,j}$, $j=1,\dots,m(n)$, $n \in [a+1,b]$: Gegeben seien eine 3D-Szene und ihre Ergänzung durch Objekttypen und daraus abgeleitete Simulationsobjekte. Sei $\Delta t(n)$ das Zeitintervall, das im Simulationsschritt zu Frame n jedem Simulationszwischen Schritt zugeordnet wird. Der Zwischenschritt von $\sigma_{n-1,j-1}$ nach $\sigma_{n-1,j}$ ist dadurch festgelegt, dass alle Simulationsobjekte in

der Reihenfolge der Objektliste einmal durchlaufen werden und die zugehörige update-Methode sowie damit auch die update-Methoden aller zugehörigen VHs in einer festen Reihenfolge aufgerufen und für $\Delta t(n)$ ausgeführt werden. Die Reihenfolge, in der die VHs eines SOs aufgerufen werden, ist festgelegt durch:

- die Anordnung der VHs in der jeweiligen SO-Hierarchie
- die Deklarationsreihenfolge von Geschwister-VHs untereinander, d.h. von direkten Unter-VHs einer VH
- die Implementierungen der Methoden `updateBeforeSubBehaviours` und `updateAfterSubBehaviours` einer VH, weil diese festlegen, welche Anweisungen die VH vor bzw. nach ihren Unter-VHs ausführt.

Für die obige Beispielszene bedeutet dies, dass zuerst alle VHs von „SO A“ ausgeführt werden, dann alle VHs von „SO B“, dann die von „SO D“, „SO E“, „SO F“, „SO G“ und schließlich alle VHs von „SO C“. Wie oben erläutert wurde, sind die SOs von Elternobjekten vor den SOs ihrer Kindobjekte in der Objektliste eingetragen, so dass in jedem Simulationszwischen-schritt stets die VHs von Elternobjekten vor den VHs ihrer Kindobjekte ausgeführt werden.

Wir gehen genauer auf die Reihenfolge ein, in der die update-Methoden eines SOs und seiner VHs aufgerufen werden. Außerdem werden die Startup-Methoden diskutiert.

Wir beginnen mit dem Quellcode der update-Methode aus `MC_Object`, die in Kapitel 8.4.2 erläutert wurde:

```
public void update(long f)
{
    if (behaviours.size() > 0)
        getBehaviour("mainBehaviour").update(f);
}
```

Als aktueller Parameter wird dieser Methode die Nummer *n* des zu simulierenden Frames übergeben.

Jedes SO führt also zunächst die update-Methode seiner Haupt-VH aus, die dann selbst mehrere Unter-VHs enthalten darf. Die Bedingung `behaviours.size() > 0` verhindert, dass ein Zugriff auf eine nicht vorhandene Haupt-VH versucht wird.

Jede Verhaltensweise führt mit ihrer update-Methode nicht nur die für sie spezifizierten Zustandsänderungen durch, sondern sie ruft rekursiv auch die update-Methoden ihrer direkten und indirekten Unter-VHs auf. Eine VH darf mehrere Unter-VHs besitzen. Diese werden in der Reihenfolge ausgeführt, in der sie in der Liste `behaviours` ihrer Besitzer-VH stehen. Diese Reihenfolge entspricht wiederum der Reihenfolge, in welcher in der Besitzer-VH die Container-Felder deklariert wurden, welche diese Unter-VHs referenzieren. Wie in Kapitel 8.6.2 ab Seite 149 bereits erwähnt wurde, bestimmt damit die Deklarationsreihenfolge der Container-Felder in einem Besitzer auch die Ausführungsreihenfolge der zugehörigen VHs.

Zusätzlich haben die VHs wie in Kapitel 8.6.3 beschrieben die Möglichkeit, sowohl vor als auch nach ihren Unter-VHs Anweisungen auszuführen.

Zur Erläuterung folgt hier der Code der update-Methode aus der Klasse MC_Behaviour:

```
public void update(long f)
{
    if (getBooleanValue("behaviourOn"))
    {
        updatingBeforeSubBehaviours=true;
        updateBeforeSubBehaviours(f);
        for (int i=0; i<behaviours.size(); i++)
        {
            behaviours.elementAt(i).update(f);
        }
        updatingBeforeSubBehaviours=false;
        updateAfterSubBehaviours(f);
    }
}
```

Diese Methode werde nun in einer VH „A“ ausgeführt. Die Bedingung „if (getBooleanValue("behaviourOn"))“ stellt zunächst sicher, dass die VH „A“ und auch ihre Unter-VHs nur dann ausgeführt werden, wenn „A“ als eingeschaltet gilt (siehe Beschreibung des Feldes behaviourOn in Kap. 8.6.1). Ist „A“ eingeschaltet, so wird zunächst ihre Methode updateBeforeSubBehaviours aufgerufen.

Danach werden die update-Methoden der direkten Unter-VHs von „A“ ausgeführt. Dies führt zu einer Rekursion, da auch deren update-Methoden analog mit ihren entsprechenden Unter-VHs verfahren. Die Reihenfolge, in der die update-Methoden von direkten Unter-VHs aufgerufen werden, entspricht dabei wieder der Deklarationsreihenfolge der Containerfelder, welche diese Unter-VHs in ihrem Besitzer referenzieren.

Nachdem die eingeschalteten Unter-VHs von „A“ ausgeführt wurden, wird in „A“ die Methode updateAfterSubBehaviours aufgerufen, die Zustandsänderungen nach der Ausführung der Unter-VHs durchführen kann.

Die SO-Hierarchie wird also rekursiv durchlaufen, wobei jede VH vor und nach ihren Unter-VHs Anweisungen ausführen kann. Dieser Vorgang sei am Beispiel der OT-Hierarchie zu „OT_{Robot}“ aus Kapitel 7.2.1 erläutert. Dabei wird angenommen, dass alle VHs eingeschaltet sind.

In der folgenden Abbildung 49 sind die VHs so mit Symbolen (m \blacktriangleright), (\blacktriangleright n) versehen, dass die Nummern m, n die Ausführungsreihenfolge der Methoden angeben. Für eine VH „A“ bedeuten (m \blacktriangleright) und (\blacktriangleright n), dass ihre Methode updateBeforeSubBehaviours die Nummer m und ihre Methode updateAfterSubBehaviours die Nummer n erhalten. Dazwischen liegen die Nummern der Methodenaufrufe der Unter-VHs von „A“.

Analog dazu werden vor der Ausführung des ersten Simulationsschrittes die startup-Methoden aller SOs und ihrer VHs ausgeführt. Dabei wird die startup-Methode einer VH „A“ immer **vor** den startup-Methoden ihrer Unter-VHs ausgeführt.

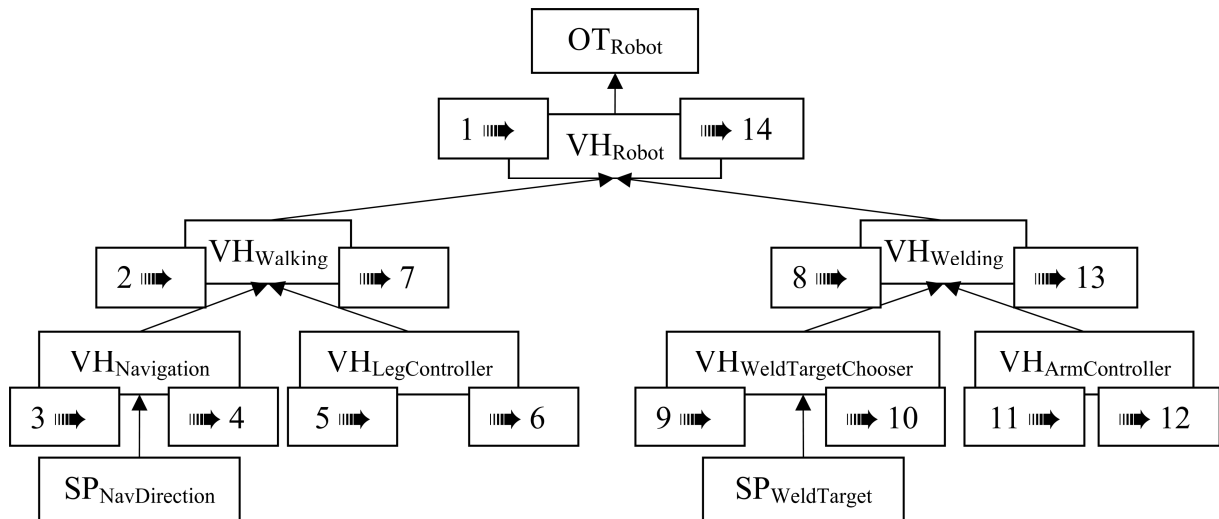


Abbildung 49: Ausführungsreihenfolge der VHs in der OT-Hierarchie zu „OT_{Robot}“

Zur Erläuterung folgt hier der Quellcode der `startup`-Methode aus `MC_Object`, die in Kapitel 8.4.2 eingeführt wurde und in ihrem Aufbau der Methode `update` aus `MC_Object` gleicht:

```

public void startup(long f)
{
    if (behaviours.size() > 0) getBehaviour("mainBehaviour").startupAll(f);
}

```

Da die `startup`-Methoden aller VHs nur vor dem ersten Simulationsschritt ausgeführt werden, wird der `startup`-Methode wie bereits in Kapitel 8.4.2 erläutert die Nummer `a+1` als aktueller Parameter übergeben.

Jede Verhaltensweise führt mit ihrer `startupAll`-Methode nicht nur ihre eigene `startup`-Methode durch, sondern sie ruft rekursiv auch die `startupAll`-Methoden ihrer direkten und indirekten Unter-VHs auf. Diese Methoden werden wieder in der Reihenfolge ausgeführt, in der die zugehörigen VHs in der Liste `behaviours` stehen.

Es folgt hier der Code der `startupAll`-Methode aus der Klasse `MC_Behaviour`:

```

public void startupAll(long f)
{
    startup(f);
    for (int i=0; i<behaviours.size(); i++)
    {
        behaviours.elementAt(i).startupAll(f);
    }
}

```

Die SP „behaviourOn“ wird hier im Gegensatz zur Vorgehensweise in der Methode `update` nicht überprüft, die `startup`-Methoden werden also auf jeden Fall in allen VHs ausgeführt, auch wenn diese ausgeschaltet sind. Dies berücksichtigt den Fall, dass eine zunächst ausgeschaltete VH zu einem späteren Zeitpunkt der Simulation entweder durch manuelle Festlegung oder durch andere VHs doch noch eingeschaltet wird. Diese VH muss dann bereits ihre in der `startup`-Methode definierten „Vorbereitungen“ getroffen haben, um korrekt arbeiten zu können. Dabei ist zu beachten, dass die `startup`-Methoden daher

wirklich nur Vorbereitungen treffen dürfen, die nicht den Zustand der Szene verändern. Die in Kapitel 8.4.1 bei der Erläuterung des Feldes `sceneName` erwähnte VH `MCB_CarLightController` stellt in ihrer `startup`-Methode beispielsweise die Liste der von ihr zu steuernden Lichtquellen zusammen und teilt den Lichtquellen dort auch ihre Aufgaben wie Blinker oder Bremslicht zu. Dabei werden nur intern in der VH gespeicherte Listen der Klasse `Vector` erstellt, der Zustand der SOs wird dabei nicht verändert.

Beim Ausführen der `startupAll`-Methode wird die SO-Hierarchie wie bei der `update`-Methode aus `MC_Behaviour` rekursiv durchlaufen, wobei jede VH vor ihren Unter-VHs Vorbereitungen treffen kann. Dieser Vorgang sei wieder am Beispiel der OT-Hierarchie zu „OT_{Robot}“ aus Kapitel 7.2.1 erläutert:

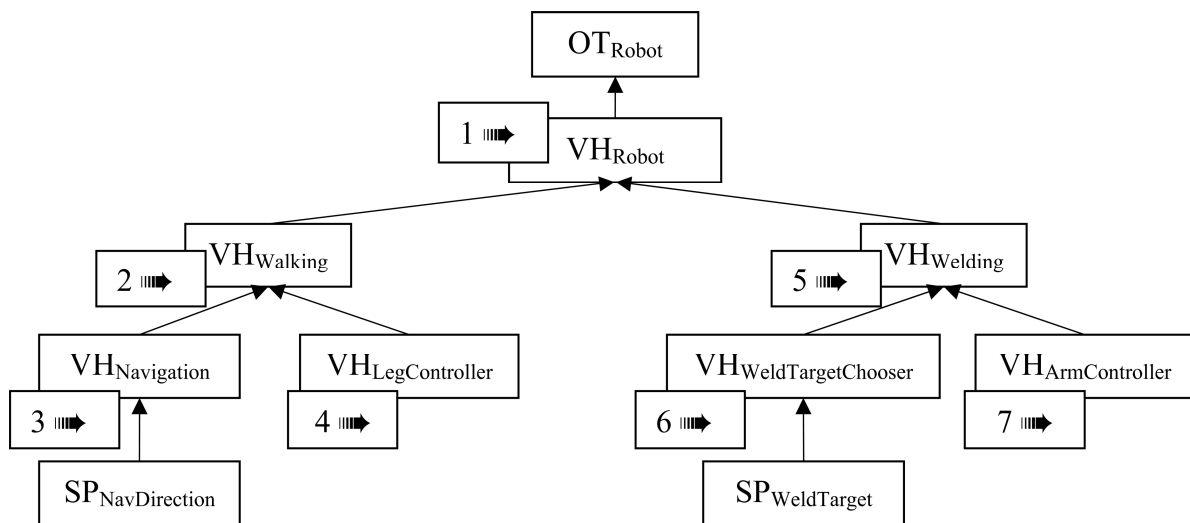


Abbildung 50: Ausführungsreihenfolge der `startup`-Methoden in „OT_{Robot}“

Im obigen Diagramm sind die VHs so mit Symbolen (m \dashrightarrow) versehen, dass die Nummern m die Ausführungsreihenfolge der `startup`-Methoden der VHs angeben. Dabei bedeutet (m \dashrightarrow), dass an dieser Stelle m die Methode `startup` der zugehörigen VH ausgeführt wird, auf welche die Ausführung der `startup`-Methoden ihrer Unter-VHs folgt.

9.3 Simulationszyklus

Auf der Basis des vorangehenden Kapitels lässt sich nun der Simulationszyklus als Flussdiagramm angeben (Abbildung 51). Dieses Flussdiagramm stellt zunächst auf einer hohen Abstraktionsstufe Operationen und deren Zusammenhänge im Programmfluss dar. Die Operationen werden im Anschluss an das Diagramm genauer erläutert. Die gestrichelt umrahmt dargestellten Anweisungen werden erst später in Kapitel 10.2.2 erläutert. Diese Abstufung entspricht weitgehend dem tatsächlichen Programmaufbau.

Die Darstellungen von Variablen, Bedingungen und Anweisungen in den Flussdiagrammen orientieren sich an Java-Syntax, wobei für Variable nur ein grober Typ angegeben wird. So wird beispielsweise bei Integer-Variablen keine Unterscheidung zwischen `integer` und `long` vorgenommen, bei Float-Variablen wird analog nicht zwischen `float` und `double` unterschieden. Die spezielle Anweisung `RETURN` steht für den Rücksprung aus einer Methode in den aufrufenden Teil des Programmcodes.

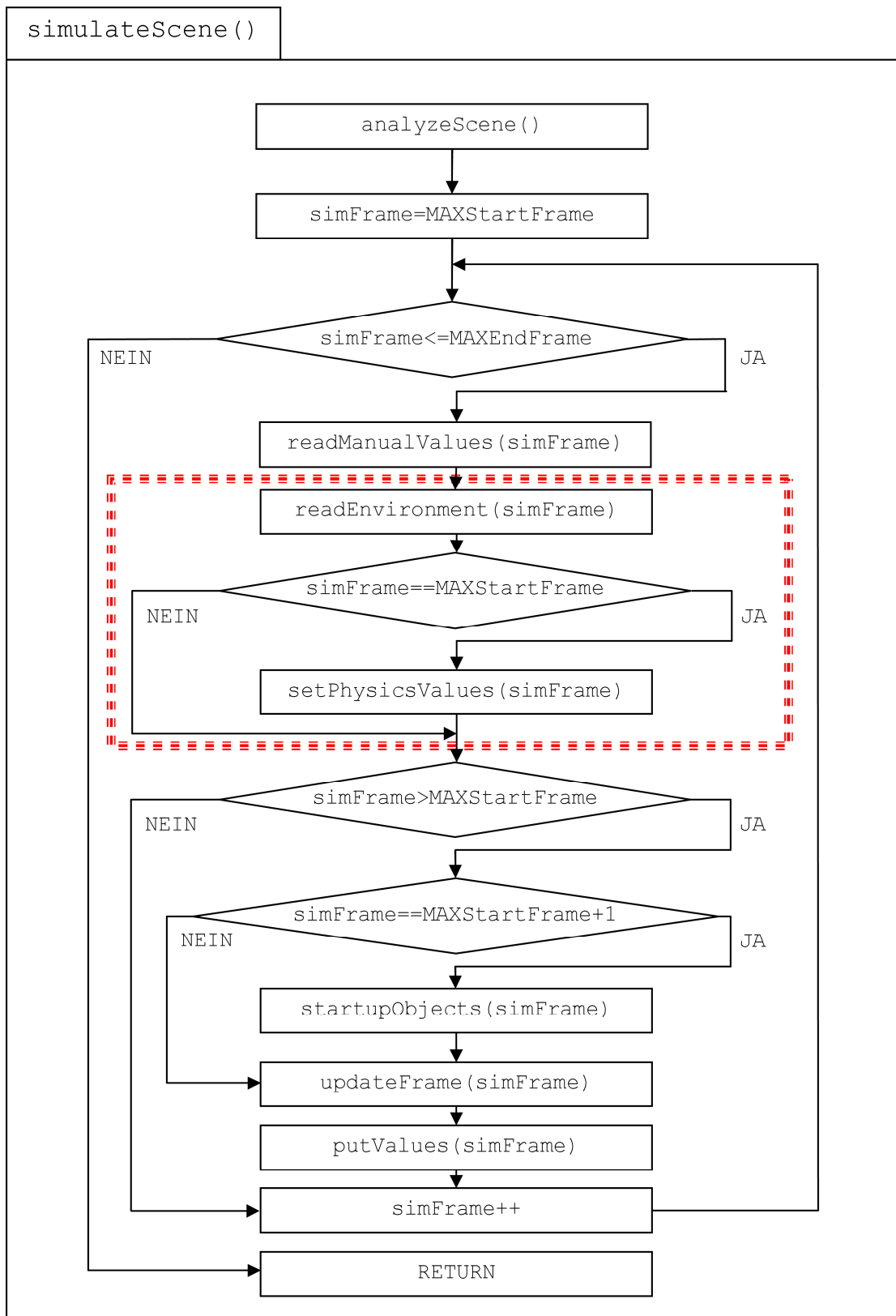


Abbildung 51: Simulationszyklus ohne genaue Kommunikation

analyzeScene() :

Analysiert wie im vorangehenden Kapitel 9.2 erläutert die Szenenstruktur und erzeugt SOs für die zugehörige Objektliste.

MAXstartFrame:

Nummer des ersten Frames im Simulationsintervall

simFrame:

Für $\text{simFrame} > \text{MAXstartFrame}$ ist simFrame die Nummer des aktuell zu simulierenden Frames.

Für $\text{simFrame} = \text{MAXstartFrame}$ ist simFrame die Nummer des Frames, der den Anfangszustand für die Simulation trägt.

MAXendFrame:

Nummer des letzten Frames im Simulationsintervall.

readManualValues(simFrame):

Liest die Werte der SPs, die im aktuellen Frame simFrame als manuell kontrolliert gelten, aus dem Szenenzustand simFrame aus und überträgt sie in den MaxControl-Zustand.

startupObjects(simFrame):

Ruft die startup-Methoden der SOs auf.

updateFrame(simFrame):

Führt den Simulationsschritt zu Frame simFrame durch und führt zu einem neuen MaxControl-Zustand

putValues(simFrame):

Überträgt die Werte der SPs aus dem neuen MaxControl-Zustand in den Szenenzustand simFrame .

Zuerst wird mit `analyzeScene` die in Kapitel 9.2 erläuterte Szenenanalyse durchgeführt.

Der Befehl `readManualValues` überträgt für jeden Frame n des Simulationsintervalls mit $\text{MAXstartFrame} < n \leq \text{MAXendFrame}$ die manuell kontrollierten Werte aus dem Szenenzustand δ_n in den MaxControl-Zustand σ_{n-1} , was diesen zu $\sigma_{n-1,0}$ abändert.

Im ersten Frame MAXstartFrame des Simulationsintervalls werden alle SPs als manuell kontrolliert behandelt, unabhängig vom tatsächlichen Wert ihrer jeweiligen MPs im Anfangszustand $\delta_{\text{MAXstartFrame}}$. Damit werden durch `readManualValues` die Werte aller SPs aus diesem Szenenzustand zu Beginn der Simulation gemäß der Vorschrift

$\sigma_{\text{MAXstartFrame}} =_{df} \delta_{\text{MAXstartFrame}}|_S$ (siehe Definition 2 aus Kapitel 6.5) in einen MaxControl-

Zustand übertragen, der als Anfangszustand $\sigma_{\text{MAXstartFrame}}$ für die Simulation gewählt wird. Die Bedingung $\text{simFrame} > \text{MAXstartFrame}$ stellt sicher, dass auf diesen Lesevorgang aus $\delta_{\text{MAXstartFrame}}$ weder ein Simulationsschritt zu Frame MAXstartFrame noch ein Aufruf von `putValues(MAXstartFrame)` folgen. Stattdessen wird simFrame um den Wert 1 inkrementiert und die eigentliche Simulation begonnen.

Der Simulationsschritt zu Frame $\text{MAXstartFrame}+1$ wird dann die Werte der in Frame $\text{MAXstartFrame}+1$ nicht manuell kontrollierten SPs aus $\sigma_{\text{MAXstartFrame}}$ übernehmen und die Werte der in Frame $\text{MAXstartFrame}+1$ manuell kontrollierten SPs aus $\delta_{\text{MAXstartFrame}+1}$ lesen. Für $\text{simFrame} == \text{MAXstartFrame}+1$ wird vor der Durchführung

des ersten Simulationsschrittes die Methode `startupObjects(simFrame)` aufgerufen, und damit indirekt auch jeweils die `startup`-Methode in allen SOs.

Für jeden auf `MAXStartFrame` folgenden Frame `simFrame > MAXStartFrame` des Simulationsintervalls wird mit `updateFrame(simFrame)` der zugehörige Simulationsschritt durchgeführt. Danach ist der Aufruf von `putValues(simFrame)` erforderlich, um die neuen Werte der SPs in den Szenenzustand `n` zu übertragen.

Wurde der Simulationsschritt zum letzten Frame `MAXEndFrame` durchgeführt, so wird die Simulation beendet und das Ergebnis befindet sich in der Szenenzustandsfolge. Die so automatisch entstandene Animation kann mit dem 3D-Animationsprogramm nun betrachtet, nachbearbeitet, abgespielt und auch gerendert werden.

Die folgende Abbildung 52 erläutert die Funktionsweise der zu Beginn der Simulation aufgerufenen Methode `startupObjects(simFrame)`, welche die `startup`-Methoden aller SOs in der Reihenfolge der Objektliste aufruft und damit rekursiv die `startup`-Methoden aller zugehörigen VHs (siehe Kap. 9.2):

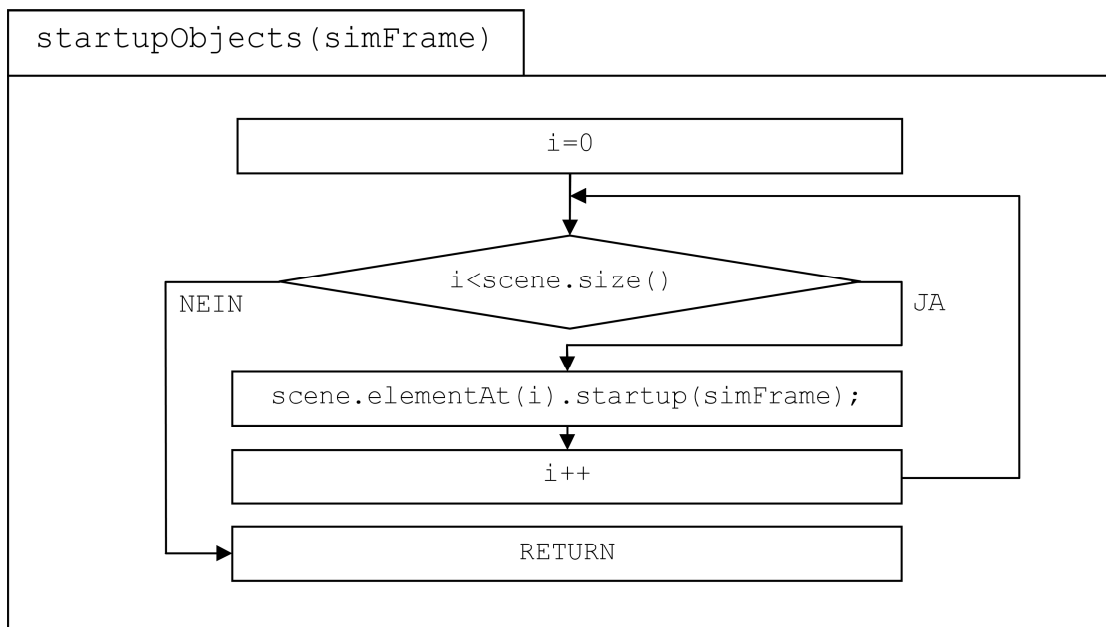


Abbildung 52: Ablaufdiagramm zu `StartupObjects`

`scene`:

Dieses Java-Objekt der Klasse `Vector` trägt die Objektliste. Die Indizierung der Simulationsobjekte in der Objektliste beginnt bei 0.

`i`:

Indexvariable für den Zugriff auf die Simulationsobjekte in der Objektliste.

`scene.size()`:

Liefert die Anzahl der Objekte in der Objektliste.

`scene.elementAt(i).startup(simFrame)`:

Führt die `startup`-Methode des Simulationsobjektes mit dem Index `i` in der Objektliste durch.

Das Diagramm in der folgenden Abbildung 53 erläutert die Funktionsweise des Aufrufs `updateFrame(simFrame)`, der den Simulationsschritt zu Frame `simFrame` durchführt. Auch hier werden die gestrichelt umrahmt dargestellten Anweisungen erst später in Kapitel 10.2.2 erläutert:

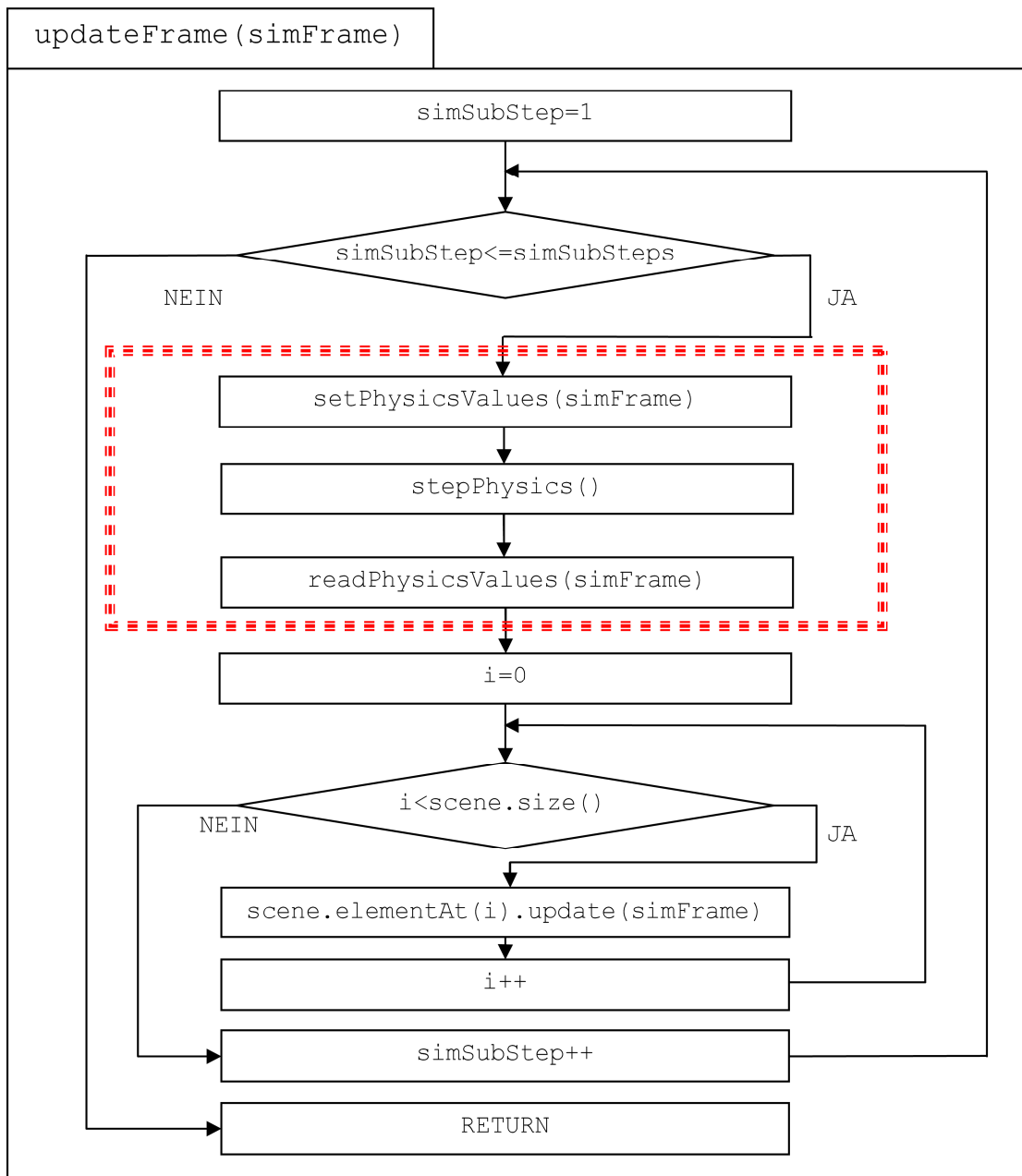


Abbildung 53: Ablaufdiagramm zu `updateFrame`

`simSubStep`:

Nummer des aktuellen Simulationszwischen schrittes

`simSubSteps`:

Anzahl der Zwischenschritte für den aktuellen Simulationsschritt (siehe Kapitel 10.2.2)

`scene, i, scene.size()`:

Siehe Erläuterungen zu Abbildung 52.

```
scene.elementAt(i).update(simFrame) :
```

Führt die update-Methode des Simulationsobjektes mit dem Index *i* in der Objektliste durch.

Die Methode `updateFrame(simFrame)` führt nacheinander die einzelnen Simulationszwischenschritte zu Frame *n* durch, deren aktuelle Anzahl in der Variablen `simSubSteps` zu finden ist. In jedem Simulationszwischenschritt wird für jedes SO seine update-Methode aufgerufen. Dabei wird die Reihenfolge aus der Objektliste eingehalten. Wie im vorangehenden Kapitel 9.2 erläutert wurde, führt dies zu einem rekursiven Aufruf der update-Methoden aller VHs der SOs.

9.4 Probleme und Besonderheiten der MaxControl-Simulation sowie daraus resultierende Design-Vorschriften

9.4.1 Zur Wiederholbarkeit von Simulationen

Wie in Kapitel 6.4 gesagt wurde, speichert MaxControl in der Regel genau einen Zustand, den so genannten MaxControl-Zustand. Wird in einem Simulationszwischenschritt die update-Methode einer VH „A“ ausgeführt, so arbeitet sie auf einem solchen MaxControl-Zustand „Z1“, d.h., sie liest die Werte von SPs aus diesem Zustand und ändert die Werte von SPs in diesem Zustand, wenn sie neue Werte schreibt. Dadurch wird der MaxControl-Zustand „Z1“ verändert und wird zu einem Zustand „Z2“. Eine in demselben Simulationszwischenschritt direkt nachfolgend ausgeführte VH „B“ arbeitet auf dem neuen Zustand „Z2“, der Zustand „Z1“ ist für die VH „B“ nicht mehr sichtbar. Damit ist das Ergebnis der Simulation im Allgemeinen von der Reihenfolge abhängig, in der die VHs nacheinander ausgeführt werden. Dieses Vorgehen ist, wie in den Kapiteln 6.5 und 9.1 erläutert, mit dem Gauß-Seidel-Prinzip vergleichbar.

Es wird hier von einer Simulation verlangt, dass sie unter bestimmten Voraussetzungen wiederholbar ist und damit dasselbe Ergebnis liefert wie eine vorangehende Simulation. „Wiederholbar“ bedeutet hier, dass eine Simulation auf einer Szenenzustandsfolge „A“ gestartet wird, die durch diese Simulation zu Szenenzustandsfolge „B“ wird. Startet man nun erneut eine Simulation auf Szenenzustandsfolge „B“, so soll sich unverändert wieder die Szenenzustandsfolge „B“ ergeben. Das ist möglich, weil der Anfangszustand und die manuell kontrollierten SP-Werte in „A“ festlegen, welches Ergebnis „B“ die Simulation auf „A“ gestartet liefern soll. In „A“ ist dieses Simulationsergebnis aber evtl. noch nicht in den SPs vorhanden, erst „B“ trägt dann wirklich das endgültige Simulationsergebnis. Da sich in „B“ gegenüber „A“ weder der Anfangszustand, noch die manuell kontrollierten SP-Werte geändert haben, sollte eine erneute Simulation auf „B“ nun diese Zustandsfolge „B“ nicht mehr verändern, da sie bereits das Endergebnis trägt, sofern die Simulation wie unten gefordert deterministisch verläuft. So kann gewährleistet werden, dass eine auf „A“ ausgeführte Simulation bei einer erneuten Simulation auf dem Ergebnis „B“ nicht vollkommen andere Ergebnisse liefert, wenn an „B“ vor der erneuten Simulation nur relativ kleine Änderungen vorgenommen wurden, die sich gezielt auf die Simulation auswirken sollen. Solche Änderungen können sinnvoll sein, wenn das Ergebnis „B“ noch nicht zufriedenstellend der Zielsetzungen der Animation entsprach und das Verhalten der Simulation durch die Änderung von SP-Werten angepasst werden soll. Sinnvoll sind zunächst kleine Änderungen an Werten des Anfangszustands, an Werteverläufen von SPs in manuell kontrollierten Bereichen oder an Werteverläufen von Nicht-SPs. Dadurch soll es möglich werden, eine Simulation nur an gewünschten Stellen gezielt zu beeinflussen, ohne dass unerwünschte Seiteneffekte auf andere Stellen des Simulationsergebnisses entstehen. Die von den Änderungen semantisch unabhängigen Simulationsergebnisse sollen sich bei der erneuten

Simulation also nicht ändern. Es muss also gewährleistet sein, dass zwei Zustandsfolgen „A“ und „B“, die sich nach den unten folgenden Anforderungen gleichen, auch jeweils das gleiche Simulationsergebnis liefern, damit nur semantisch relevante Änderungen an „B“ gezielte Abweichungen des Simulationsergebnisses vom Ergebnis der Simulation auf „A“ erzeugen.

Im Folgenden sind notwendige Voraussetzungen dafür aufgelistet, dass aus zwei vorläufigen Szenenzustandsfolgen „A“ und „B“ durch Simulation die gleiche endgültige Szenenzustandsfolge entsteht:

Zunächst müssen beide vorläufigen Szenenzustandsfolgen den gleichen Anfangszustand und das gleiche Simulationsintervall haben. Durch die Gleichheit der Anfangszustände stimmen „A“ und „B“ auch bezüglich der zugrundeliegenden Szenestruktur überein. Zusätzlich müssen beide Szenenzustandsfolgen in den Nicht-SPs übereinstimmen und für jede SP in den Intervallen aus dem Simulationsintervall, in denen diese SP manuell kontrolliert ist, den gleichen Werteverlauf haben. Für die Keydarstellung sei angemerkt, dass die Folgen von Keys, welche die Werteverläufe von SPs speichern, verschieden sein dürfen, sofern sie den gleichen Werteverlauf repräsentieren.

Da MaxControl wie oben erläutert nach dem Gauß-Seidel-Prinzip vorgeht, ist es für eine Wiederholbarkeit einer Simulation ebenfalls erforderlich, dass die VHs bei beiden Szenenzustandsfolgen in entsprechenden Simulationszwischenritten jeweils in der gleichen Reihenfolge ausgeführt werden. Dafür reicht es nicht aus, dass die Szenengraphen beider vorläufigen Szenenzustandsfolgen „A“ und „B“ übereinstimmen, auch die Objektlisten müssen gleich sein. Besondere Aufmerksamkeit benötigen die Reihenfolgen von SOs, deren zugehörige 3D-Objekte im Szenengraphen „Geschwister“ sind, die also denselben Elternknoten haben. In welcher Reihenfolge diese SOs bei der Simulation behandelt werden, hängt von der jeweiligen Reihenfolge der Speicherung der zugehörigen 3D-Objekte im verwendeten 3D-Animationsprogramm ab, die in „A“ und „B“ jeweils unterschiedlich sein kann und sich in der jeweiligen Objektliste widerspiegelt.

Wird wie oben erläutert zur Anpassung des Simulationsergebnisses eine Änderung an Szene „B“ vorgenommen, so kann neben Änderungen an SP-Werten auch an das Hinzufügen oder Löschen von 3D-Objekten oder ein streng begrenztes lokales Ändern der der Struktur des Szenengraphen gedacht werden. Allerdings muss man dabei die Auswirkungen auf die Speicherungsreihenfolge der 3D-Objekte und damit die Auswirkungen auf die Objektliste genau kennen.

Um die Wiederholbarkeit einer Simulation unter den oben genannten Voraussetzungen zu gewährleisten, muss angenommen werden, dass sich die Reihenfolge, in der Geschwisterknoten gespeichert werden, nur durch Änderungen am Szenengraphen, wie etwa durch das Hinzufügen oder Löschen von 3D-Objekten, ändern kann, nicht aber durch Änderungen von SP-Werten. Das verwendete 3D-Animationsprogramm 3D-Studio-MAX zeigte bisher dieses Verhalten.

9.4.2 Aus den Anforderungen resultierende Designvorschriften

Um die Wiederholbarkeit zu sichern, müssen bei Verwendung von Zufallszahlen und bei Nutzung externer Datenquellen folgende Überlegungen und Regeln beachtet werden.

Zufallszahlen:

Zufallszahlen werden in den meisten Programmiersprachen und Anwendungen auf der Basis einer Funktion berechnet, die nur scheinbar zufällig durch Iteration dieser Funktion eine

deterministische Folge von Zahlen erzeugt. Wenn man eine bestimmte Zahl als Startwert angibt, wird darauf folgend eine feste Folge von Zufallszahlen erzeugt, die allein durch den Startwert bestimmt wird.

Wenn dieser Startwert als SP in den Anfangszustand der Simulation übernommen wird, so kann eine Simulation auch bei der Verwendung solcher Zufallswerte im Sinne von Kapitel 9.4.1 wiederholbar sein.

Verfahren, die eine Wiederholbarkeit der Anwendung von Zufallswerten verhindern, sollten vermieden werden. Dazu gehört beispielsweise, den Startwert der Zufallszahlenfolge anhand der aktuellen Uhrzeit zu bestimmen. Dadurch würde eine später ausgeführte Simulation eine andere Folge von Zufallswerten verwenden als eine früher ausgeführte Simulation.

Externe Datenquellen:

Sollte für die Simulation z.B. auf externe Datenbanken, Dateien oder das Internet zugegriffen werden, muss für die Wiederholbarkeit der Simulation sichergestellt sein, dass diese Zugriffe bei jedem Simulationsdurchlauf dieselben Daten in derselben Reihenfolge liefern. Zu relationalen Datenbanken siehe [KaKI93].

Der Zustand einer Datenbank, der Inhalt einer Datei bzw. die Daten aus dem Internet, auf die zugegriffen wird, sollten sich während der verschiedenen Simulationsdurchläufe also nicht unabhängig von der Simulation ändern, damit die Simulation wiederholbar bleibt.

9.4.3 Wirkung der Ausführungsreihenfolge der VHs auf das Simulationsergebnis

Die VHs kommunizieren in der Regel über ihre SPs miteinander. Sie reagieren aufeinander, indem sie Zustandsänderungen beobachten, die jeweils andere VHs verursacht haben. Wann eine VH „A“ auf eine Zustandsänderung reagieren kann, die von einer VH „B“ durchgeführt wurde, hängt ebenfalls von der Ausführungsreihenfolge der VHs ab und ist somit nach Kapitel 9.2 auch von der Szenenhierarchie und der Reihenfolge abhängig, in der Geschwisterknoten des Szenengraphen in dem 3D-Animationsprogramm gespeichert sind.

Es sind zwei Fälle zu unterscheiden:

- 1.: Die VH „A“ wird in jedem Simulationszwischen Schritt **nach** der VH „B“ ausgeführt. In diesem Fall kann „A“ noch in demselben Simulationszwischen Schritt die Zustandsänderungen sehen, die „B“ durchgeführt hat, und darauf reagieren.
- 2.: Die VH „A“ wird in jedem Simulationszwischen Schritt **vor** der VH „B“ ausgeführt. In diesem Fall kann „A“ nicht mehr in demselben Simulationszwischen Schritt die Zustandsänderungen sehen, die „B“ durchgeführt hat. Sie kann diese Änderungen dann erst im nächsten Simulationsschritt sehen und darauf reagieren.

Der Unterschied zwischen diesen zwei Fällen kann in der resultierenden Animation sichtbar werden, da jeder Frame n der entstehenden Animation das Ergebnis des letzten Simulationszwischen Schrittes zu Frame n speichert. Es kann bei der Betrachtung des Frames n also erkennbar sein, wenn die VH „A“ im letzten Simulationszwischen Schritt nicht mehr auf die Zustandsänderungen reagieren konnte, die „B“ in diesem Simulationszwischen Schritt durchgeführt hat.

Es kann also eine sichtbare Verzögerung entstehen⁶³, was an folgendem Beispiel verdeutlicht werden soll: In einer Szene wurde ein einfaches 3D-Objekt „Box01“ erzeugt, dem der OT „MCD_SimpleMotion“ zugeordnet wurde, der in Kapitel 8.6.3 im Zusammenhang mit der Methode `updateBeforeSubBehaviours` erläutert wurde. Das 3D-Objekt „Box01“ wurde nun über seine SPs so eingestellt, dass es sich mit einer konstanten Geschwindigkeit in Richtung der X-Achse des Weltkoordinatensystems bewegt.

Es wurde zusätzlich der OT „MCD_SimpleFollower“ implementiert, dessen Instanzen die Bewegungen eines anderen wählbaren 3D-Objekts mitmachen, wobei sie einem bestimmten Abstand zu diesem 3D-Objekt einhalten. Die Haupt-VH „MCB_SimpleFollower“ enthält nur die VH „MCB_SimpleFollowerMover“, die das Bewegen zusammen mit dem gewählten anderen 3D-Objekt steuert und deren Quellcode in Abbildung 54 angegeben ist:

```
public class MCB_SimpleFollowerMover extends MC_Behaviour
{
    public final MCP_SceneObjectNA targetToFollow=
        new MCP_SceneObjectNA ("F-Target", this);
    public final MCP_Double xOffset=new MCP_Double ("X-Offset", this);
    public final MCP_Double yOffset=new MCP_Double ("Y-Offset", this);
    public final MCP_Double zOffset=new MCP_Double ("Z-Offset", this);

    public MCB_SimpleFollowerMover(MC_Owner _owner)
    {
        super(_owner);
    }

    public void updateBeforeSubBehaviours(long f)
    {
        MC_Object targetToFollow=getMC_ObjectValue("targetToFollow");
        double x=targetToFollow.getXValue("position");
        double y=targetToFollow.getYValue("position");
        double z=targetToFollow.getZValue("position");
        double xOffset=getDoubleValue("xOffset");
        double yOffset=getDoubleValue("yOffset");
        double zOffset=getDoubleValue("zOffset");

        x+=xOffset;
        y+=yOffset;
        z+=zOffset;

        setXValue("position",x);
        setYValue("position",y);
        setZValue("position",z);
    }
}
```

Abbildung 54: Quellcode zu MCB_SimpleFollowerMover

Die SP „targetToFollow“ speichert in einem SO „F“ eine Referenz auf das SO „T“, dem „F“ folgen soll. Das SO „T“ habe dabei wie das 3D-Objekt „Box01“ im Beispiel den OT „MCD_SimpleMotion“. Die SPs „xOffset“, „yOffset“ und „zOffset“ speichern einen Abstand, der im Weltkoordinatensystem stets zu „T“ eingehalten werden soll. In der Methode „updateBeforeSubBehaviours“ werden zunächst diese SPs in gleichnamige lokale Variable gelesen. Zusätzlich wird die aktuelle Position von „T“ in die lokalen

⁶³ Da auch Töne automatisch erzeugt werden (s. Kap. 5.3, 10.2.4), könnte sich auch eine akustische Verzögerung zeigen.

Variablen x , y und z gelesen. Nachdem die Variablen x , y und z jeweils um die entsprechende Komponente des einzuhaltenden Abstandes inkrementiert wurden, werden ihre Werte in die SP „position“ von „F“ geschrieben.

Genau hier kann das oben angesprochene Problem auftreten. Es hängt von der Reihenfolge ab, in der die VHs von „F“ und „T“ in jedem Simulationszwischen Schritt ausgeführt werden, wie „F“ auf die Position von „T“ reagiert. Wird „F“ vor „T“ simuliert, so wird „T“ seine Position im aktuellen Simulationszwischen Schritt noch ändern, so dass „F“ vorher in diesem Simulationszwischen Schritt auf eine inzwischen veraltete Position von „T“ reagiert hat. In diesem Fall wird man in jedem Frame der erzeugten Animation sehen, dass „F“ verzögert hinter „T“ her bewegt wird.

Zur Verdeutlichung wurden der Szene mit dem oben angegebenen Objekt „Box01“ mit dem OT „MCD_SimpleMotion“ noch drei weitere Objekte „Box02“, „Box03“ und „Box04“ hinzugefügt, denen jeweils der OT „MCD_SimpleFollower“ zugeordnet wurde. Sie wurden so eingestellt, dass sie zu dem 3D-Objekt, dem sie folgen sollen, jeweils einen Mittelpunktsabstand von -10 Einheiten in Richtung der Y-Achse des Weltkoordinatensystems einhalten. Die drei Objekte folgen sich in einer Kette, „Box02“ folgt „Box01“, „Box03“ folgt „Box02“ und „Box04“ folgt „Box03“. Alle vier Objekte haben keine Elternobjekte im Szenengraphen, sie gehören also direkt zum Wurzelknoten und sind somit Geschwister. Die Reihenfolge, in der die VHs dieser Objekte ausgeführt werden, hängt also von der Reihenfolge ab, in der sie im 3D-Animationsprogramm gespeichert werden (siehe zur Festlegung der Ausführungsreihenfolge Kapitel 9.2). Hier sei angenommen, dass das 3D-Animationsprogramm Geschwisterobjekte in der Reihenfolge speichert, in der sie erstellt wurden. Bei allen bisherigen Tests zeigte das in der technischen Umsetzung verwendete 3D-Animationsprogramm „3D-Studio-MAX“ dieses nahe liegende Verhalten. Die Objekte wurden in der Reihenfolge („Box01“, „Box02“, „Box03“, „Box04“) erstellt.

Die daraus resultierende Simulationsreihenfolge der SOs führt daher zu **keinem** Verzögerungseffekt. In jedem Simulationszwischen Schritt wird zuerst das SO zu „Box01“ simuliert. Seine VH „MCB_SimpleMotionMover“ bewegt das Objekt dabei um einige Einheiten weiter in Richtung der X-Achse. Danach wird das SO zu „Box02“ simuliert. Seine VH „MCB_SimpleFollowerMover“ bewegt „Box02“ nun entsprechend der **neuen** Position von „Box01“. Analog werden danach „Box03“ und „Box04“ bewegt. Diese Ausführungsreihenfolge bewirkt, dass jedes SO auf die Änderungen reagiert, die das von ihm beobachtete SO noch im aktuellen Simulationszwischen Schritt gemacht hat. Dies gilt auch für den jeweils letzten Simulationszwischen Schritt zu einem Simulationsschritt n , dessen Ergebnis danach in Frame n sichtbar ist. In jedem Frame $n > a$ ergibt sich folgendes Bild:

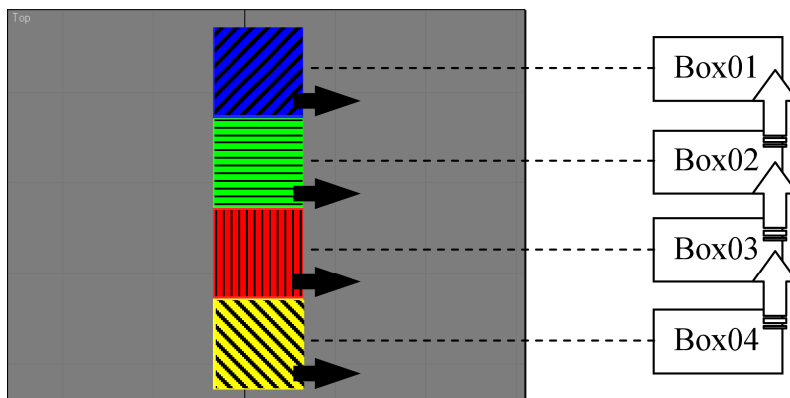


Abbildung 55: Synchroner Bewegung ohne Verzögerung



Die Pfeile der Form  geben in der obigen Darstellung an, welches Objekt A ein anderes Objekt B verfolgt. Der Pfeil zeigt von dem verfolgenden Objekt auf das verfolgte Objekt. Die horizontalen Pfeile im Bild geben die Bewegungsrichtung der Objekte an.

Kehrt man die Ausführungsreihenfolge nun um, so entsteht ein Verzögerungseffekt. Um dies zu demonstrieren, wurde aus der obigen Szene zunächst das Objekt „Box01“ gelöscht. Es wird hier davon ausgegangen, dass dies nicht die Reihenfolge der übrigen Objekte zueinander ändert, was z.B. in 3D-Studio-MAX bisher auch nicht zu beobachten war. Nun wird ein neues Objekt „Box05“ des Typs „MCD_SimpleMotion“ erzeugt, das sich genau so bewegt, wie vorher „Box01“. Die Abstände, welche die Objekte „Box02“ bis „Box04“ beim Verfolgen ihrer Zielobjekte einhalten, werden nun so eingestellt, dass sie sich in der obigen Zeichnung nicht mehr unter sondern über dem verfolgten Objekt anordnen. Auch die Verfolgungsordnung wurde umgekehrt. So folgt nun „Box04“ dem Objekt „Box05“, „Box03“ folgt „Box04“ und „Box02“ folgt „Box03“.

Da „Box05“ das letzte erzeugte 3D-Objekt ist, wird sein SO auch das letzte SO in der neuen Objektliste und somit in jedem Simulationszwischen schritt als letztes SO simuliert (siehe zur Erstellung der Objektliste Kapitel 9.2). Die Simulationsreihenfolge wird dadurch sehr ungünstig. In jedem Simulationszwischen schritt wird nun zuerst das SO zu „Box02“ simuliert. Es passt sich der letzten bekannten Position von „Box03“ an. Diese Position resultiert aus dem vorangehenden Zwischen schritt, im aktuellen Zwischen schritt wird „Box03“ seine Position noch ändern. Dies gilt analog für „Box03“ und „Box04“. Auch „Box04“ reagiert auf die letzte bekannte Position von „Box05“, die sich im aktuellen Zwischen schritt noch ändern wird.

Es ergibt sich in jedem Frame $n > a$ des Simulationsintervalls folgendes verändertes Bild:

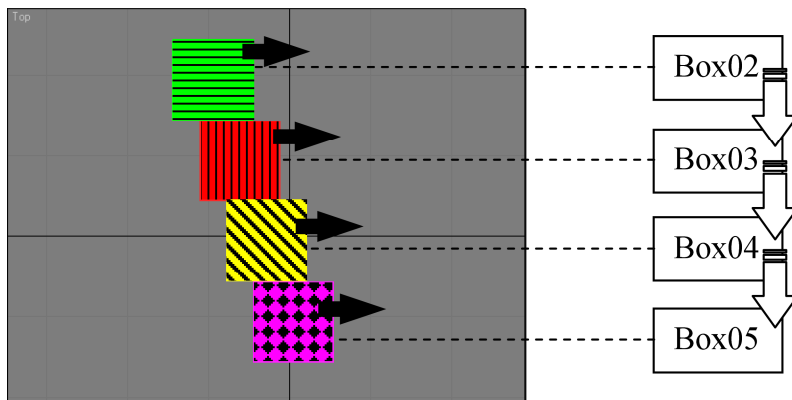


Abbildung 56: Synchroner Bewegung mit Verzögerung

Erhöht man die Anzahl an Simulationszwischen schritten, so lässt sich die Wirkung der Verzögerung zwar herabsetzen, aber nie vollkommen aufheben. Das Abschwächen der Verzögerung ergibt sich aus der kleineren Wegstrecke, die „Box05“ dann in jedem Simulationszwischen schritt zurücklegt (siehe Abbildung 57).

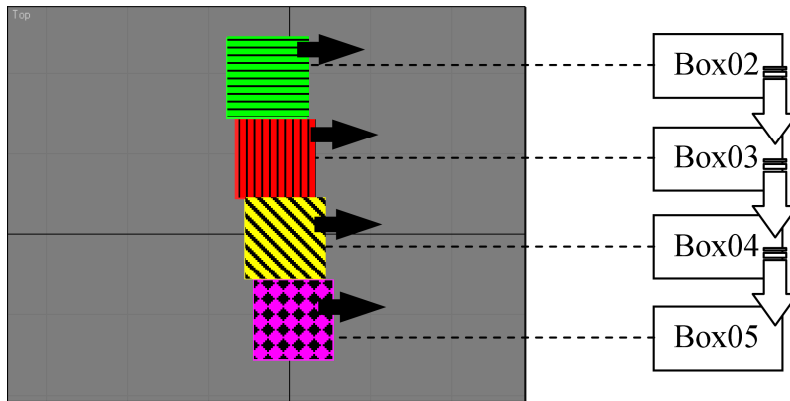


Abbildung 57: Synchrone Bewegung mit verringerter Verzögerung

9.4.4 Ausführungsreihenfolge der SOs aufgrund von Eltern-Kind-Beziehungen des Szenengraphen

Die Transformation eines Kindobjektes wird nach Kapitel 3.3 immer relativ zu der Transformation seines Elternobjektes gespeichert und interpretiert. Bewegt sich ein Elternobjekt, so bewegen sich deshalb im Weltkoordinatensystem seine Kinder automatisch mit, als wären sie an dem Elternobjekt befestigt, weil sie dabei ihre Position und Rotation im Koordinatensystem ihres Elternobjektes nicht ändern.

Will sich nun ein Kindobjekt selbständig zu einem bestimmten Punkt im Weltkoordinatensystem bewegen, so muss es dazu die Transformation seines Elternobjektes kennen und die entsprechende Position im Koordinatensystem des Elternobjektes berechnen, die es dort einnehmen muss, um im Weltkoordinatensystem an den gewünschten Punkt zu gelangen. Gleiches gilt für seine Ausrichtung (=Rotation) im Raum. Diese Berechnungen können von Methoden der Klasse „MC_Owner“ und „MCH_Transform3D“ übernommen werden (Siehe Kap. 10.1.2 und 10.1.3), so dass der Benutzer bei der Implementation von Verhaltensweisen in der Regel keine Algorithmen für solche Berechnungen erstellen muss.

Dennoch ergibt sich hier das Problem, dass ein Kindobjekt für solche Berechnungen die Transformation seines Elternobjektes kennen muss, die bis zum Ende des aktuellen Simulationsschrittes zu einem Frame n gültig bleibt, damit in Frame n keine unerwünschten Abweichungen erkennbar sind. Die folgende Darstellung soll dies verdeutlichen. Zur Vereinfachung wird das Problem hier nur in zwei Dimensionen betrachtet:

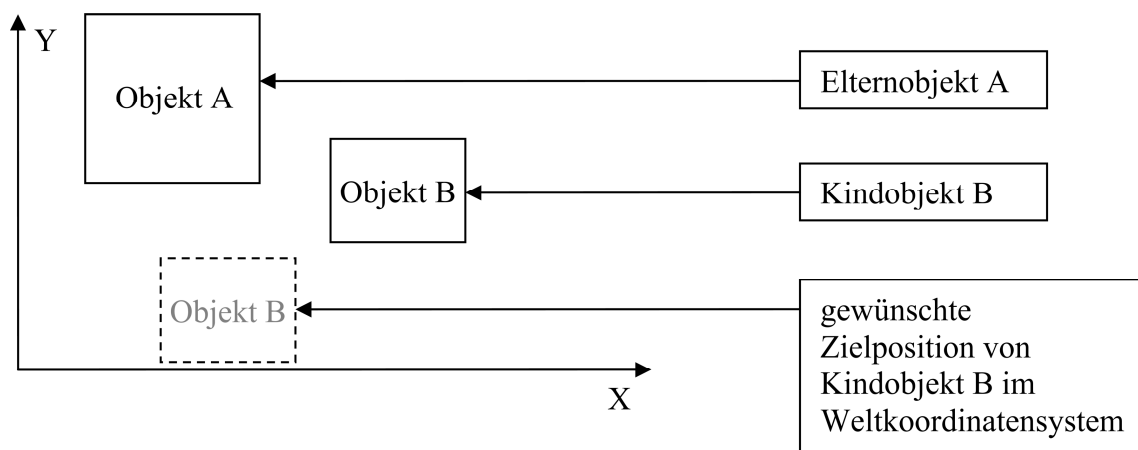


Abbildung 58: Positionscommunication zwischen Eltern- und Kindobjekten, Teil 1

Objekt B soll die angegebene Zielposition in einem Simulationszwischen schritt erreichen, und kann berechnen, welche Position es im Koordinatensystem seines Elternobjektes A einnehmen muss, um die gezeigte Zielposition im Weltkoordinatensystem zu erreichen.

Wenn aber nach dieser Positionsänderung des Objektes B das Elternobjekt A noch in demselben Simulationszwischen schritt z.B. seine Rotation ändert, so kann sich folgendes unerwünschte Bild ergeben:

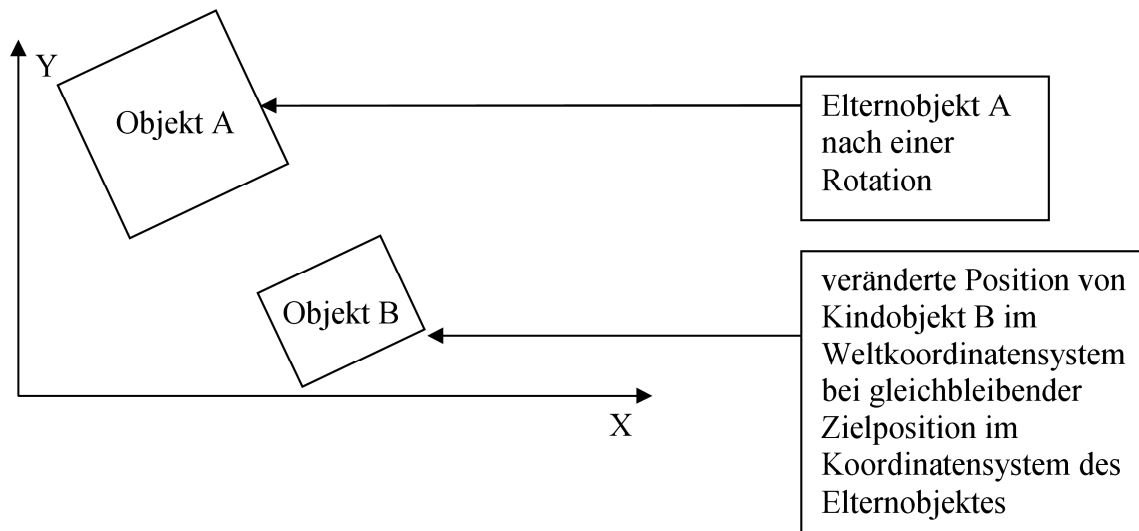


Abbildung 59: Positionskommunikation zwischen Eltern- und Kindobjekten, Teil 2

Die Rotation von Objekt A hat Objekt B im Weltkoordinatensystem mitbewegt, so dass Objekt B im Weltkoordinatensystem nicht mehr an der gewünschten Position ist.

Es ist also von Vorteil, wenn Elternobjekte in jedem Simulationszwischen schritt immer vor ihren Kindobjekten simuliert werden, damit die Kindobjekte bei ihrer Simulation die Transformation des Elternobjektes sehen, die im aktuellen Simulationszwischen schritt bis zum Ende dieses Simulationszwischen schritts gültig bleibt. Dies gilt allerdings auch nur dann, wenn die Elternobjekte nur selbst ihre Transformation ändern. Ändert ein anderes SO „X“ die Transformation eines Elternobjektes „A“, und wird „X“ in jedem Simulationszwischen schritt nach den Kindern des Objektes A simuliert, so kann auch dann noch der oben gezeigte Effekt eintreten.

Um sicherzustellen, dass Elternobjekte in jedem Simulationszwischen schritt vor ihren Kindobjekten simuliert werden, werden alle Elternobjekte wie in Kapitel 9.2 erläutert vor ihren Kindobjekten in die Objektliste eingetragen.

Dies ist auch für die Simulation von Mechanik erforderlich. Denn sowohl die hier angenommene Art der Simulation von Mechanik (s. Kap.9.5) als auch die in der technischen Umsetzung verwendete Komponente zur Simulation von Mechanik (s. Kap.10.2.3) arbeiten mit Weltkoordinaten. Dennoch können einige mechanisch simulierte Objekte Kinder anderer mechanisch simulierter Objekte sein. Die mechanisch simulierten Reifen eines Fahrzeugs wurden hier in praktischen Anwendungen als Kindobjekte der ebenfalls mechanisch simulierten Karosserie definiert.

Um nun die neue nach den Gesetzen der Mechanik berechnete Transformation eines Kindobjektes aus dem Weltkoordinatensystem in das Koordinatensystem des Elternobjektes

zu übertragen, muss die für den aktuellen Simulationszwischen-schritt endgültige Transformation des Elternobjektes vorher bekannt sein.

Dies wird durch die hier beschriebene Ordnung der Objektliste gewährleistet. Denn die hier angenommene Methode zur Simulation von Mechanik berechnet aufgrund dieser Ordnung in jedem Simulationszwischen-schritt zuerst die neue Transformation eines Elternobjektes und erst dann die neuen Transformationen seiner Kindobjekte. Dies wird dadurch gewährleistet, dass die Simulation von Mechanik hier als VH angenommen wird, die jedem SO zugeordnet wird, dass nach den Gesetzen der Mechanik simuliert werden soll (siehe Kap. 9.5). Diese VH wird wie jede andere VH eines SOs ausgeführt und berechnet dabei nur die neue Transformation des zugehörigen SOs (und noch nicht die der anderen SOs). Somit überträgt sich die Reihenfolge, in der die zugehörigen SOs in jedem Simulationszwischen-schritt simuliert werden auf die Reihenfolge, in der ihre VHs zur Simulation von Mechanik ausgeführt werden und eine neue Transformation für das jeweilige SO berechnen. Analog wird beim Lesen der Simulationsergebnisse aus der Komponente zur Simulation von Mechanik, die in der technischen Umsetzung verwendet wird (siehe Kap. 10.2.3), zuerst die neue Transformation eines Elternobjektes gelesen und erst dann die neuen Transformationen seiner Kindobjekte. Dies wird dadurch gewährleistet, dass diese Daten entsprechend der Reihenfolge der SOs in der Objektliste ausgelesen werden.

9.4.5 Designvorschriften bezüglich der Ausführungsreihenfolge von Verhaltensweisen

Die Reihenfolge, in der die VHs eines gegebenen Simulationsobjektes in einem Simulationszwischen-schritt ausgeführt werden, hängt nach Kapitel 9.2 vom Aufbau der zugehörigen OT-Hierarchie und von der Deklarationsreihenfolge der Container-Felder ab. Diese Ausführungsreihenfolge kann damit relativ frei festgelegt werden. Diese Möglichkeit sollte beim Design der OT-Hierarchien genutzt werden, um den Datenfluss im Sinne von Kapitel 9.4.3 möglichst günstig zu gestalten.

Ein Beispiel hierfür ist die Haupt-VH „MCB_Car“ von „MCD_Car“, die ein Fahrzeug steuern kann. Die Reihenfolge, in der die Container Felder der Unter-VHs von „MCB_Car“ deklariert sind, wurde in Kapitel 8.6.2 auf Seite 148 angegeben.

Es wurde eine Reihenfolge gewählt, die für den Datenfluss zwischen den VHs möglichst günstig ist. In jedem Simulationszwischen-schritt wird zuerst die VH „MCB_CarControlAI“. Sie trifft die höheren Entscheidungen des Fahrzeugs, wie z.B. die optimale Kurvengeschwindigkeit oder die Zielfahrtrichtung sowie die Zustände von Gaspedal und Bremse. Die nachfolgend ausgeführte VH „MCB_CarAutoGear“ bestimmt unter anderem aus der aktuellen Motordrehzahl einen passenden Gang und legt diesen ein. Dazu wird auch der Zustand der Kupplung verändert und basierend auf dem Gaspedalzustand, den „MCB_CarControlAI“ bestimmt hat, wird ein eventuell abweichender Zustand des Gaspedals bestimmt, da bei ausgekuppeltem Getriebe auch die Gaszufuhr herabgeregelt wird. Die nachfolgend ausgeführte VH „MCB_CarControlBasic“ verwendet diese Werte wie die Zustände von Gaspedal, Bremspedal, Getriebe und Kupplung sowie die Zielfahrtrichtung und setzt diese Werte in konkrete mechanische Wirkungen um wie Drehmomente an den Rädern oder einen neuen Einschlagwinkel der Vorderräder zum Lenken. Die VH „MCB_CarLightController“ leitet schließlich aus dem neuen Gesamtzustand des Fahrzeugs einen passenden Zustand der Lichtanlage ab, indem sie z.B. beim Abbiegen den entsprechenden Blinker setzt, beim Bremsen die Bremsleuchten einschaltet oder beim Rückwärtsfahren die Rückleuchten aktiviert.

Die Ausführungsreihenfolge der Unter-VHs wurde also so festgelegt, dass SP-Werte durch eine VH neu bestimmt werden, bevor andere Unter-VHs von „MCB_Car“ diese SP-Werte in demselben Simulationszwischen Schritt lesen und verwenden. So treten keine Verzögerungen der Form auf, dass eine VH einen neuen SP-Wert bestimmt, der von einer VH weiterverarbeitet werden soll, die erst im nächsten Simulationszwischen Schritt wieder ausgeführt wird und erst dann den neuen Wert lesen kann.

In den meisten Fällen kann ein möglicher Verzögerungseffekt durch eine Erhöhung der Anzahl an Simulationszwischen Schritten verringert werden. Möchte man derartige Verzögerungen jedoch gänzlich ausschließen, muss man die Reihenfolge der VHs und damit deren Ausführungsreihenfolge wie im vorangehenden Beispiel entsprechend festlegen. Dies kann auch erforderlich werden, wenn bestimmte Berechnungen einer VH „A“ aus Gründen der Effizienz immer nur im letzten Simulationszwischen Schritt jedes Simulationsschrittes durchgeführt werden und die Ergebnisse dieser Berechnungen von einer anderen in jedem Simulationszwischen Schritt **vorher ausgeführten** VH „B“ benötigt werden. Dann kann auch eine Erhöhung der Anzahl der Zwischen Schritte mögliche Verzögerungen nicht vermindern, da die VH „B“ frühestens im nächsten Simulationsschritt diese Ergebnisse verarbeiten und darauf reagieren kann. Diese Reaktion von VH „B“ wäre dann nicht mehr im aktuellen Frame sondern erst im nächsten Frame sichtbar, unabhängig von der Zahl der Zwischen Schritte.

Ein Beispiel dafür ist der Objekttyp „MCD_FoleySound“, dessen Instanzen Tonquellen in der Szene repräsentieren. Seine Haupt-VH „MCB_FoleySound“ enthält „MCB_SoundAverager“ als erste und „MCB_DTsoundPitchController“ als zweite Unter-VH. Die VH „MCB_SoundAverager“ kann durch andere VHs gesteuert werden, indem z.B. Werte für die gewünschte Tonhöhe in die SP „preRtPitch“ geschrieben werden und Werte für die gewünschte Tonlautstärke in die SP „preVolume“ übertragen werden. Dies kann auf der Basis von Werten aus der Simulation von Mechanik geschehen, die unter anderem feststellen kann, mit welcher Geschwindigkeit ein Objekt über die Oberfläche eines anderen Objektes gleitet. So ist es beispielsweise möglich, bei einem nach den Gesetzen der Mechanik simulierten Fahrzeug ein passendes Quietschen der Reifen zu erzeugen, wenn diese über den Asphalt rutschen und nicht rollen.

Die berechnete Reibungsgeschwindigkeit, aus der die Tonhöhe und die Tonlautstärke für das Reifenquietschen abgeleitet werden, unterliegt relativ starken Schwankungen. Um diese Schwankungen hörbar abzuschwächen, wird ein Durchschnitt über mehrere zuletzt erzeugte Werte berechnet, so dass der Gesamtverlauf dieser Durchschnittswerte gegenüber dem Verlauf der Eingangswerte geglättet wird. Zu diesem Zweck kann die VH „MCB_SoundAverager“ die Eingangswerte über einen wählbaren Zeitraum der Länge „averagingTime“ sammeln und dann im letzten Simulationszwischen Schritt jedes Simulationsschrittes den Durchschnittswert der Eingangswerte für „preRtPitch“ und „preVolume“ berechnen, die in dem jeweils zurückliegenden Zeitraum der Länge „averagingTime“ geliefert wurden. Die Durchschnittswerte für Tonlautstärke und Tonhöhe werden dann in die SPs „SFXvolume“ bzw. „realtimePitch“ geschrieben. Die SP „SFXvolume“ ist dabei eine Standard-SP, deren Wert direkt auf den erzeugten Ton wirkt. Die SP „realtimePitch“ ist dagegen nur ein vorläufiger Wert, der von „MCB_DTsoundPitchController“ weiterverarbeitet wird. Weil „MCB_SoundAverager“ diese Durchschnittswerte aus Gründen der Effizienz immer nur in dem letzten Simulationszwischen Schritt berechnet, weil sie erst hier für den Szenenzustand n wirklich benötigt werden, muss „MCB_DTsoundPitchController“ noch in demselben Simulationszwischen Schritt den Durchschnittswert „realtimePitch“ weiterverarbeiten,

damit das Ergebnis noch in den Szenenzustand n geschrieben werden kann. Andernfalls würde sich immer eine Verzögerung um genau einen Frame ergeben. Die Verzögerung könnte auch durch eine beliebig hohe Anzahl an Simulationszwischenritten nicht verringert werden. Aus diesem Grund ist „MCB_DTsoundPitchController“ die zweite Unter-VH **nach** „MCB_SoundAverager“ in „MCB_FoleySound“, weil sie so direkt in demselben Simulationszwischenritt auf die Ergebnisse von „MCB_SoundAverager“ reagieren kann. Die VH „MCB_DTsoundPitchController“ stellt anhand der Rückgabewerte der Methoden `getDeltaT` und `getRealtimeDeltaT`⁶⁴ fest, ob die Simulation aktuell einen Zeitlupen- oder Zeitraffereffekt verwendet und passt die Tonhöhe entsprechend an, welche dann in die endgültige Standard-SP „SFXpitch“ für die Tonhöhe übertragen wird.

Es folgt das entsprechende Codefragment aus „MCB_DTsoundPitchController“:

```
public void updateBeforeSubBehaviours(long f)
{
    double realtimePitch=getDoubleValue("realtimePitch");

    double pitch=realtimePitch*(getDeltaT()/getRealtimeDeltaT());

    setDoubleValue("SFXpitch",pitch);
}
```

9.4.6 Simulation nach dem Gesamtschrittverfahren als mögliche Alternative

Es wäre möglich, den Simulationsmechanismus nach dem Prinzip des Gesamtschrittverfahrens (siehe Kap. 9.1) so zu konstruieren, dass die Ausführungsreihenfolge der VHs wesentlich weniger Einfluss auf das Simulationsergebnis hätte als bei dem implementierten Einzelschrittverfahren.

Wenn statt nur einem zwei MaxControl-Zustände verwendet würden, könnten in jedem Simulationszwischenritt der Ausgangszustand und der entstehende Zustand getrennt werden. Dies könnte dadurch realisiert werden, dass alle lesenden Zugriffe auf SPs nur den ersten Zustand betreffen, während schreibende Zugriffe auf SPs nur auf den zweiten Zustand wirken. Nach jedem Simulationszwischenritt würde der zweite Zustand zum ersten Zustand für den folgenden Simulationszwischenritt. Nach dem letzten Simulationszwischenritt zu einem Frame n würde der entstandene zweite Zustand in den Szenenzustand n übertragen und unter Berücksichtigung der in Frame $n+1$ manuell kontrollierten Werte als Ausgangszustand des folgenden Simulationszwischenritt zu Frame $n+1$ gewählt.

So existieren dann in jedem Simulationszwischenritt zwei Zustände, von denen nur aus dem ersten gelesen wird und nur der zweite verändert wird. Da die VHs immer nur den ersten Zustand sehen würden, wäre das Simulationsergebnis unter bestimmten Bedingungen nicht mehr von der Reihenfolge abhängig, in der die VHs ausgeführt werden und die VHs würden bei jeder möglichen Ausführungsreihenfolge die gleichen Simulationsergebnisse in die entsprechenden SPs schreiben.

Diese Reihenfolgeunabhängigkeit kann jedoch leicht gestört werden:

⁶⁴ Erläutert in Kapitel 8.1.2.

Wenn die Simulationsergebnisse der VHs nicht nur von den Werten der SPs abhängen, sondern die VHs über interne Variablen oder Methoden miteinander interagieren, wäre eine erneute Abhängigkeit von der Ausführungsreihenfolge möglich. Denn die internen Variablen sind von der hier vorgeschlagenen Zustandstrennung ausgenommen und Methoden können ebenfalls die Werte solcher internen Variablen lesen und schreiben.

Eine weitere Bedingung für eine reihenfolgeunabhängige Simulation ist, dass in jedem Simulationszwischen Schritt keine zwei VHs auf dieselbe SP schreibend zugreifen dürften. Denn dann sind schon Anweisungen wie die folgenden reihenfolgeabhängig:

```
VH1: setLongValue ("X", 11) ;  
VH2: setLongValue ("X", 21) ;
```

Eine derartige Situation haben wir in 9.1 für Gesamtschrittverfahren generell verworfen.

Auch bei sequentiellen Zugriffen auf externe Daten oder bei Verwendung von Zufallszahlen, kann es von der Ausführungsreihenfolge der VHs abhängig sein, welche VH welche dieser Daten erhält, selbst wenn die in Kapitel 9.4.2 angegebenen Designvorschriften beachtet werden.

Wären jedoch alle Probleme dieser Art ausgeschlossen, so wäre der Hauptvorteil dieser Methode, dass das Simulationsergebnis nicht mehr von der Ausführungsreihenfolge der VHs abhängen würde. Damit wäre auch nicht länger die Reihenfolge relevant, in der Geschwisterobjekte im verwendeten 3D-Animationsprogramm gespeichert werden.

Ein technischer Vorteil wäre die Verteilbarkeit der Simulation, da die Berechnungen einer VH nun nicht mehr von den Zustandsänderungen abhängen, die eine andere VH vorher in demselben Simulationszwischen Schritt durchführt. Die VHs könnten also zeitgleich auf verschiedenen Rechnern ausgeführt werden. Ebenso könnte die Simulation der VHs dann auf mehrere Prozessoren eines Rechners verteilt werden. Deshalb bleibt die Implementierung eines Gesamtschrittverfahrens für zukünftige Weiterentwicklungen von MaxControl zu bedenken, besonders im Hinblick auf Multiprozessorsysteme und die zu deren Nutzung erforderliche Parallelisierbarkeit.

Der Nachteil dieses Systems wäre allerdings, dass der in Kapitel 9.4.3 erläuterte Verzögerungseffekt nun immer auftreten würde, sowohl zwischen verschiedenen VHs als auch innerhalb einer VH, weil die Ergebnisse eines Simulationszwischen Schrittes nun in jedem Fall von den VHs erst im folgenden Simulationszwischen Schritt gelesen werden können und keinesfalls noch in demselben Simulationszwischen Schritt.

Bezüglich der SPs würde sich auch der Speicherbedarf verdoppeln, der nun für die beiden MaxControl-Zustände erforderlich wäre.

9.5 Simulation von Mechanik als Teilbereich der Physik

Es wird vor Kapitel 10 zunächst angenommen, dass die Simulation von Mechanik anders als in der technischen Umsetzung (s. Kap. 10) nicht durch eine externe Softwarekomponente sondern direkt durch eine bestimmte VH durchgeführt wird. Diese VH sei als „MCB_Physics“ bezeichnet.

Ein Simulationsobjekt, das in seiner SO-Hierarchie die VH „MCB_Physics“ enthält, wird nachfolgend auch als MSO bezeichnet. Es folgt automatisch den Gesetzen der Mechanik, wenn diese VH eingeschaltet ist. Ist sie ausgeschaltet, bleibt das SO in der

Mechaniksimulation, es folgt ihr jedoch nicht mehr. Stattdessen werden alle Positions- und Rotationsveränderungen, die eine andere VH eventuell an diesem SO vornimmt, in der Mechaniksimulation erzwungen, so dass andere 3D-Objekte in der Mechaniksimulation ggf. mit entsprechenden Kräften beiseite gedrückt werden, um diese Änderungen zu ermöglichen (siehe auch Kap. 10.2.2). So wird auch verfahren, wenn für die Position oder die Rotation eines MSOs manuelle Kontrolle vorliegt. Diese beiden Möglichkeiten wurden in der Animation genutzt, die in Kap. 10.4.1.6 erläutert wird.

Wird diese VH in einem MSO ausgeführt, so berechnet sie nur den neuen Zustand dieses MSOs und nicht den neuen Zustand der anderen MSOs, da dieser von deren VH „MCB_Physics“ berechnet wird, sobald diese aufgerufen wird.

Jede VH kann Kräfte setzen⁶⁵, die auf ein beliebiges MSO wirken sollen. Da die Wirkungen dieser Kräfte erst berechnet werden, wenn die VH „MCB_Physics“ des betroffenen MSOs ausgeführt wird, müssen diese Kräfte bis dahin zwischengespeichert werden.

Dazu könnte jede Instanz der VH „MCB_Physics“ Listen der Klasse `Vector` besitzen, welche die bisher für das zugehörige MSO gesetzten Kräfte speichern. Diese müssten für jedes MSO gesammelt und gespeichert werden, bis sie bei dem nächsten Aufruf der VH „MCB_Physics“ dieses MSOs verarbeitet werden. Direkt danach können und müssen sie gelöscht werden, damit diese Kräfte kein zweites Mal an dem MSO wirken, ohne erneut gesetzt worden zu sein. Ob eine gesetzte Kraft noch im aktuellen Simulationszwischen schritt Auswirkungen auf das betroffene MSO hat, hängt also davon ab, ob die VH „MCB_Physics“ dieses MSOs im aktuellen Simulationszwischen schritt bereits ausgeführt wurde oder noch ausgeführt werden wird. Im ersten Fall würde eine Verzögerung im Sinne von Kapitel 9.4.3 auftreten, während die Kraft im zweiten Fall noch auf das Ergebnis des aktuellen Simulationszwischen schrittes wirken würde.

Da die Simulation von Mechanik mit Fähigkeiten wie der Simulation von Reibungskräften und Federn und der Erkennung und Auflösung von Kollisionen zwischen MSOs ein komplexes Problem ist, für das hier in der technischen Umsetzung eine externe mit Shockwave entwickelte Komponente verwendet wurde (s. Kap. 5.6 und 10.2.3), soll hier nicht näher darauf eingegangen werden, wie eine VH zu implementieren wäre, welche ebenfalls die Fähigkeiten dieser Komponente hätte.

Die in der technischen Umsetzung entwickelte Lösung wird in Kapitel 10.2.3 erläutert.

⁶⁵ Siehe entsprechende Methoden `addCenterForce`, `addForce`, `addLocalCenterForce`, `addLocalForce`, `addTorque` und `addWorldTorque` aus Kapitel 8.3.7.

10 Die technische Umsetzung

Dieses Kapitel geht näher auf die konkrete technische Umsetzung der Prinzipien aus den vorangehenden Kapiteln ein, die zur Implementation des Simulationssystems MaxControl führte.

MaxControl wurde bisher für die Unterstützung eines konkreten 3D-Animationssystems entwickelt. Als 3D-Animationssystem wurde 3D-Studio-MAX gewählt (siehe Kapitel 5.2). Eine Unterstützung weiterer 3D-Animationsprogramme ist als mögliche spätere Weiterentwicklung von MaxControl geplant.

Für dieses Kapitel werden insbesondere die Erläuterungen aus Kapitel 5 vorausgesetzt. Es werden hier technische Details behandelt, die in Kapitel 5 noch nicht erläutert wurden.

In Kapitel 2.2 wurde ein Überblick über die technische Architektur von MaxControl gegeben, auf den hier nur verwiesen werden soll ohne diesen zu wiederholen. Ebenso fasste Kapitel 5 bereits die als Komponenten verwendeten Werkzeuge und ihre Aufgaben in MaxControl zusammen.

10.1 Technische Details wichtiger Klassen

Dieses Kapitel geht analog zu Kapitel 8 auf die in MaxControl verwendeten Klassen ein. Die Klassen aus Kapitel 8 werden hier als bekannt vorausgesetzt und es werden technische Details einiger dieser Klassen näher erläutert. Zusätzlich werden weitere Klassen erläutert, die eher technische als konzeptionelle Relevanz haben und daher in Kapitel 8 noch nicht behandelt wurden. Auch in diesem Kapitel werden jedoch nicht alle Methoden und Felder der behandelten Klassen erläutert, ebenso bleiben einige Klassen von MaxControl auch hier unbehandelt, da eine vollständige Erläuterung zu umfangreich und für das Gesamtverständnis des Systems nicht ausreichend relevant wäre.

10.1.1 MC_Property: Definition von GUI-Repräsentationen für SPs

Die Klasse `MC_Property` wurde bereits grundsätzlich in Kapitel 8.2 beschrieben.

Methoden dieser Klasse zur Kommunikation mit 3D-Studio-MAX, um SP-Werte aus der Szenenzustandsfolge zu lesen oder in diese zu schreiben, werden im Kapitel 10.3.1.3 in ihren optimierten Formen aus der Unterklasse `MC_PropertyCommOptimized` erläutert. Erforderliche Anpassungen der dabei involvierten Methoden für die Implementation neuer SP-Typen werden hier nicht behandelt.

Eine hier nicht näher erläuterte Gruppe von Methoden ermöglicht die Definition eigener GUI-Repräsentationen in 3D-Studio-MAX für SPs und ggf. auch für ihre MPs. Für nicht-standard SPs können über diese Methoden auch die SPs selbst zu den 3D-Objekten in 3D-Studio-MAX hinzugefügt werden, ebenso wie die MPs für animierbare SPs.

Grundsätzlich können Unterklassen `A` von `MC_Property` diese Methoden so implementieren, dass sie zur Erstellung von Repräsentationen von SPs dieser Klasse `A` in der grafischen Benutzeroberfläche von 3D-Studio-MAX verwendet werden können. Diese GUI-Repräsentationen werden über MAXScript-Anweisungen erzeugt, welche von diesen Methoden generiert werden.

Wie eine solche Repräsentation für grafische Vektoren der Klasse „MCD_Vector“ aussehen kann zeigt Abbildung 25 in Kapitel 6.6 auf Seite 89. Für nähere Erläuterungen zur Klasse „MCD_Vector“ siehe Kapitel 7.2.4 ab S. 108.

10.1.2 MC_Owner: Zugriff auf die Transformationen von Simulationsobjekten

Über die folgenden Methoden kann auf die Transformationen von SOs zugegriffen werden. Die Transformation enthält die Position, Rotation und Skalierung eines SOs (siehe Kap. 3.2). Die Klasse MC_Owner ist eine Oberklasse von MC_Object und MC_Behaviour. Damit können diese Methoden nicht nur in SOs direkt, sondern insbesondere auch in ihren VHs aufgerufen werden, wobei sich diese Methoden dann immer auf das in rootOwner (siehe Kap. 8.1.1) gespeicherte zugehörige SO beziehen.

Die Transformationen werden als Objekte der Klasse MCH_Transform3D gespeichert, die weiter unten in Kapitel 10.1.3 erläutert wird.

MCH_Transform3D getTransform() :

Diese Methode liefert die Transformation des SOs zurück. Hat das SO ein Elternobjekt, so bezieht sich die Transformation auf das lokale Koordinatensystem des Elternobjekts, andernfalls auf das Weltkoordinatensystem (siehe dazu Kap. 3.3).

Im Vergleich dazu liefert **MCH_Transform3D getTransformAtWorldCS()** die Transformation des SOs im Weltkoordinatensystem zurück, auch wenn es ein Elternobjekt hat.

Über die Methode **MCH_Transform3D getTransformAtCSof(MC_Owner other)** erhält man die Transformation eines SOs „A“ im Koordinatensystem eines beliebigen anderen SOs „B“ zurück, auch wenn „B“ kein direktes oder indirektes Elternobjekt von „A“ ist. So stellen beispielsweise Fahrzeuge der Klasse MCD_Car fest, wo sie sich im Koordinatensystem eines SOs der Klasse MCD_LaneMarker befinden. SOs dieser Klasse repräsentieren Abschnitte einer Fahrstrecke, der das Auto mittig folgen soll. Innerhalb eines solchen Abschnittes kann das Auto so leicht feststellen, ob es sich nach links oder rechts von der Mittellinie entfernt und entsprechend gegenlenken muss.

void setTransform(MCH_Transform3D trans) :

Wurde ein durch die vorangehend erläuterten Methoden erhaltenes Transformationsobjekt der Klasse MCH_Transform3D verändert (siehe dazu Kapitel 10.1.3), so kann es über diese Methode wieder auf das SO übertragen werden.

Da ein SO seine Transformation in Weltkoordinaten oder ggf. im Koordinatensystem seines Elternobjektes speichert, muss das übergebene Transformationsobjekt vorher eventuell umgewandelt werden, wenn es sich auf ein anderes Koordinatensystem bezieht. Das SO, auf dessen Koordinatensystem sich ein Transformationsobjekt bezieht, ist daher stets im Transformationsobjekt gespeichert (im Falle des Weltkoordinatensystems als NULL, siehe dazu nächstes Kapitel 10.1.3). So kann setTransform die eventuell nötige Umwandlung automatisch durchführen.

10.1.3 MCH_Transform3D (Transformationen)

Die Klasse `MCH_Transform3D` gehört zwar nach den Namenskonventionen aus Kapitel 7.3 zur Klasse `MC_Help` und ist damit eine Hilfsklasse, die weder SOs noch VHs repräsentiert, tatsächlich erbt sie jedoch von der Klasse `Transform3D` aus der Java-Erweiterung `Java-3D` (siehe Kap. 5.8). Schon `Transform3D` kann zur Speicherung und Manipulation von Transformationen für 3D-Objekte verwendet werden. Die Unterklasse `MCH_Transform3D` erweitert diese Klasse nun um nützliche und erforderliche Eigenschaften und Funktionen und ihre Instanzen speichern Transformationsobjekte, welche die Methoden aus Kap. 10.1.2 zurückliefern bzw. annehmen.

In Bezug auf die aus `Transform3D` geerbten Felder und Methoden sei auf die offiziellen Anleitungen zu `Java-3D` verwiesen⁶⁶. Im Folgenden werden einige wichtige Felder und Methoden erläutert, die in `MCH_Transform3D` dazukommen.

MCO_WithTransform atCSof:

Dieses Feld speichert das SO, in dessen Koordinatensystem die Transformation liegt. Da die Transformation eines SOs „A“ mit einem Eltern-SO „B“ intern im Koordinatensystem des Eltern-SOs „B“ gespeichert wird, liefert die Methode `getTransform` aus Kapitel 10.1.2 in SO „A“ aufgerufen ein Objekt der Klasse `MCH_Transform3D` zurück, in dessen Feld `atCSof` eine Referenz auf das Eltern-SO „B“ gespeichert ist. Hat „A“ kein Elternobjekt, so wird seine Transformation im Weltkoordinatensystem zurückgeliefert mit dem Wert `NULL` im Feld `atCSof`.

Diese Information kann die oben in Kapitel 10.1.2 behandelte Methode `setTransform` wie dort erläutert verwenden, um eine übergebene Transformation ggf. in das erforderliche Koordinatensystem umzuwandeln.

Dem erweiterten Konstruktor `MCH_Transform3D(MCO_WithTransform aco)` der Klasse `MCH_Transform3D` wird das SO für das oben erläuterte Feld `atCSof` als Parameter übergeben.

Eine ggf. ins Weltkoordinatensystem umgewandelte Kopie der gespeicherten Transformation kann über die Methode `MCH_Transform3D getThisTransformAtWorldCS()` erhalten werden.

Analog kann eine Kopie der Transformation über die Methode `MCH_Transform3D getThisTransformAtCSof(MC_Owner other)` auch im lokalen Koordinatensystem eines beliebigen SOs geliefert werden, das als Parameter übergeben wird.

10.1.4 MCH_Spring (Federn)

Die Klasse `MCH_Spring` erbt von der Klasse `MC_Help` und ist damit eine Hilfsklasse, die weder SOs noch VHs repräsentiert sondern Federn, die innerhalb der Mechaniksimulation verwendet werden können, um SOs miteinander zu verbinden. Die Federn können mit der in Kapitel 8.3.7 erläuterten Methode `createSpring` erstellt werden und sind dann in der Liste `springs` gespeichert, die in Kapitel 8.4.3 behandelt wurde.

⁶⁶ <https://java3d.dev.java.net/binary-builds.html>, unter „API Documentation & Utils“, 12.05.2007

Die nachfolgend erläuterten Methoden der Federn erlauben vor und während der Simulation eine Festlegung bzw. Änderung der Federeigenschaften, die sich auf den Ablauf der Mechaniksimulation auswirken können. In Kapitel 10.2.3.3 wird erläutert, wie ein SO des Typs `MCD_CarTyre`, das ein Autorrad repräsentiert, Federn verwendet, um eine Radachse nachzubilden, die das Rad mit der Karosserie des Fahrzeugs verbindet.

`void deleteSpring() :`

Diese Methode löscht die Feder aus der Mechaniksimulation und aus der Liste `springs` des SOs, das diese Feder ursprünglich erstellt hat.

Die Methoden **`void setObject1(MC_Object _o1)`** und **`void setPoint1(MCH_Vector3D t)`** legen das als Parameter übergebene SO „X“ fest, dass an einer Seite „A“ der Feder befestigt sein soll bzw. den Punkt, an dem die Feder mit ihrer Seite „A“ an SO „X“ befestigt sein soll. Dieser Punkt kann auch außerhalb des Volumens von SO „X“ liegen. Er wird innerhalb des Koordinatensystems von SO „X“ angegeben.

Der zur Festlegung dieses Punktes übergebene Parameter wird als Objekt der hier nicht dokumentierten Hilfsklasse `MCH_Vector3D` angegeben, die dreidimensionale Vektoren und auch Punkte im 3D-Raum speichern kann. Ein Punkt mit den Koordinaten (x, y, z) kann erzeugt werden als `„v=new MCH_Vector3D(x, y, z)“`, wobei x, y und z den Typ `double` haben müssen. Der Zugriff auf die Wertekomponenten x, y und z ist in diesem Beispiel über `v.x`, `v.y` und `v.z` möglich.

Die Methoden **`setObject2`** und **`setPoint2`** legen analog das SO und den Befestigungspunkt für die andere Seite „B“ der Feder fest.

Die nachfolgend genannten Methoden legen jeweils die Ruhelänge, die Elastizität (=Stärke) bzw. die Dämpfung der Feder fest und ob die Feder auf Stauchung bzw. Dehnung mit entsprechenden Gegenkräften reagieren soll:

```
void setRestLength(double rl)  
void setElasticity(double el)  
void setDamping(double da)  
void setOnCompression(boolean oc)  
void setOnExtension(boolean oe)
```

10.1.5 MCH_CollisionInfo (Informationen über Kollisionen)

Die Klasse `MCH_CollisionInfo` gehört nach den Namenskonventionen aus Kapitel 7.3 zur Klasse `MC_Help` und erbt auch von dieser. Ein Objekt dieses Typs repräsentiert Informationen über eine Kollision zwischen zwei SOs, die nach den Gesetzen der Mechanik simuliert werden (=MSOs).

Objekte dieses Typs werden von den in Kapitel 8.3.7 erläuterten Methoden `getCollisions` und `getGlobalCollisions` als Elemente der zurückgegebenen Listen erzeugt.

Die Mechaniksimulation liefert nur die in den folgenden Feldern der Klasse `MCH_CollisionInfo` gespeicherten Daten über eine Kollision:

MC_Object o1, o2: Die MSOs, die an der Kollision beteiligt sind.

MCH_Vector3D contactPoint: Der „Kollisionspunkt“ im Weltkoordinatensystem, an dem beide Objekte kollidiert sind.

MCH_Vector3D contactNormal: Die „Kollisionsnormale“. Sie entspricht der Oberflächennormalen des SOs o2 am Kollisionspunkt contactPoint.

double normalRelativeVelocity: Die relative Kollisionsgeschwindigkeit, mit der beide SOs entlang der Kollisionsnormalen kollidiert sind. Da dieser Wert ein Maß für die Stärke des Aufpralls ist, können auf dieser Basis automatisch Toneffekte für ein Aufprallereignis erzeugt werden (siehe Kap. 10.2.4).

Die dabei oft für dreidimensionale Werte verwendete Hilfsklasse `MCH_Vector3D` wurde im vorangehenden Kapitel 10.1.4 im Zusammenhang mit der Methode `setPoint1` kurz erläutert.

Ein Objekt der Klasse `MCH_CollisionInfo` leitet aus den obigen Daten weitere Informationen über die Kollision ab, die mit den im Folgenden erläuterten Methoden erhalten werden können. Diese Informationen werden mit Hilfe zusätzlicher Angaben über die beiden an der Kollision beteiligten MSOs bestimmt, wie ihrer aktuellen linearen Geschwindigkeit und Drehgeschwindigkeit.

MCH_Vector3D getSkiddingVector():

Der zurückgegebene Vektor gibt die Richtung und den Betrag der Geschwindigkeit an, mit der SO o1 am Kollisionspunkt über die Oberfläche von SO o2 reibt. Dabei werden die linearen Geschwindigkeiten und die Winkelgeschwindigkeiten beider SOs berücksichtigt. Auf dieser Basis können automatisch Toneffekte bei Reibung erzeugt werden, wie z.B. quietschendes Metall oder Autoreifen (siehe Kap. 10.2.4).

Analog gibt `MCH_Vector3D getRollingVector()` die entsprechende Rollgeschwindigkeit an, die z.B. für Rollgeräusche von auf Asphalt rollenden Autoreifen oder Tonnen verwendet werden kann (siehe Kap. 10.2.4).

MCH_AxisAngleDegrees getRollingAngularVelocity():

Analog zu `getRollingVector` gibt diese Methode die Rollgeschwindigkeit als Winkelgeschwindigkeit in Grad zusammen mit einer Drehachse an. Dieser Wert wird als Objekt der hier nicht dokumentierten Hilfsklasse `MCH_AxisAngleDegrees` angegeben. Diese speichert eine Rotationsachse im 3D-Raum über drei Komponenten (x, y, z) und die Winkelgeschwindigkeit w in Grad pro Sekunde. Der Zugriff auf die Wertekomponenten x, y, z und w ist in einem Objekt v dieser Klasse über `v.x`, `v.y`, `v.z` bzw. `v.angleDegrees` möglich.

10.2 Simulation

Der Ablauf einer Simulation mit MaxControl wurde in Kapitel 9 bereits behandelt, hier wird die Simulation unter Aspekten der technischen Umsetzung näher erläutert, wobei auf technisch bedingte Besonderheiten, Probleme und Problemlösungen eingegangen wird.

10.2.1 Vorbereitender Schritt bezüglich der Simulationseigenschaften

Bevor eine Simulation mit MaxControl gestartet werden kann, müssen die SPs und ggf. ihre MPs den 3D-Objekten in der Szenenzustandsfolge hinzugefügt worden sein. Dazu generieren die im Verlauf der Szenenanalyse aus Kap. 9.2 erzeugten SOs selbst entsprechende MAXScript-Anweisungen. Da dieser Vorgang, der auf den in Kapitel 10.1.1 angedeuteten Methoden basiert, relativ lange dauern kann, wird er nicht vor jeder Simulation ausgeführt sondern muss bei Bedarf manuell gestartet werden. Nachdem die SPs und MPs so der Szenenzustandsfolge hinzugefügt wurden, können sie mit der entsprechenden Szene gespeichert und auch wieder geladen werden.

Dennoch kann es für eine Solche Szene erforderlich werden, diesen Vorgang auch bei bereits erstellten MPs und SPs zu wiederholen. Wurden 3D-Objekte in der Szene gelöscht, hinzugefügt oder einem 3D-Objekt ein neuer Objekttyp zugewiesen, muss der Vorgang wiederholt werden, um Objektindizes anzupassen, neuen 3D-Objekten ihre SPs und MPs hinzuzufügen oder die SPs und MPs eines 3D-Objektes seinem neu festgelegten Objekttyp anzupassen. Im letztgenannten Fall führt 3D-Studio-MAX automatisch eine Schemamigration durch. Dies bedeutet, dass bei einem Objekttypwechsel bereits vorhandene SPs und MPs, die auch zum alten Objekttyp gehörten, erhalten bleiben und dabei auch ihre Werteverläufe behalten. Es werden lediglich neue benötigte SPs und MPs hinzugefügt und nicht mehr benötigte gelöscht.

10.2.2 Simulationszyklus aus technischer Sicht mit Zeitablaufkontrolle und Einbindung der Mechaniksimulation

Der Simulationszyklus wurde bereits in Kapitel 9.3 erläutert und wird hier in einer um technische Details erweiterten Version behandelt.

Insbesondere kommen hier das Lesen von globalen Parametern aus SOs des Typs `MCO_Universe`, die den Simulationsablauf beeinflussen, und die Einbindung der Physiksimulation über die externe Shockwave-Komponente hinzu. Im Folgenden werden die gestrichelt umrahmt dargestellten Anweisungen in Abbildung 51 aus Kapitel 9.3 erläutert.

Die Methode `readEnvironment` liest globale Parameter für die Simulation aus SOs des Typs `MCO_Universe`, speziell aus SOs der Klassen `MCU_Time` und `MCU_Physics`.

Wie in Kapitel 8.1.2 erläutert wurde, enthält ein SO der Klasse `MCU_Time` Parameter, die den Zeitablauf einer Simulation steuern können. Damit diese Parameter in der gesamten Simulation gültig sind, werden sie hier gelesen und auf interne, global gültige Variablen übertragen. Der folgende Auszug aus der Klassendefinition von `MCU_Time` zeigt die entsprechenden globalen Parameter, deren Verarbeitung durch `readEnvironment` nachfolgend erläutert wird:

```
public final MCP_DeltaT timePerSecond=...
public final MCP_DeltaT realTimeTimePerSecond=...
public final MCP_Long normalSubSteps=...
public final MCP_Long minNormalSubSteps=...
public final MCP_Long simSubSteps=...
```

Aus den Parametern `normalSubSteps` und `minNormalSubSteps` wird zunächst der Wert von `simSubSteps` bestimmt. Diese globale Variable legt für jeden Simulationsschritt

die Anzahl verwendeter Zwischenschritte fest, die für eine erhöhte Genauigkeit einiger Simulationsverfahren in den Verhaltensweisen nötig sein können (siehe Kap. 6.2 ab S. 78).

Der Wert wird über die folgende Formel berechnet, wobei „max“ das Maximum der beiden Parameter als Rückgabewert wählt:

```
max(minNormalSubSteps,  
    normalSubSteps*timePerSecond/realTimeTimePerSecond)
```

Das Ergebnis dieser Berechnung wird gerundet und im Parameter `simSubSteps` gespeichert. Dadurch werden mindestens so viele Zwischenschritte durchgeführt, wie der Wert von `minNormalSubSteps` angibt.

Im Normalfall, d.h. wenn `minNormalSubSteps < normalSubSteps` und `timePerSecond/realTimeTimePerSecond = 1` gelten, wird die Anzahl an Zwischenschritten aus `normalSubSteps` übernommen. Wie in Kapitel 8.1.2 erläutert wurde, ist in `realTimeTimePerSecond` die Zeit angegeben, die im Normalfall für eine Sekunde des produzierten Films in der Simulation ablaufen soll. Wie aus den dortigen Erläuterungen im Zusammenhang mit der Simulation von Toneffekten ersichtlich ist, kann man aus dem Wert von `timePerSecond/realTimeTimePerSecond` das Maß an aktueller Zeitlupe (<1) oder aktuellem Zeitraffer (>1) ableiten. Entsprechend ändert die oben angegebene Formel dann die Anzahl verwendeter Zwischenschritte. Bei Zeitlupe läuft für jeden Frame der Animation weniger simulierte Zeit ab, weshalb die Näherungsverfahren der VHs mit einer entsprechend verringerten Anzahl an Zwischenschritten für den aktuellen Frame auskommen. Analog sind mehr Zwischenschritte erforderlich, wenn ein Zeitraffer vorliegt, bei dem in jedem Frame der Animation mehr simulierte Zeit vergeht und sich so unter anderem mehrere komplexe Vorgänge wie Kollisionen innerhalb eines Frames ereignen können, die nicht mit zu großen Simulationsschritten übersprungen werden dürfen. Auf diese Weise wird die Anzahl an Zwischenschritten dynamisch dem sich eventuell ändernden Zeitablauf angepasst.

Die Parameter `timePerSecond` und `realTimeTimePerSecond` werden nun, wie in Kapitel 8.1.2 erläutert, in Abhängigkeit von dem zuvor bestimmten Wert der globalen Variablen `simSubSteps` jeweils über die Formeln

$\Delta t = (\text{timePerSecond} / \text{FrameRate}) / \text{simSubSteps}$ bzw.

$\text{simRealtimeDeltaT} = (\text{realTimeTimePerSecond} / \text{FrameRate}) / \text{simSubSteps}$

in die entsprechenden globalen Variablen überführt. Die zugrundeliegenden Werte `timePerSecond` und `realTimeTimePerSecond` werden unabhängig von der `FrameRate` und der aktuellen Anzahl von Simulationszwischenschritten je Simulationsschritt angegeben.

Wie in Kapitel 8.1.2 in Bezug auf Δt erläutert wurde, werden diese Werte nur einmal am Anfang jedes Simulationsschrittes bestimmt, so dass im Verlauf jedes Simulationsschrittes die Werte von Δt , `simRealtimeDeltaT` und `simSubSteps` konstant bleiben. Dies stimmt insbesondere in Bezug auf Δt und `simSubSteps` mit den Festlegungen aus Kapitel 6.5 überein, nach denen in jedem Simulationsschritt die Anzahl `simSubSteps` von Simulationszwischenschritten und die Simulationszeit Δt für jeden Simulationszwischenschritt konstant sind.

SOs der Klasse `MCU_Physics` enthalten Parameter, die allgemeingültig für die Mechaniksimulation sind, wie die den aktuellen Wert der Gravitation oder die aktuelle Anzahl zu berechnender Zwischenschritte für die Mechaniksimulation. Dabei ist zu beachten, dass die

Mechaniksimulation in jedem Simulations**zwischen**schrift ein bis **mehrere eigene Zwischenschritte** berechnen kann (siehe dazu weiter unten Kapitel 10.2.3.1).

Der folgende Auszug aus der Klassendefinition von `MCU_Physics` zeigt ihre durch `readEnvironment` gelesenen globalen Parameter, deren Verarbeitung nachfolgend erläutert wird:

```
public final MCP_Double gravityX=...
public final MCP_Double gravityY=...
public final MCP_Double gravityZ=...
public final MCP_Long   normalPSubSteps=...
public final MCP_Long   pSubSteps=...
```

Die ersten drei SPs `gravityX`, `gravityY` und `gravityZ` repräsentieren die 3 Komponenten des Vektors, der die aktuell wirkende Gravitation speichert. Dieser Vektor wird an die Komponente zur Simulation von Mechanik übermittelt.

Wie Kapitel 10.2.3.2 erläutert wird, kann es für die Stabilität der Mechaniksimulation erforderlich sein, dass diese in jedem Simulationszwischen-schritt mehrere eigene Zwischenschritte durchführt, so dass die Mechaniksimulation dann insgesamt für jeden Frame mehr Zwischenschritte durchführt als die Simulation der Verhaltensweisen. Der Parameter `pSubSteps` gibt die aktuelle Anzahl an Zwischenschritten an, welche die Mechaniksimulation in jedem Simulations**zwischen**schrift durchführt. Der Parameter `normalPSubSteps` gibt dagegen die Anzahl an Zwischenschritten an, welche die Mechaniksimulation in jedem Simulation**s**chritt durchführt, wenn weder Zeitlupe noch Zeitraffer vorliegen. Analog zu den Parametern `simSubSteps` und `normalSubSteps` wird der Parameter `pSubSteps` im Rahmen von Mechanismen, die weiter unten in Kapitel 10.2.3.2 zu Instabilitäten erläutert werden und die insbesondere den Parameter `normalPSubSteps` den Stabilitätserfordernissen entsprechend anpassen, über folgende Formel aus diesem Parameter hergeleitet:

```
max(1, normalPSubSteps*
    timePerSecond/realTimeTimePerSecond/simSubSteps)
```

Dadurch werden auch hier Zeitraffer- und Zeitlupen-Effekte berücksichtigt, indem die Anzahl `pSubSteps` an Zwischenschritten bei Zeitraffer erhöht und bei Zeitlupe über den Faktor `timePerSecond/realTimeTimePerSecond` herabgesetzt wird. Gleichzeitig wird jedoch auch die aktuelle Anzahl `simSubSteps` an Simulationszwischen-schritten pro Frame berücksichtigt. Wurde diese bereits wie oben erläutert für Zeitlupe oder Zeitraffer angepasst, so gleicht ihre Änderung den Faktor `timePerSecond/realTimeTimePerSecond` in der Regel ungefähr aus, wobei jedoch die Rundungsfehler bei der Berechnung von `simSubSteps` in Betracht gezogen werden müssen. Dieses Verhalten ist korrekt, denn wenn die Anzahl `simSubSteps` erhöht oder herabgesetzt wird, führt auch die Mechaniksimulation bei konstantem Wert von `pSubSteps` eine entsprechend höhere oder niedrigere Anzahl an Zwischenschritten pro simulierter Zeiteinheit durch, da `pSubSteps` die Anzahl an Zwischenschritten angibt, welche die Mechaniksimulation in **jedem** Simulations**zwischen**schrift ausführt. So muss die Anzahl `pSubSteps` in den meisten Fällen nicht angepasst werden. Jedoch kann eine Ausnahme vorliegen, wenn der Wert von `simSubSteps` bei Zeitlupe nicht mehr weiter sinkt, weil das Minimum `minNormalSubSteps` erreicht wurde. Dann führt die obige Formel dazu, dass die Anzahl `pSubSteps` entsprechend herabgesetzt wird. Durch die Einbeziehung des Minimums 1 führt

die Mechaniksimulation jedoch in jedem Simulationszwischen-schritt mindestens immer einen eigenen Simulationsschritt aus.

Liegen weder Zeitlupe noch Zeitraffer vor, so wird die Anzahl `normalPSubSteps` an Zwischenschritten, welche die Mechaniksimulation in jedem **Simulationsschritt** durchführt auf die Anzahl `simSubSteps` der Simulationszwischen-schritte verteilt, so dass die Mechaniksimulation in jedem Simulationszwischen-schritt $\text{normalPSubSteps}/\text{simSubSteps}$ Zwischenschritte ausführt.

Das Ergebnis dieser später durchgeführten Berechnung wird gerundet und in der globalen Variablen `pSubSteps` gespeichert. Beide Parameter, `normalPSubSteps` und `pSubSteps`, werden hier zunächst nur auf entsprechende globale Variablen übertragen.

Änderungen an den durch `readEnvironment` bestimmten Werten der oben behandelten globalen Variablen sind im Verlauf der Simulation auf zwei Arten möglich. Zum Einen können für die zugrundeliegenden Parameter in den SOs der Klassen `MCU_Time` und `MCU_Physics` zeitliche Änderungen durch manuelle Kontrolle festgelegt werden. Ist einer dieser Parameter in Frame `n` manuell kontrolliert, so wird dieser Parameter zu Beginn des Simulationsschrittes zu Frame `n` über `readManualValues` eingelesen und mit `readEnvironment` auf die entsprechende globale Variable übertragen. Deren Wert bleibt dann, wie auch für manuell kontrollierte SPs gefordert, über den gesamten Simulationsschritt zu Frame `n` konstant. Eine weitere Möglichkeit zur Änderung der globalen Variablen besteht darin, dass VHs in `updateFrame` einen entsprechenden Parameter ändern. Wie bereits in Kapitel 8.1.2 in Bezug auf Δt erläutert wurde, werden diese Änderungen erst beim nächsten Aufruf von `readEnvironment` auf die entsprechenden globalen Variablen übertragen und erhalten erst dann Gültigkeit. Eine Änderung an `timePerSecond`, die eine VH während des Simulationsschrittes zu Frame `n` durchführt, wird also erst zu Beginn des nachfolgenden Simulationsschrittes zu Frame `n+1` auf die globale Variable Δt übertragen und wirkt erst dann auf die Simulation. Dieser neue Wert von Δt bleibt dann für den gesamten Simulationsschritt zu Frame `n+1` konstant, auch wenn VHs im Verlauf dieses Simulationsschrittes, der aus mehreren Zwischenschritten bestehen kann, den Wert von `timePerSecond` erneut ändern. Eine Ausnahme ist dabei die oben genannte Berechnung von `pSubSteps`, da diese Berechnung vor jedem Simulationszwischen-schritt wiederholt wird.

Die Methode `setPhysicsValues` übermittelt vor Beginn der Simulation wichtige SPs von MSOs, also nach den Gesetzen der Mechanik simulierten SOs, an die Mechaniksimulation. Dazu gehören beispielweise ihre Masse, ihre Elastizität und ihr Reibungskoeffizient. Diese Werte wurden vorher über `readManualValues` aus dem Start-Frame `a` der Szenenzustandsfolge gelesen. Die geometrische Form aller MSOs wurde dabei direkt im Verlauf von `readManualValues` an die Mechaniksimulation übermittelt, beim Lesen einer speziellen SP des Typs `MCP_GeometryMessenger`. Diese speichert die Form des jeweils zugehörigen MSOs nicht im `MaxControl`-Zustand, sondern übermittelt sie nur an die Mechaniksimulation.

Es folgt eine erneute Betrachtung der Methode `updateFrame`, die hier um technische Details in Bezug auf die Mechaniksimulation erweitert wird. Die gestrichelt umrahmt dargestellten Anweisungen in Abbildung 53 aus Kapitel 9.3 werden nun erläutert.

Die Methode `setPhysicsValues` übermittelt auch hier SPs von MSOs an die Mechaniksimulation, die sich eventuell geändert haben können. Dazu gehören neben den

oben bereits erwähnten Eigenschaften Masse, Elastizität und Reibungskoeffizient beispielweise auch die Position, die Rotation, die Geschwindigkeit und die auf das MSO wirkende Kraft. Diese Werte können vorher über `readManualValues` aus dem aktuellen Frame `n` der Szenenzustandsfolge gelesen worden sein, wenn für sie manuelle Kontrolle in Frame `n` eingestellt wurde, oder sie können im vorangehenden Simulationszwischen-schritt durch Verhaltensweisen geändert worden sein.

Masse, Elastizität und Reibungskoeffizient können in jedem Frame sowohl durch manuelle Kontrolle als auch durch VHs geändert werden, diese Änderungen übermittelt `setPhysicsValues` dann an die Mechaniksimulation.

Die Geschwindigkeit kann in jedem Frame nur manuell direkt verändert werden, Änderungen an der Geschwindigkeit durch VHs werden ignoriert, da VHs MSOs ausschließlich durch das Setzen von Kräften bewegen sollen. Gleiches gilt für die Drehgeschwindigkeit.

Auch die lineare Kraft und das Drehmoment sowie die entsprechende lineare Beschleunigung und Winkelbeschleunigung, die auf das MSO wirken, können über die entsprechenden Parameter des MSOs nur durch manuelle Kontrolle beeinflusst werden, da VHs Kräfte ausschließlich durch die in Kapitel 8.3.7 erläuterten Methoden setzen sollen. Die manuell eingestellten Kräfte und Beschleunigungen werden als Kräfte den Kräften hinzugefügt, die bisher auf das MSO wirken. Dabei sind Kräfte zu beachten, die eventuell im vorangehenden Simulationszwischen-schritt durch VHs für das MSO gesetzt wurden (zu dieser Wirkungsreihenfolge siehe Kap. 10.2.3.1).

Auch die Position und die Rotation eines MSOs können durch manuelle Kontrolle gesteuert werden. Ist die Verhaltensweise „`MCB_Physics`“, die jedes MSOs in seiner SO-Hierarchie enthalten muss, abgeschaltet, so können die Position und Rotation des MSOs auch durch VHs gesteuert werden. In beiden Fällen werden die so erzeugten Bewegungen des MSOs, wie in Kapitel 9.5 erläutert, für die Mechaniksimulation erzwungen, so als hätten entsprechende Kräfte auf das MSO gewirkt. Ist die VH „`MCB_Physics`“ jedoch eingeschaltet, werden Änderungen an der Position oder Rotation, die durch VHs verursacht wurden, wie im Fall von Kräften, Beschleunigungen und Geschwindigkeiten ignoriert und es ist nur die manuelle Kontrolle weiterhin möglich.

Der Aufruf von `stepPhysics` führt einen Schritt der Mechaniksimulation aus. Simuliert wird dabei ein Schritt der Länge Δt mit `pSubSteps` Zwischenschritten. Der nachfolgende Aufruf `readPhysicsValues` liest dann die Ergebnisse der Mechaniksimulation aus der entsprechenden Programmkomponente aus und speichert sie in den zugehörigen SPs der MSOs. Dabei erhalten die MSOs auch die Listen der Kollisionen, an denen sie jeweils in diesem Mechanik-Simulationsschritt beteiligt waren (siehe Kap. 8.3.7 und 10.1.5).

10.2.3 Technik der Simulation von Mechanik

10.2.3.1 Vorgehensweise der Mechaniksimulation

Im Gegensatz zur theoretischen Betrachtung aus Kap. 9.5 ist die Mechaniksimulation in der technischen Umsetzung nicht so eng in die Simulation der VHs eingebunden, wie dort vorgeschlagen wurde. Dies ist dadurch bedingt, dass die externe in Shockwave implementierte Komponente zur Simulation von Mechanik in einem Aufruf eines Simulationsschrittes nur den nächsten Zustand aller MSOs berechnen kann und nicht isoliert den Zustand einzelner MSOs. Damit kann die Mechaniksimulation eines MSOs nicht durchgeführt werden, wenn seine entsprechende VH des Typs `MCB_Physics` nach Kapitel

9.2 zu simulieren wäre. Wäre dies in Shockwave möglich, würde auch eine ungünstig hohe Anzahl an Kommunikationsschritten zwischen MaxControl und Shockwave nötig werden, die aus Gründen der Effizienz zu vermeiden ist (siehe Kap. 10.3.2).

Die Mechaniksimulation muss in jedem Simulationsschritt daher für alle MSOs zusammen in einem Schritt durchgeführt werden, in dessen Verlauf keine VHs ausgeführt werden können. Dies schränkt die Interaktionsmöglichkeiten zwischen der Mechaniksimulation und den MSOs ein. Entweder muss die Mechaniksimulation zuerst ausgeführt werden, damit die VHs noch im aktuellen Simulationszwischen Schritt auf ihre Ergebnisse reagieren können, indem sie z.B. neue Kräfte setzen. Dann kann die Mechaniksimulation jedoch nicht mehr in demselben Simulationszwischen Schritt auf diese neuen Kräfte reagieren. Oder die Mechaniksimulation wird nach den VHs ausgeführt. Dann kann die Mechaniksimulation zwar noch in demselben Simulationszwischen Schritt auf neue Kräfte reagieren, die VHs gesetzt haben, doch die VHs können nicht mehr im aktuellen Simulationszwischen Schritt auf die Ergebnisse der Mechaniksimulation reagieren. Hier wurde die erste Möglichkeit gewählt. Wie schon Abbildung 53 aus Kapitel 9.3 (erläutert in Kap. 10.2.2) zeigt, wird also in jedem Simulationszwischen Schritt zuerst über ein Zeitintervall der Länge Δt ein Schritt der Mechaniksimulation für alle MSOs ausgeführt, der eigene Zwischenschritte enthalten kann, und erst danach folgt die Simulation der Verhaltensweisen aller SOs, inklusive der MSOs, die sich jedoch auf dasselbe Zeitintervall der Länge Δt bezieht.

Dieser Ablauf bei der Berechnung eines Simulationszwischen Schrittes wird in der folgenden Abbildung 60 illustriert.

Dabei wird das Beispiel zu „OT_{Robot}“ aus Kapitel 9.2 fortgesetzt, unter der Voraussetzung, dass sich neben einigen MSOs auch ein SO „R“ des Typs „OT_{Robot}“ in der Szene befindet und jede seiner VHs nur in ihrer `updateBeforeSubBehaviours`-Methode (siehe Kap. 8.6.3) Anweisungen ausführt, so dass die Ausführungsreihenfolge seiner VHs in jedem Simulationszwischen Schritt der Ausführungsreihenfolge seiner `startup`-Methoden entspricht, die in Kapitel 9.2 in Abbildung 50 gezeigt wird. Zur Vereinfachung wird neben der Mechanik-Simulation der MSOs nur die Abarbeitung der VHs von „R“ gezeigt. Die Abarbeitung der VHs der MSOs würde sich analog entsprechend der SO-Reihenfolge in der Objektliste in das folgende Diagramm einfügen. Dabei werden auch VHs eines MSOs in jedem Fall **nach** der Mechaniksimulation ausgeführt. Die VH „MCB_Physics“ eines MSOs führt bei ihrem Aufruf in der technischen Umsetzung keine Anweisungen aus, da sie nur SPs und Hilfsmethoden enthält und die eigentliche Mechaniksimulation der Shockwave-Komponente überlässt.

Wie in Kapitel 9.4.1 erläutert wurde, bauen alle Schritte eines Simulationszwischen Schrittes der Simulationsreihenfolge entsprechend aufeinander auf. Die gestrichelten Pfeile in der obigen Abbildung 60 illustrieren dieses Vorgehen. Wie schon bei den VHs wird also dasselbe Zeitintervall mehrfach hintereinander simuliert, zuerst durch die Mechaniksimulation, dann durch die einzelnen VHs.

Aus dem Diagramm wird ersichtlich, dass Kräfte, die von VHs gesetzt werden, erst im nächsten Simulationszwischen Schritt durch die Mechaniksimulation bearbeitet und somit wirksam werden können. Danach werden die gesetzten Kräfte, wie in Kap. 9.5 gefordert, gelöscht und müssen ggf. neu gesetzt werden.

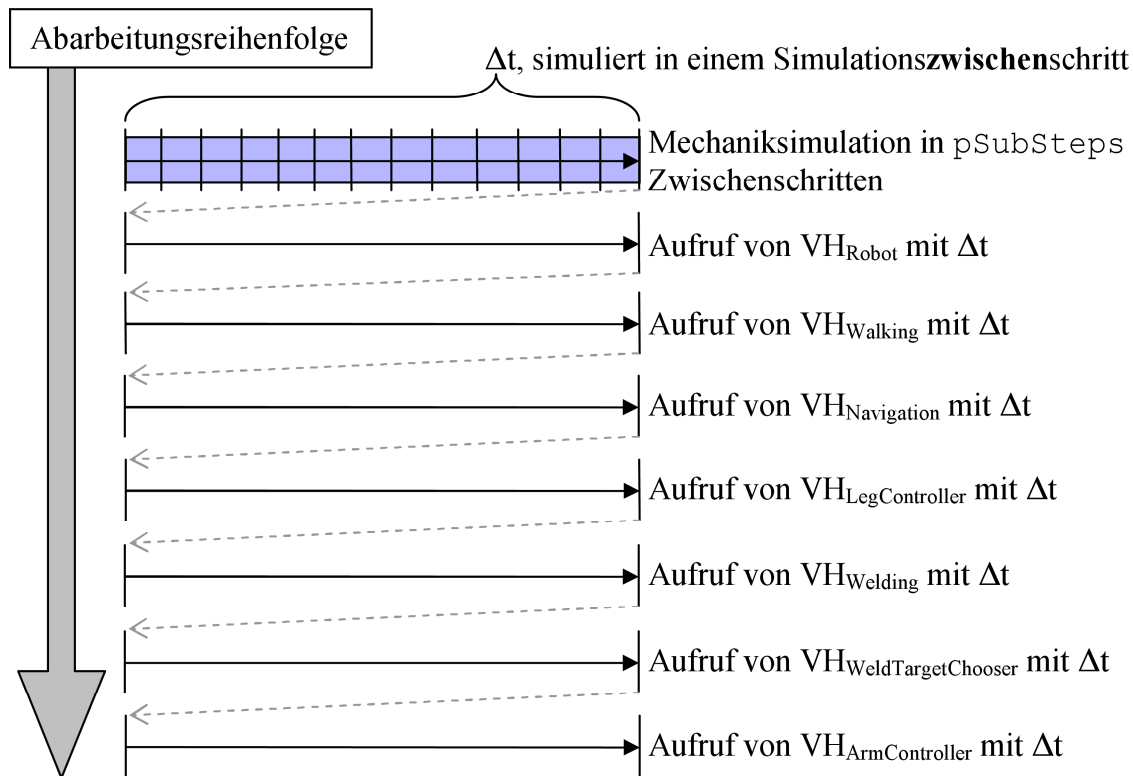


Abbildung 60: Ablauf der Mechaniksimulation in einem Simulationszwischen-schritt

10.2.3.2 Probleme und Problemlösungen bei der Simulation von Mechanik

Instabilitäten:

Das größte Problem für die Simulation von Mechanik stellen mögliche Simulationsinstabilitäten dar. Diese Problematik und entsprechende Lösungsansätze werden auch in [Jack06] erläutert. Da für komplexe mechanische Systeme in der Regel keine analytischen Lösungen existieren, werden die mechanischen Vorgänge hier durch die schrittweise Lösung von Differentialgleichungen simuliert, die das mechanische System modellieren. Die Stabilität solcher Verfahren kann stark von der verwendeten Schrittweite abhängen, also dem Zeitintervall, das in jedem Einzelschritt simuliert wird. Instabilitäten können insbesondere bei der Verwendung sehr steifer Federn in der Simulation auftreten.

Solche Probleme können durch entsprechend kleinere Schrittweiten gelöst werden. Eine sehr kleine Schrittweite erhöht jedoch entsprechend die erforderliche Anzahl an Zwischenschritten, welche die Mechaniksimulation für einen Simulationszwischen-schritt von MaxControl durchführen muss, weshalb keine konstant kleine Schrittweite verwendet werden sollte, weil dies die erforderliche Simulationszeit stark erhöhen kann.

Stattdessen bietet MaxControl verschiedene Verfahren an, die zunächst mit einer geringeren Anzahl an Zwischenschritten arbeiten und während der Simulation Instabilitäten erkennen, um einen instabilen Simulationsschritt dann mit einer erhöhten Anzahl an Zwischenschritten zu wiederholen, bis ein stabiles Simulationsergebnis erzielt wurde.

Wichtig ist bei diesen Verfahren, woran das Ergebnis eines instabilen Simulationsschrittes erkannt werden kann. Nach [Bara98] sind Instabilitäten durch die plötzlichen starken Änderungen z.B. in Federlängen erkennbar. Nach [Jack06] scheint ein simulierter Körper bei instabilem Simulationsergebnis zu „explodieren“. Nach [Brid02] und [Kacic03] sollte eine

Feder ihre Länge in einem Simulationsschritt nicht um mehr als 10% ändern, sonst sollte die Simulationsschrittweite verkleinert werden.

Aus diesen empirischen Beobachtungen heraus wurden im Wesentlichen zwei Methoden zur Stabilitätssicherung für die Mechaniksimulation in MaxControl entwickelt, da die Komponente zur Mechaniksimulation in Shockwave selbst keine solche Möglichkeit bietet und daher von außen eine entsprechende Schrittweite vorgegeben werden muss. Diese beiden Methoden werden im Folgenden kurz betrachtet, wobei nicht jede Einstellungsmöglichkeit und nicht jeder Lösungsansatz dieser Methoden behandelt werden, sondern nur die wichtigsten Grundprinzipien.

Die erste Methode wiederholt für jeden Simulationszwischen schritt den entsprechenden Schritt der Mechaniksimulation in jedem Fall mindestens einmal mit um 10% erhöhter Anzahl an Zwischenschritten und vergleicht die Ergebnisse beider Simulationsversuche miteinander. Dies geschieht unter der Annahme, dass sich zwei solcher Simulationsschritte bezüglich der Positionen und Rotationen der simulierten MSOs sehr deutlich voneinander unterscheiden, wenn einer der beiden Mechanik-Simulationsschritte instabil war. Dies liegt daran, dass bei instabilen Simulationsergebnissen sehr große Kräfte entstehen, so dass die Szene wie oben erwähnt chaotisch zu explodieren scheint. Durch die chaotische Natur dieser „Explosion“ unterscheiden sich auch zwei instabile Ergebnisse stark voneinander, weil schon die Änderung der Anzahl an Zwischenschritten um 10% zu ausreichend starken Veränderungen an einem weiterhin instabilen Simulationsergebnis führt. Weiterhin wird angenommen, dass sich zwei stabile Simulationsergebnisse mit nur um 10% veränderter Anzahl an Zwischenschritten wesentlich weniger voneinander unterscheiden. So wird eine stabile Anzahl an Zwischenschritten daran erkannt, dass ihr Ergebnis kaum vom Ergebnis einer wiederholten Simulation mit 10% mehr Zwischenschritten unterscheidet. Die dabei zugrunde gelegten Toleranzen für Abweichungen der MSOs in Position und Rotation können dieser Methode als Parameter übergeben werden.

Die zweite Methode nutzt Formen der oben genannten Kriterien für Federn und weitere empirisch bestimmte Merkmale, an denen einen instabilen Simulationsschritt direkt erkannt werden kann. Sie hat damit gegenüber der ersten Methode den Vorteil, dass sie auch einen stabilen Simulationsschritt direkt erkennt und dann keinen weiteren Versuch benötigt, während die erstgenannte Methode immer mindestens zwei Versuche durchführt, um zwei Ergebnisse vergleichen zu können. Der Nachteil dieser Methode ist jedoch, dass sie Federn als Hauptursache von Instabilitäten annimmt, während die erste Methode genereller vorgeht, ohne Ursachen für die Instabilitäten kennen zu müssen. Diese zweite Methode beobachtet alle Federn in der Mechaniksimulation. Wurde eine Feder in einem Mechaniksimulationsschritt im Vergleich zum Vorzustand über eine Toleranzschwelle hinaus gedehnt oder gestaucht, so wird die Simulation mit 10% mehr Zwischenschritten wiederholt, bis ein stabiles Simulationsergebnis vorliegt. Da sich zu hohe Federkräfte auch auf andere MSOs durch entsprechend hohe Stoßkräfte auswirken können, ohne dass sich dabei Federn entsprechend stark in ihrer Länge verändern, werden ebenfalls die lineare Beschleunigung und die Winkelbeschleunigung aller MSOs beobachtet. Überschreiten diese für ein MSO einen bestimmten Wert, so gilt das Simulationsergebnis ebenfalls als instabil. Damit ist diese Methode nicht mehr gänzlich vom Verhalten der Federn abhängig, sondern bezieht auch die empirischen Überlegungen aus der ersten Methode mit ein, die instabile Simulationsergebnisse am explosionsartigen Verhalten der MSOs erkennt, was sich in entsprechend hohen Beschleunigungen zeigt. Einstellbare Parameter dieser Methode sind die maximal erlaubte Federlängenänderungen und Toleranzschwellen für lineare Beschleunigungen und Winkelbeschleunigungen.

Insbesondere die zweite Methode kann jedoch eine Situation falsch beurteilen, in der hohe Federlängenänderungen und Beschleunigungen stabile Simulationsergebnisse darstellen, weil beispielsweise große Massen mit hohen Geschwindigkeiten aufeinandertreffen. Daher haben beide Methoden weitere Parameter gemeinsam, die sie berücksichtigen. Insbesondere wird eine maximale Anzahl an Zwischenschritten angegeben, die nicht überschritten wird. Diese Zahl wird auch daher angegeben, weil die Mechaniksimulation in Shockwave bei einer zu hohen Anzahl an Zwischenschritten auch inkorrektes Verhalten zeigte, indem Festkörper z.B. einander durchdringen und dann ineinander stecken bleiben. Zusätzlich gibt es auch ein einstellbares Minimum an Zwischenschritten, das nicht unterschritten wird. Dies kann sich günstig auf die Simulationszeit auswirken, damit nicht unnötig viele Simulationsversuche mit bei Weitem zu wenigen Zwischenschritten durchgeführt werden. Führt ein Mechanik-Simulationsschritt zu einem instabilen Ergebnis, wird die Anzahl an Zwischenschritten von beiden Methoden erhöht, bis ein stabiles Ergebnis gefunden wurde. Danach wird diese Anzahl nach jedem stabilen Simulationsschritt wieder herabgesetzt, bis entweder das Minimum erreicht oder ein instabiles Ergebnis erhalten wurde. Es zeigte sich jedoch, dass es während der Simulation oft Phasen gibt, in denen über längere Zeit eine erhöhte Anzahl an Zwischenschritten benötigt wird. Damit in solchen Phasen diese benötigte Anzahl nicht ständig unterschritten wird, was zu unnötigen erneuten Simulationsversuchen im instabilen Fall führt, wurden die Parameter `stepsToIncreaseMin` und `stepsToDecreaseMin` eingeführt. Wurde für `stepsToIncreaseMin` Mechanik-Simulationsschritte das bisher genutzte Minimum an Zwischenschritten nicht mehr erfolgreich mit einem stabilen Simulationsergebnis genutzt, so wird das während dieser Schritte erfolgreich genutzte Minimum an Zwischenschritten als neues Minimum festgelegt. Wurde dagegen für `stepsToDecreaseMin` Mechanik-Simulationsschritte ausschließlich dieses neue Minimum erfolgreich genutzt, so kann davon ausgegangen werden, dass die „kritische Phase“ vorüber ist und das Minimum wird wieder um 10% herabgesetzt, sofern dabei nicht das ursprünglich als Parameter angegebene Minimum unterschritten wird.

Insgesamt ist es durch diese Methoden möglich, eine stabile und dennoch zeiteffiziente Simulation durchzuführen, auch wenn steife Federn verwendet werden. Dies ist besonders wichtig, weil steife Federn zur Imitation von Constraints wie Drehgelenken verwendet werden mussten, da die Mechaniksimulationskomponente von Shockwave keine solchen Constraints unterstützt (siehe dazu Kap. 10.2.3.3).

Deaktivierung ruhender Körper:

Die Erkennung und Behandlung von Kollisionen zwischen MSOs stellt eines der größten Probleme für die Mechaniksimulation dar (siehe [Jack06]), weshalb es insbesondere für die Effizienz der Simulation wichtig ist, möglichst viele auf Kollision zu testende MSO-Paare schnell auszuschließen. Denn generell muss jedes MSO auf eine Kollision mit jedem anderen MSO getestet werden. Insbesondere bei aufeinander gestapelten MSOs treten ständig wechselseitige Kollisionen auf, auch wenn diese nicht zur Bewegung der MSOs beitragen, sondern im Gegenteil dazu führen, dass diese bewegungslos wie in den folgenden Beispielen in Abbildung 61 und Abbildung 62 aufeinander ruhen.



Abbildung 61: Gestapelte Kartons

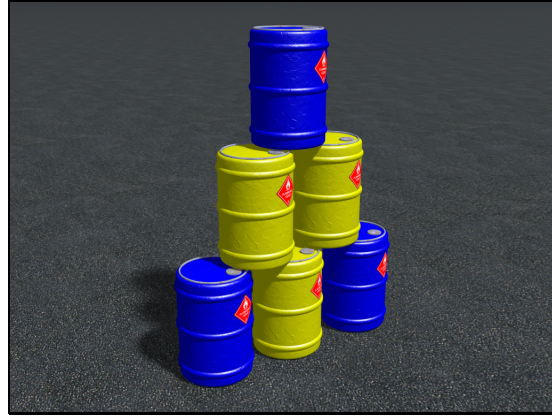


Abbildung 62: Gestapelte Tonnen

Eine Maßnahme zur Steigerung der Effizienz ist die vorübergehende Abschaltung der Mechaniksimulation für solche ruhenden Objekte. Ist ein MSO auf diese Weise deaktiviert, wird es nicht länger auf die Kollision mit anderen ebenfalls deaktivierten MSOs getestet und seine Bewegungsgleichungen werden nicht länger gelöst. Dieser Zustand bleibt erhalten, bis das MSO von einem aktiven MSO angestoßen oder durch eine externe Kraft bewegt wird, die neben direkt gesetzten Kräften auch von einer Feder stammen kann.

Shockwave unterstützt dies zwar grundsätzlich, das entsprechende System neigte jedoch dazu, MSOs zu deaktivieren, die sich sichtbar noch bewegten. Deshalb wird dieses automatische System abgeschaltet und extern durch MaxControl gesteuert, indem MSOs gezielt deaktiviert werden, sobald ihre lineare Geschwindigkeit und ihre Winkelgeschwindigkeit jeweils als Parameter wählbare Werte unterschreiten.

Shockwave reaktiviert ein deaktiviertes MSO automatisch, sobald es wie oben erläutert durch einen Stoß oder eine Kraft wieder bewegt wird. Dadurch ist es nicht trivial, einen stabilen Objektstapel wie in Abbildung 61 und Abbildung 62 bestehend aus deaktivierten MSOs zu erreichen. Werden einige MSOs in einem Stapel deaktiviert, stoßen die noch nicht deaktivierten MSOs dieses Stapels die deaktivierten MSOs im nächsten Mechaniksimulationsschritt sofort wieder an, was zu einer Kettenreaktion führt, die alle MSOs im Stapel wieder reaktiviert. Zu deaktivierende MSOs werden von MaxControl daher nach jedem Mechaniksimulationsschritt erneut deaktiviert, so dass alle MSOs eines Stapels zusammen deaktiviert werden, sobald sie alle gemeinsam die entsprechenden Kriterien erfüllen. Dann stößt keines dieser MSOs mehr die anderen MSOs des Stapels an, alle MSOs dieses Stapels bleiben deaktiviert und tragen damit erheblich weniger zum Zeitbedarf der folgenden Mechaniksimulationsschritte bei, bis der Stapel von außen wieder bewegt wird. Deaktivierte MSOs müssen lediglich auf Kollision mit anderen noch aktiven MSOs und auf externe Kräfte getestet werden, die Tests auf die zahlreichen wechselseitigen Kollisionen innerhalb des Stapels und die Lösung der Bewegungsgleichungen dieser MSOs entfallen.

Auf diese Weise können relativ viele gestapelte Objekte in einer Szene aufgestellt werden, die erst aktiv werden, wenn sie von anderen MSOs angestoßen werden, ohne erheblich mehr Simulationszeit zu benötigen. Dies ist für realistische Szenen wichtig, da sich in der Realität gerade in natürlichen aber auch in von Menschen gestalteten Umgebungen häufig sehr viele ruhende Gegenstände befinden, die erst bei Kollisionen mit bewegten Objekten ihre Ruhelagen verändern. Beispiele sind herumliegende Steine oder Stapel von Gegenständen wie Kisten, Tonnen, Rohre oder Holzstapel.

Winkelsprung-Problem bei Motion-Blur:

Die Mechanik-Simulation und zahlreiche Funktionen für den Umgang mit Transformationen (siehe Kap. 10.1.3) und insbesondere Rotationen liefern Winkel ausschließlich aus dem Intervallen $[-180^\circ, 180^\circ]$ zurück, auch wenn sich ein 3D-Objekt bereits mehrfach um eine Achse gedreht hat. Werden so eingeschränkte Winkel für die Ausrichtung eines 3D-Objektes im Raum verwendet und entsprechend in der Szenenzustandsfolge gespeichert, so führt dies bei einfachem Rendering der Filmsequenz von Frame zu Frame noch zu keinen Problemen. Denn zu jedem Winkel x gehört die Äquivalenzklasse $\{w \mid x + 360^\circ \cdot i, i \in \mathbb{Z}_0\}$, die zu x optisch äquivalente Winkel enthält. Das bedeutet, es macht für die Erscheinung eines 3D-Objektes keinen Unterschied, ob es z.B. um die X-Achse des Weltkoordinatensystems um 65° , 425° oder -295° Grad rotiert dargestellt wird.

Probleme entstehen in diesem Zusammenhang erst bei dem Einsatz von so genanntem „Motion-Blur“ (Bewegungsunschärfe), das zur Verminderung von „Temporal-Aliasing“-Effekten verwendet werden kann, die bei der Darstellung schneller Bewegungen, insbesondere schneller Rotationen auftreten (siehe [Steph03], genauere Erläuterung des Temporal-Aliasing siehe [Foley97]). Ein sich schnell drehendes Rad scheint beispielsweise ohne Motion-Blur bei bestimmten Winkelgeschwindigkeiten rückwärts zu laufen, obwohl es sich vorwärts dreht. Motion-Blur simuliert nun die Belichtungszeit einer Filmkamera, indem mehrere Bilder für einen Frame berechnet und miteinander gemischt werden. Diese Bilder entstehen durch die Interpolation der Positions- und Rotationswerte der 3D-Objekte zwischen den Frames. Die folgenden Bilder Abbildung 63 und Abbildung 64 zeigen den Unterschied zwischen der Darstellung desselben schnell rotierenden Autoreifens mit und ohne Motion-Blur.



Abbildung 63: Reifen ohne Motion-Blur



Abbildung 64: Reifen mit Motion-Blur

Rotiert nun beispielsweise ein als MSO simulierter Autoreifen zwischen zwei Frames im Uhrzeigersinn von 180° auf 181° , so wird die Mechaniksimulation die neue Rotation bei entsprechendem Werteintervall $[-180^\circ, 180^\circ]$ als -179° zurückliefern. Würde diese neue Rotation unverändert so in der Szenenzustandsfolge gespeichert werden, würde das Motion-Blur zwischen beiden Frames nicht langsam im Uhrzeigersinn von 180° auf 181° rotieren sondern schnell gegen den Uhrzeigersinn von 180° auf -179° . Ein spezieller Controller „Smooth Rotation“ für Rotations-Tracks in 3D-Studio-MAX kann hier helfen, da er in solchen Fällen aus der Äquivalenzklasse des zweiten Winkels immer den zum ersten Winkel nächsten Winkel auswählt. Im Beispiel würde er zum ersten Winkel 180° als zweiten Winkel nicht -179° sondern $(-179^\circ + 360^\circ) = 181^\circ$ wählen.

Doch auch diese Lösung ist nicht korrekt für sehr hohe Winkelgeschwindigkeiten von über 180° pro Zeitintervall zwischen zwei Frames, die z.B. bei Autoreifen wie in Abbildung 64 oder den Rotoren von Düsentriebwerken durchaus vorkommen können. Bei einer Winkelgeschwindigkeit im Uhrzeigersinn von $185^\circ/\text{Frame}$ würde der „Smooth Rotation“-Controller ausgehend von einem Anfangswinkel 0° nicht im Uhrzeigersinn zu 185° interpolieren, sondern gegen den Uhrzeigersinn zu -175° . Bei weiter steigender Geschwindigkeit würde sich das Objekt scheinbar immer langsamer drehen und bei $360^\circ/\text{Frame}$ schließlich zum Stillstand kommen.

Um dies zu vermeiden nutzt MaxControl die Daten über die tatsächliche Winkelgeschwindigkeit eines MSOs, welche die Mechaniksimulation ebenfalls zurückliefert. Dazu werden zwei Schritte durchgeführt. Zunächst wird anhand der aktuell von der Mechaniksimulation gelieferten Winkelgeschwindigkeit und der Rotation des MSOs aus dem vorangehenden Mechaniksimulationsschritt die neue Rotation P des MSOs geschätzt, unter der Annahme, dass diese Winkelgeschwindigkeit während des gesamten aktuellen Mechaniksimulationsschrittes konstant war. Diese Rotation P wird nun mit der tatsächlich zurückgelieferten Rotation X verglichen. Aus der Äquivalenzklasse von X wird die Rotation Y gewählt, die der geschätzten Rotation P am nächsten ist. Diese Rotation Y wird in der Szenenzustandsfolge gespeichert, wenn der zugehörige Simulationszwischen-schritt der letzte aus dem aktuellen Simulationsschritt war. So bleibt die exakte Rotation aus der Mechaniksimulation erhalten, jedoch unter Berücksichtigung der tatsächlichen Winkelgeschwindigkeit.

Wie Abbildung 64 zeigt, ist so realistisches Motion-Blur für MSOs mit beliebigen Winkelgeschwindigkeiten möglich.

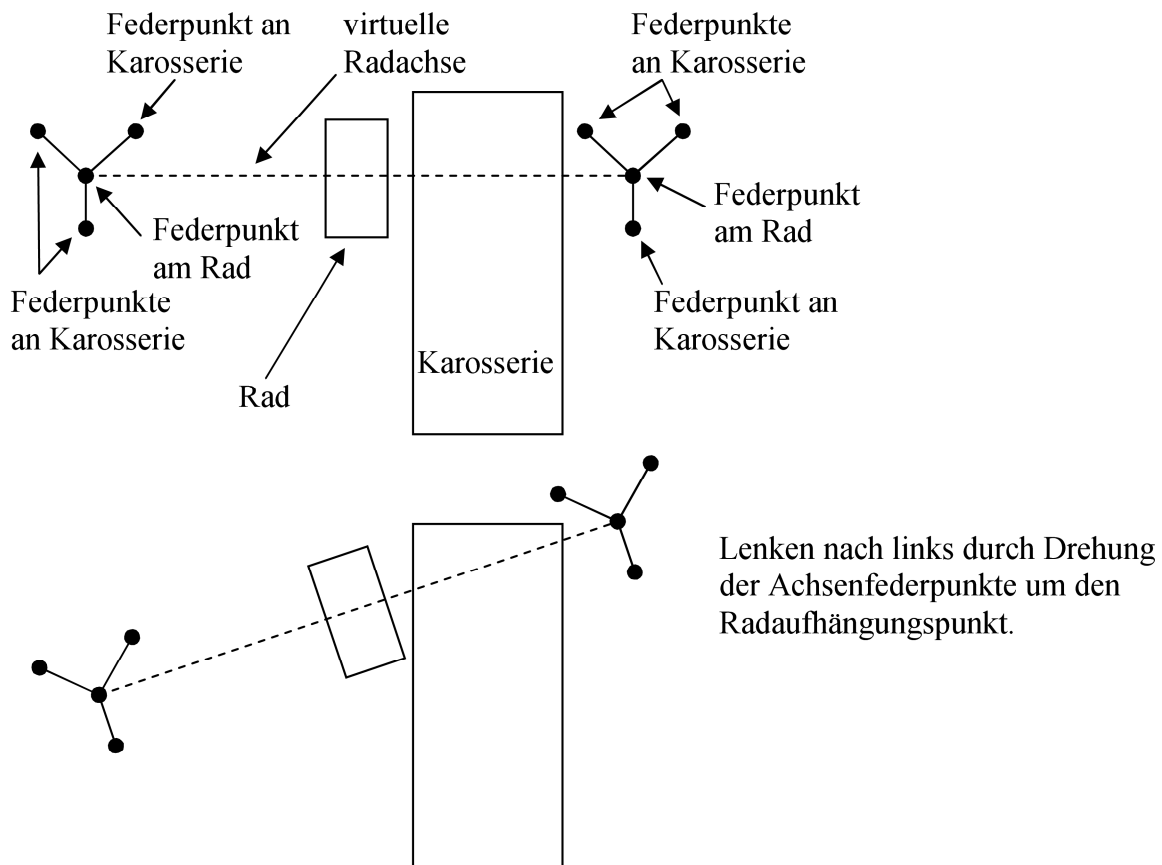
10.2.3.3 Simulation von Fahrzeugphysik

Um Fahrzeuge realistisch simulieren zu können, war eine Nachbildung von Radachsen mit gefederter Einzelradaufhängung erforderlich. Shockwave unterstützt jedoch keine Gelenk-Constraints, so dass Radachsen eigentlich nicht simulierbar sind.

Eine solche gefederte Radachse wurde jedoch über relativ steife Federn nachgebildet. Dies führt einerseits zu den in Kap. 10.2.3.2 angesprochenen Instabilitäten, andererseits können Federn nie so steif sein, dass sie absolut starre Achsen nachbilden. Doch auch das System aus [Kacic03] betrachtet dieses Vorgehen als praktikabel und erzeugt alle Constraints ausschließlich über Federn.

Jedes Rad des Typs `MCD_CarTyre` erhält eine eigene Radachse, die aus Federn aufgebaut ist. Für eine gefederte Radachse muss die Besonderheit umgesetzt werden, dass sie nicht nur eine freie Drehung des Rades um diese Achse ermöglicht, sondern dass jedes Rad einzeln nach oben und nach unten federn kann, ohne jedoch seitlich oder vor und zurück zu federn. Die Achse selbst muss auch möglichst starr mit der Karosserie verbunden sein und darf nicht kippen, ausgenommen das Fahrzeug lenkt die Vorderräder durch eine entsprechende Drehung der vorderen Radachsen. Um diese Vorgaben zu erfüllen wurde die Möglichkeit genutzt, die Federpunkte weit außerhalb des Volumens der beteiligten MSOs zu setzen (siehe Kap. 10.1.4). Die folgende Abbildung 65 zeigt den Aufbau und die Funktionsweise des verwendeten Federsystems für ein Rad:

Draufsicht:



Vorderansicht:

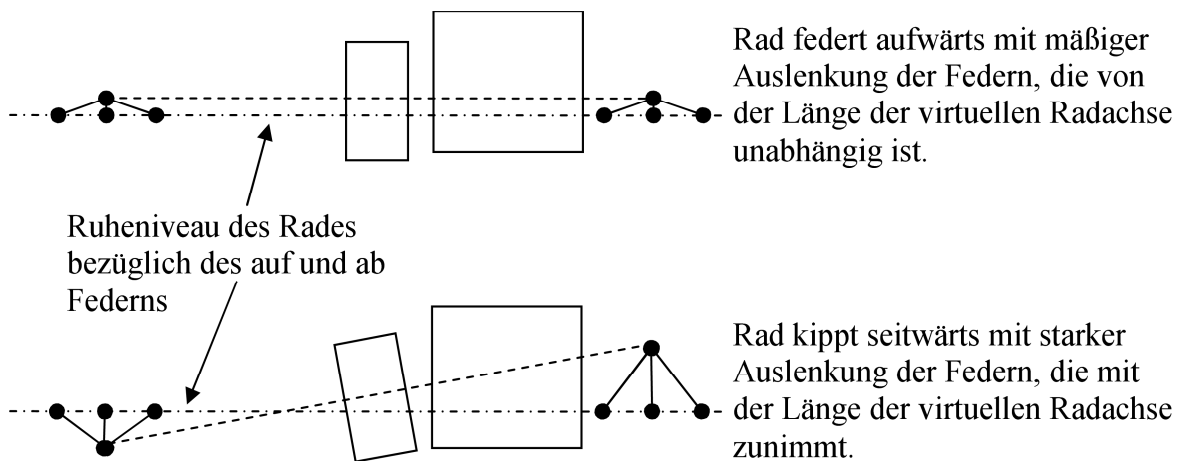


Abbildung 65: Radachsen durch Federn

Die obige Abbildung 65 zeigt, dass ein Rad auf jeder Seite über 3 Federn mit der Karosserie verbunden ist. Um das Rad zum Lenken zu schwenken, werden die Federpunkte entsprechend um den Radaufhängungspunkt rotiert, so dass das Rad von den Federn in die entsprechende Stellung gezogen wird.

Durch die mit einem Trampolin vergleichbare Aufhängung an drei Punkten in einer Ebene auf beiden Seiten kann das Rad nach oben und nach unten federn, aber nur schwer vor, zurück

und zur Seite. Dies wird durch die Verwendung sehr steifer Federn erreicht, die beim Federn des Rades nach oben und nach unten weniger stark gedehnt werden als bei Bewegungen in die anderen Richtungen.

Dadurch könnte das Rad jedoch wie unten in Abbildung 65 gezeigt seitlich kippen, da hier eine Auslenkung der Federn wie beim Bewegen des Rades nach oben und nach unten stattfindet, jedoch in zwei entgegengesetzte Richtungen. Während im letztgenannten Fall die Länge der virtuellen Achse, die zwischen den am Rad befestigten Federpunkten liegt, keine Auswirkung auf die Auslenkung der Federn hat, verursacht die Rotation dieser virtuellen Achse beim Kippen eine um so stärkere Auslenkung nach oben bzw. unten auf beiden Seiten der Achse, je länger diese virtuelle Achse ist. Daher wurde diese Achse erheblich länger gewählt, als sie in Abbildung 65 dargestellt wird, um dieses Kippen zu verhindern.

Insgesamt konnte so eine relativ realistische Federung der Räder eines Fahrzeugs erreicht werden, über die eine auf Mechanik basierende Simulation von sich selbst steuernden Fahrzeugen in einem Autorennen möglich wurde (siehe Kap. 10.4.1.5).

10.2.4 Automatische Erzeugung von Tönen

In Kapitel 5.3 wurde bereits erläutert, dass Foley Studio MAX für die automatische Erzeugung von Toneffekten verwendet wird. Dieses Plugin erlaubt es insbesondere, in einer Szene Tonemitter-Objekte zu erstellen, die neben anderen Parametern eine Position im Raum haben, jedoch nicht optisch im Rendering der Szene erscheinen. Stattdessen können die von solchen Objekten erzeugten Töne „gerendert“ werden, wodurch eine Tonspur erzeugt wird, die mit dem aus derselben Szene durch visuelles Rendering erstellten Film kombiniert werden kann. Indem MaxControl diese Objekte steuert, kann es auch automatisch passende Toneffekte in die entstehenden Filme einbauen.

Dazu wurde der Objekttyp `MCD_FoleySound` entwickelt, der mit Foley Studio MAX erstellte Tonemitter steuern kann, indem er ihnen als OT zugewiesen wird und über entsprechende Standard-SPs die Parameter dieser Emitter animiert. Dies sind die gesteuerten Standard-SPs:

```
public final MCP_BooleanStandardProperty SFXtrigger=  
    new MCP_BooleanStandardProperty("Trigger", this);  
public final MCP_DoubleStandardProperty SFXvolume=  
    new MCP_DoubleStandardProperty("emm_Volume", this);  
public final MCP_DoubleStandardProperty SFXpitch=  
    new MCP_DoubleStandardProperty("emm_Pitch", this);
```

Dabei sei daran erinnert, dass die Bezeichnungen der zugehörigen ursprünglichen Objekteigenschaften für Standard-SPs im Konstruktor angegeben werden (Kap. 8.2.1), so dass die Standard-SP „`SFXtrigger`“ beispielsweise die Emitter-Eigenschaft „`Trigger`“ steuert.

Jedem Emitter kann eine Tondatei zugewiesen werden. Diese wird abgespielt, so lange die boolesche SP `SFXtrigger` den Wert `true` hat. Tonemitter haben noch weitere Parameter, die `MCD_FoleySound` nicht zugänglich gemacht wurden. So bestimmt einer dieser Parameter, ob ein abgespielter Toneffekt sofort verstummt, sobald `SFXtrigger` den Wert `false` erhält, oder ob er auf einer eigenen Tonspur weiterläuft, auch wenn durch erneutes Setzen von `SFXtrigger` auf `true` dieser Toneffekt bereits ein weiteres Mal auf einer weiteren Tonspur gestartet wurde. Ebenso können eine Wiederholung des Toneffektes, seine

Reichweite oder der Dopplereffekt für diese Tonquelle eingestellt werden. Dies sind nur einige Beispiele für die zahlreichen Einstellungsmöglichkeiten. Die SP „SFXvolume“ kontrolliert die Lautstärke des Toneffektes und „SFXpitch“ steuert die Tonhöhe.

Über solche Emittierer wurden die meisten Filme aus Kap. 10.4.1 automatisch mit Toneffekten versehen, ohne dass eine manuelle Nachvertonung nötig wurde. Dazu wurden auch erweiterte Unterklassen von `MCD_FoleySound` eingesetzt, die insbesondere Daten der Mechaniksimulation zur Tonerzeugung verwenden. Auf diese wird weiter unten näher eingegangen.

Der OT `MCD_FoleySound` hat zwei wichtige Unter-VHs, `MCB_DTsoundPitchController` und `MCB_SoundAverager`, die bereits in Kapitel 9.4.5 ab Seite 182 erläutert wurden.

Eine wichtige Unterklasse von `MCD_FoleySound` ist die Klasse `MCD_ContactSFX`, welche Toneffekte bei Kollisionen von MSOs ermöglicht. Dabei verarbeitet ein SO des Objekttyps `MCD_ContactSFX` automatisch die Kollisionen seines Eltern-MSOs und reagiert mit entsprechenden Toneffekten darauf. Unterklassen von `MCD_ContactSFX` sind `MCD_ImpactSFX`, `MCD_SkiddingSFX` und `MCD_RollingSFX`, die jeweils auf Stöße, Rutschen bzw. Rollen des MSOs reagieren. Dabei kann auch nach bestimmten Kontaktmaterialien gefiltert werden, so dass ein Tonemitter des Typs `MCD_ContactSFX` nur auf Kontakt mit Holz, ein anderer nur auf Kontakt mit Asphalt reagiert. Werden solche Tonemitter einem MSO als Kindobjekte hinzugefügt, werden für dieses MSO automatisch Geräusche für die oben genannten Kontakt Ereignisse erzeugt, die sich je nach Art des Kontakt Ereignisses und je nach Kontaktmaterial unterscheiden können. Diese Möglichkeit wurde insbesondere bei den Autorennsimulationen aus Kap. 10.4.1.5 genutzt, so dass jedes MSO Kontaktgeräusche erzeugt, jeder Reifen, jeder Karosserie, im Falle eines Stoßes sowie beim Umfallen oder Rollen auch jeder auf der Strecke herumliegende Gegenstand.

10.3 Einbindung der verwendeten Werkzeuge

10.3.1 Kommunikation mit 3D-Studio-MAX und ihre Optimierung

10.3.1.1 Prinzipien der Optimierung

Für die folgenden Betrachtungen setzen wir voraus, dass in der vorläufigen Szenenzustandsfolge jeweils Key-Techniken verwendet werden. Damit sind im Allgemeinen zu Beginn der Simulation bereits Folgen von Keys in den Tracks vorhanden, welche die vorläufige Szenenzustandsfolge repräsentieren (siehe Kap. 6.5).

Wie in den Kapiteln 6.5 und 6.7 erläutert, werden bei der Simulation einer Szene SPs von 3D-Objekten von Frame zu Frame durch die Simulationsschritte verändert. Diese Veränderungen müssen aus dem jeweiligen MaxControl-Zustand durch dynamisch generierte MAXScript-Befehle an den zugehörigen Szenenzustand in 3D-Studio-MAX gesendet werden. Ein trivialer Ansatz zur Realisierung dieser Kommunikation würde nach jedem Simulationsschritt Skript-Befehle erzeugen, welche dann direkt die aktuellen Werte der SPs in den zum Simulationsschritt gehörigen endgültigen Szenenzustand schreiben würden.

Einige SPs können für einen Simulationsschritt als manuell kontrolliert festgelegt worden sein. Die Werte solcher SPs müssen für diesen Simulationsschritt aus dem aktuellen vorläufigen Szenenzustand in den entsprechenden MaxControl-Zustand eingelesen werden, um dort im aktuellen Simulationsschritt berücksichtigt zu werden. Ein trivialer Ansatz könnte

vor jedem Simulationsschritt zu einem Frame n aus dem zugehörigen Szenenzustand δ_n die Werte der SPs sowie die Werte der zugehörigen MPs auslesen. Die MPs würden dann bestimmen, ob die Werte der zugehörigen SPs im aktuellen Frame n als manuell kontrolliert gelten und damit beim aktuellen Simulationsschritt berücksichtigt werden sollen oder ob sie durch die Ergebnisse dieses Simulationsschrittes überschrieben werden können. Die Werte der SPs würden also auch dann gelesen, wenn ihre MP den Wert false hat, obwohl der Wert der SP in diesem Fall nicht benötigt wird. Auch für diese Kommunikation müssten dynamisch MAXScript-Befehle erzeugt werden, welche diese Werte der SPs und MPs aus 3D-Studio-MAX auslesen und sie in eine Datei schreiben, die dann von MaxControl gelesen werden könnte.

Die Ausführung dieser Skripte für das Lesen und Schreiben von Werten würde im Allgemeinen unnötig lange dauern. Es wurden zwei Methoden zur Optimierung der Kommunikation entwickelt. Die erste Methode „**CommOptMPs**“ beruht auf effizient eingesetzten MP-Werten, die zweite Methode „**CommOptKeys**“ auf Nutzung der Key-Techniken. Keys und die darauf basierende Technik der Keyframe-Animation wurden in Kapitel 3.5 erläutert. Da Key-Techniken für fast alle SPs verwendet werden, ist CommOptKeys die wichtigste Optimierungsmethode. CommOptMPs kann für die Implementierung von SP-Klassen genutzt werden, wenn eine Nutzung von Key-Techniken für den betreffenden SP-Typ nicht oder nur schwer möglich ist.

Zunächst wird auf das **Lesen** von Werten eingegangen: Bei der oben beschriebenen trivialen Lösung werden für jeden Simulationsschritt zu Frame n zuerst die Werte aller SPs und MPs aus δ_n gelesen. CommOptMPs liest die Werte einer SP nur dann aus dem Szenenzustand, wenn ihre MP den Wert true hat. Um dies zu ermöglichen, werden bei dieser Methode die Werte der MPs immer einen Frame im Voraus gelesen und neben den aktuellen MP-Werten zwischengespeichert. Beim Simulationsschritt zu Frame n werden also bereits die MP-Werte aus δ_{n+1} gelesen. So ist beim folgenden Simulationsschritt zu Frame $n+1$ bereits bekannt, welche SP-Werte aus δ_{n+1} gelesen werden müssen, weil ihre MP in Frame $n+1$ jeweils den Wert true hat. So kann das Lesen nicht benötigter SP-Werte vermieden werden.

Dieses Verfahren wird durch den in Kapitel 6.6 erwähnten vordefinierten Werteverlauf für MPs von nicht-animierbaren SPs unterstützt. Wie bei allen SPs (siehe Kap. 9.3 ab S. 170) ist auch bei nicht animierbaren SPs die MP im ersten Frame a des Simulationsintervalls auf „true“ festgelegt, damit der Szenenzustand aus Frame a vollständig in den MaxControl-Zustand gelesen wird, auf dem die Simulation beginnt. Dies kann auch durch den Benutzer nicht geändert werden. In allen folgenden Frames ist die MP jeder nicht-animierbaren SP dann als „false“ festgelegt. Damit kann der Wert der SP nicht durch manuell definierte Animationen verändert werden, die für eine nicht-animierbare SP in der Szenenzustandsfolge ohnehin nicht festgelegt werden können. Ebenso ist eine Änderung ihres Wertes durch die Simulation für eine nicht-animierbare SP in MaxControl durch entsprechende Mechanismen grundsätzlich ausgeschlossen. Der Vorteil des so festgelegten Werteverlaufs für MPs von nicht-animierbaren SPs liegt für CommOptMPs darin, dass dieses Verfahren dann auch nur aus Frame a der Szenenzustandsfolge den Anfangswert einer solchen SP liest, aber weitere Lesevorgänge für diese SP einspart, da ihre MP in den folgenden Frames den Wert „false“ hat. Somit handelt CommOptMPs für diese SPs nicht nur korrekt, sondern durch die Einsparung unnötiger Leseschritte auch effizient.

Das Verfahren CommOptKeys, das nur für animierbare SPs eingesetzt wird, liest zwar aus technischen Gründen immer die Werte aller SPs, auch wenn diese im aktuellen Frame nicht manuell kontrolliert sind, jedoch lohnt sich der so entstehende Mehraufwand im Vergleich zu

„CommOptMPs“, weil dadurch die Key-Techniken effizient genutzt werden können. Dies liegt daran, dass, z.B. bei linearer Interpolation, zwei Keys die Werte einer SP in einem beliebig großen Intervall bestimmen können. Beim Lesen von Werten dieser SP müssen nicht alle Werte aus dem betrachteten Intervall gelesen werden, sondern nur die beiden Keys. Die Werte zwischen diesen Keys können intern in MaxControl durch Interpolation berechnet werden, ohne dabei erneut auf 3D-Studio-MAX zugreifen zu müssen. Auch die Werte vor dem ersten bzw. nach dem letzten Key eines Animations-Tracks können in MaxControl ohne weitere Kommunikation bestimmt werden, sobald der erste oder der letzte Key bekannt sind, da die Interpolatoren in 3D-Studio-MAX auf Extrapolation durch konstante Fortsetzung eingestellt werden.

Auch das **Schreiben** der Simulationsergebnisse in die Szenenzustandsfolge würde bei der oben genannten trivialen Lösung unnötige Kommunikationsbefehle erzeugen, wodurch die Ausführung dieser Befehle entsprechend länger als erforderlich dauern würde. Nach dem Simulationsschritt zu einem Frame n werden bei der trivialen Lösung die in σ_n gespeicherten Simulationsergebnisse durch dynamisch generierte Skripte in denselben aktuellen Szenenzustand δ_n geschrieben, aus dem vorher die Werte der MPs und SPs gelesen wurden, wodurch der endgültige Szenenzustand δ_n' entsteht. Die Werte werden auch dann in δ_n geschrieben, wenn sich durch den Simulationsschritt für einige SPs gar kein anderer als der vorher in δ_n gültige Wert ergeben hat. Dies ist insbesondere bei SPs der Fall, die im aktuellen Frame als manuell kontrolliert gelten und deren Wert somit durch den Simulationsschritt im Vergleich zum vorher gelesenen Wert nicht verändert worden sein kann. CommOptMPs ist nur beim Lesen optimiert und geht beim Schreiben genau wie die hier erläuterte triviale Lösung vor.

CommOptKeys nutzt dagegen auch beim Schreiben von SP-Werten Key-Techniken. Wenn neu zu schreibende Werte durch Interpolation zwischen Keys ausgedrückt werden können, werden so Befehle eingespart. Bei einem Schreiben der neuen Werte in die ohne Ausnutzung von Interpolationstechniken würde dagegen für jede SP in jedem Frame ein neuer Animations-Key erstellt, um so den jeweiligen Wert in die Szenenzustandsfolge zu schreiben. Dies hätte neben einer unnötig hohen Zahl von Schreibbefehlen auch zur Folge, dass diese Keys einen unnötig hohen Teil des Speichers belegen würden. Dies würde die Ausführungsgeschwindigkeit und -stabilität von 3D-Studio-MAX senken und die Szenendateien unnötig groß werden lassen.

Außerdem ist eine möglichst geringe Anzahl von Keys meistens auch vorteilhaft, wenn ein manuelles Nacheditieren der Werteverläufe einer SP nötig ist, wie das folgende Beispiel zeigt:

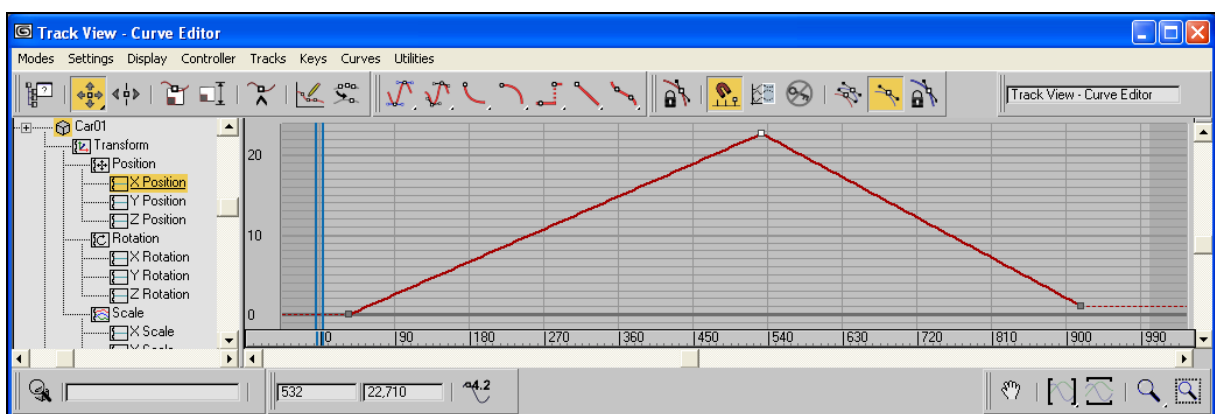


Abbildung 66: Track: Abzuflachende Kurve mit 3 Keys

Möchte man diese Kurve abflachen, so muss nur die Position des mittleren Keys verändert werden:

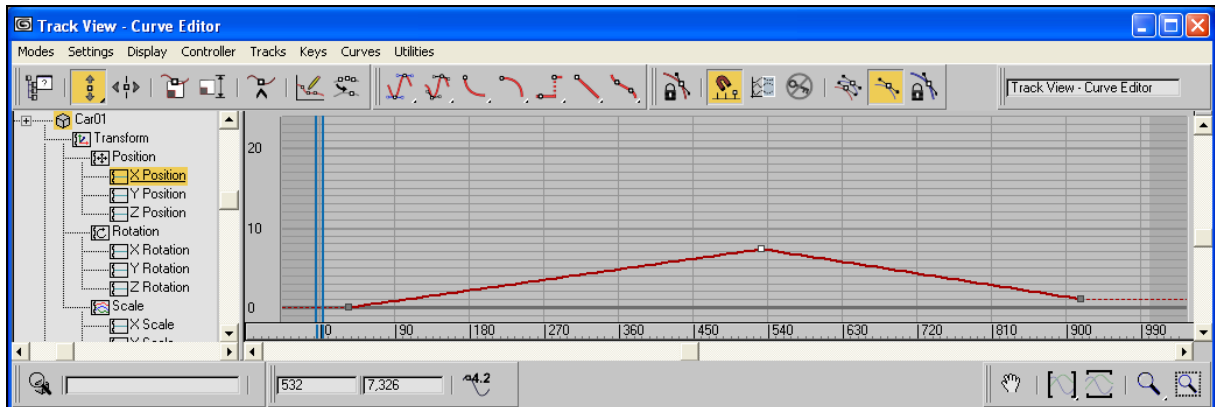


Abbildung 67: Track: Abgeflachte Kurve mit 3 Keys

Wird die ursprüngliche Kurve allerdings durch Keys festgelegt, von denen mehrere redundant sind, fällt dieser Editierungsschritt ungleich schwerer:



Abbildung 68: Track: Abzuflachende Kurve mit redundanten Keys

Will man diese Kurve in ähnlicher Weise wie oben abflachen, reicht das Verschieben eines Keys nicht mehr aus. Jetzt müssen alle Keys außer dem letzten und dem ersten verändert werden:

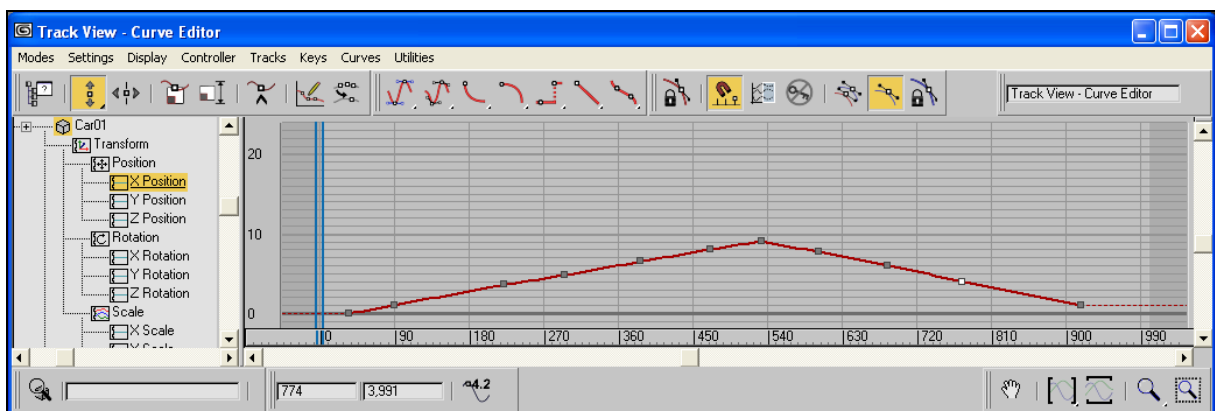


Abbildung 69: Track: Abgeflachte Kurve mit redundanten Keys

MAXScript selbst bietet ebenfalls eine Funktion zum automatischen Eliminieren unnötiger Keys an, die so genannte Key-Reduktion. Als Beispiel für die Anwendung des entsprechenden MAXScript-Befehls „reduceKeys“ können Abbildung 68 und Abbildung

66 dienen. Wird diese Anweisung auf den Track aus Abbildung 68 angewendet, werden die redundanten Keys entfernt und es ergibt sich der Track aus Abbildung 66. Dabei werden lineare Zusammenhänge erkannt und für die Key-Reduktion genutzt.

Sich allein auf diese Key-Reduktion zu beschränken würde keine Kommunikationsoptimierung ermöglichen, da diese Reduktion ein nachträglicher Vorgang ist, der unnötige Keys erst nach deren Erstellung erkennen und entfernen kann. Ferner wird die Key-Reduktion beschleunigt, wenn sich bereits vor dem Ausführen dieser Funktion eine möglichst geringe Anzahl von Keys in den Tracks der SPs befindet. Da dieses Verfahren auch dann noch sehr lange dauern kann, versucht MaxControl zunächst ohne die Key-Reduktion möglichst wenig Keys zu erzeugen. Eine Key-Reduktion kann dann optional nachträglich durch den Benutzer gestartet werden.

Eine Key-Reduktion ist auch deshalb sinnvoll, weil das Lesen der Keys bei einer eventuell erneuten Simulation derselben Szene dadurch verbessert wird. Denn bei „CommOptKeys“ muss zum Lesen von Werten zwar nicht mehr jeder Wert, aber dennoch jeder Key der SPs und MPs gelesen werden.

Um bei „CommOptKeys“ die Animations-Keys für die Optimierung der Kommunikation zu nutzen, ist es erforderlich, die von den Interpolatoren produzierten Werte auch in MaxControl berechnen zu können, ohne dazu unnötig oft mit 3D-Studio-MAX kommunizieren zu müssen. Um dies zu erleichtern, wurden die Interpolatoren für die Animations-Tracks der SPs und MPs auf einfache Typen eingeschränkt. Für die MPs werden nur „On/Off“-Controller verwendet, bei denen jeder Key einen Wechsel des Wertes zwischen „true“ und „false“ bedeutet. SPs mit Werten vom Typ Boolean werden auf „Boolean Float“-Controller eingestellt, bei denen jeder Key im Gegensatz zum „On/Off“-Controller auch einen Wert besitzt, auf den der SP-Wert beim Erreichen dieses Keys schlagartig umgestellt wird. SPs mit Float- oder Integer-Werten werden auf „Linear Float“-Controller beschränkt, die zwischen den Werten der Keys linear interpolieren. Dies bedeutet zwar eine Einschränkung der Animationsmöglichkeiten, jedoch wird der Implementationsaufwand durch die Einschränkung verringert, da nur eine kleine Anzahl von Interpolatoren behandelt werden muss. Außerdem kann die Ausführungsgeschwindigkeit von MaxControl durch die so mögliche Optimierungsmethode „CommOptKeys“ im Durchschnitt verzehnfacht werden, weshalb die Einschränkungen der Controllertypen hier gerechtfertigt sind. Eine Erweiterung um die Unterstützung anderer Interpolationsmethoden wird für die Zukunft nicht ausgeschlossen.

Für die jetzt begrenzte Anzahl möglicher Interpolatoren wird die optimierte Kommunikation „CommOptKeys“ zwischen MaxControl und 3D-Studio-MAX im Folgenden erläutert.

Den größten positiven Effekt erzielt die Kommunikationsoptimierung bei SPs, deren Werte sich im Simulationsintervall fast nie ändern, weil diese SPs z.B. eher weitgehend unveränderliche Parameter für die Simulation sind. Ändert sich der Wert einer SP im Verlauf der Simulation fast nie, so muss bei der optimierten Kommunikation nur selten ein neuer Key für diese SP in die Szenenzustandsfolge geschrieben werden, sofern für diese SP nicht schon viele Keys in der vorläufigen Szenenzustandsfolge vorliegen, deren Werte von den zu schreibenden Werten abweichen. Und es gibt häufig auch nur wenige Keys, die für eine solche SP aus der Szenenzustandsfolge gelesen werden müssen, weil für eine solche SP in der Regel weder manuell noch durch eine vorhergehende Simulation viele Keys in der Szenenzustandsfolge erzeugt werden.

10.3.1.2 Zustandsübergänge bei der Kommunikationsoptimierung

Es wird nun noch einmal auf die relevanten Teile des Simulationszyklus eingegangen, um die dabei durchgeführten Kommunikationsoptimierungen auf dieser Basis erklären zu können. Wie im Verlauf der Simulation grundsätzlich Zustandsübergänge entstehen wurde in den Kapiteln 6.5 und 6.7 erläutert. Die dort verwendeten Bezeichnungen werden übernommen, um die Zustandsübergänge hier im Hinblick auf die nötigen Kommunikationsschritte genauer zu erläutern. Aus technischen Gründen kommen einige neue Zustandsübergänge und Kommunikationsschritte hinzu.

MaxControl übernimmt die Daten über das Animationsintervall aus 3D-Studio-MAX und bestimmt daraus das Simulationsintervall mit den Framenummern $a, a+1, \dots, b$. Schon vor der Simulation ist immer eine vorläufige Szenenzustandsfolge $(\delta_a, \dots, \delta_b)$ im 3D-Animationsprogramm gespeichert. Durch die Simulation wird für jeden Frame n in MaxControl ein Zustand σ_n erzeugt. Somit erzeugt MaxControl für ein Simulationsintervall von Frame a bis Frame b die Zustände $\sigma_a, \dots, \sigma_b$. Durch die Kommunikation mit 3D-Studio-MAX soll eine Szenenzustandsfolge $(\delta'_a, \dots, \delta'_b)$ in 3D-Studio-MAX entstehen, welche in Bezug auf die SPs die entsprechenden Zustände $\sigma_a, \dots, \sigma_b$ möglichst exakt repräsentiert. Wie bereits in Kapitel 6.4 erwähnt wurde, kann $(\sigma_a, \dots, \sigma_b)$ insbesondere in Bezug auf Fließkommawerte von den SP-Werten in $(\delta'_a, \dots, \delta'_b)$ abweichen, da 3D-Studio-MAX Fließkommawerte mit einer geringeren Genauigkeit speichert als MaxControl. Dies wirkt insbesondere auch auf die Übertragung solcher Werte von 3D-Studio-MAX nach MaxControl über MAXScript-Anweisungen, da MAXScript Fließkommawerte in Form von Strings mit einer noch geringeren Genauigkeit ausgibt als diese in 3D-Studio-MAX gespeichert werden.

Bei jedem Simulationsschritt zu einem Frame n werden zunächst MAXScript-Befehle an 3D-Studio-MAX gesendet, welche Werte von SPs und MPs lesen und sie in eine Datei schreiben. Diese Datei wird dann von MaxControl gelesen und interpretiert. Bei CommOptKeys werden dabei nicht nur die Werte der im aktuellen Frame n manuell kontrollierten SPs aus δ_n gelesen, sondern es werden die Werte aller SPs aus δ_n gelesen, wobei jedoch viele Optimierungen bezüglich der Keyframe-Technik ausgenutzt werden. Nur die Werte von in Frame n manuell kontrollierten SPs werden auch in den entsprechenden MaxControl-Zustand $\sigma_{n-1,0}$ übernommen. Die Werte der übrigen SPs werden nur für die Optimierung der Kommunikation verwendet und intern in MaxControl gespeichert.

Im Zuge der Optimierung werden bei den Lesevorgängen nicht in jedem Frame die jeweils gültigen Werte der SPs und MPs direkt gelesen, sondern es werden nur die **Keys** eingelesen, welche diese Werte festlegen. Dies spart viele unnötige Kommunikationsschritte ein. Wird im Folgenden vom Lesen oder Schreiben von SP- oder MP-Werten gesprochen, so wird vorausgesetzt, dass dabei auf effiziente Weise mit den zugehörigen Keys gearbeitet wird. Dieses effiziente Vorgehen wird weiter unten in Kap. 10.3.1.3 genauer erläutert.

Das Lesen der Werte der SPs und MPs aus 3D-Studio-MAX ist wie oben beschrieben in zwei Arbeitsschritten aufgeteilt. Zunächst werden die Werte aus den Szenenzuständen durch MAXScript-Befehle aus 3D-Studio-MAX gelesen und in eine Datei geschrieben. Im zweiten Schritt werden die so gewonnenen Daten aus dieser Datei in MaxControl eingelesen. Diese Trennung hat die folgenden technischen Gründe:

Zunächst zeigte sich, dass eine große Anzahl an Befehlen am schnellsten von 3D-Studio-MAX ausgeführt werden kann, wenn sie in einem einzigen Kommunikationsschritt von MaxControl an 3D-Studio-MAX durchgeführt werden. Unnötig viele Kommunikationsschritte verlangsamen die Ausführung signifikant. Daher werden zunächst so

viele Befehle wie möglich zu einer Befehlsgruppe zusammengefasst und dann in einem Kommunikationsschritt als Datei an 3D-Studio-MAX gesendet und dort ausgeführt. Dies gilt sowohl für Schreib- als auch für Lesebefehle. Während der Ausführung einer Befehlsgruppe zum Lesen von Werten kann MaxControl noch keine Daten entgegennehmen. Erst nach der Ausführung der Befehlsgruppe werden die so produzierten Daten von MaxControl aus der erstellten Datei gelesen.

Der Umweg über Dateien sowohl bei der Erstellung der auszuführenden MAXScript-Befehle als auch bei der Zwischenspeicherung der dabei gelesenen Daten wird durch Instabilitäten der beteiligten Programmkomponenten nötig. Darauf wird in Kapitel 10.3.2 näher eingegangen.

Um die alternative Verwendung des Verfahrens CommOptMPs zu ermöglichen, werden die Werte der MPs wie in Kap. 10.3.1.1 ab Seite 206 erläutert immer einen Frame im Voraus gelesen, damit im nächsten Simulationsschritt bereits bekannt ist, welche SPs dann als manuell kontrolliert gelten sollen. CommOptMPs verwendet diese Daten, um in jedem Frame gezielt nur die SP-Werte aus 3D-Studio-MAX auszulesen, die im aktuellen Frame auch wirklich als manuell kontrolliert gelten sollen. Dies erfordert, dass die Werte der zugehörigen MPs bereits im Voraus bekannt sind, noch bevor der aktuelle Leseschritt für die SPs durchgeführt wird. Alternativ zuerst in einem Leseschritt die Werte der MPs aus δ_n zu lesen und dann in einem zweiten Leseschritt nur die manuell kontrollierten SPs aus δ_n zu lesen hätte die erforderliche Anzahl an Leseschritten verdoppelt, was die für die Kommunikation benötigte Zeit wieder erhöht hätte. Beide Schritte können in dieser Reihenfolge nicht in einem einzigen Kommunikationsschritt durchgeführt werden, weil MaxControl die Ergebnisse eines Kommunikationsschrittes wie oben erwähnt erst dann auswerten kann, wenn dieser Schritt vollständig durchgeführt wurde. Das Lesen der MP-Werte muss also beendet sein, bevor diese MP-Werte in MaxControl verwendet werden können, um in einem weiteren Kommunikationsschritt gezielt nur die erforderlichen SP-Werte zu lesen. Werden die MP-Werte jedoch immer einen Frame im Voraus gelesen, so können in **einem** Leseschritt die manuell kontrollierten SPs für den aktuellen Frame n sowie die MPs für den nächsten Frame $n+1$ gelesen werden.

Nach dem Lesen wird ggf. der eigentliche Simulationsschritt zu Frame n durchgeführt, der die neuen Werte der SPs bestimmt. Nach dem Simulationsschritt werden MAXScript-Befehle erzeugt und ausgeführt, welche den neuen MaxControl-Zustand in den zugehörigen Szenenzustand δ_n' in 3D-Studio-MAX schreiben. Danach kann es aus technischen Gründen notwendig sein, die Werte einiger SPs noch einmal aus dem neuen Szenenzustand δ_n' zu lesen und in den MaxControl-Zustand zu übertragen. Dies ist nötig, wenn die Werte einiger SPs nach dem Schreiben der neuen SP-Werte in δ_n' indirekt durch 3D-Studio-MAX „nachbearbeitet“ werden, als Folge von Veränderungen an anderen SPs durch MaxControl.

Dies trifft unter anderem auf inverse Kinematik (siehe [Jack06]) zu, eine 3D-Studio-MAX-Funktion, die beispielsweise automatisch die Beine einer Figur bewegen kann, wenn durch den Benutzer oder auch einen Simulationsschritt mit MaxControl nur die Füße dieser Figur bewegt wurden. Eine Änderung an den Positionen der Füße führt indirekt auch zu einer sofortigen Veränderung der Positionen der Beine, die innerhalb von 3D-Studio-MAX automatisch durchgeführt wird und nicht aus der eigentlichen Simulation in MaxControl hervorgeht. Die Veränderungen eines Szenenzustandes n durch den entsprechenden Simulationsschritt ziehen also eine weitere Veränderung des Szenenzustandes n nach sich, die von MaxControl zunächst unbemerkt bleibt, auch wenn die Positionen der Beine SPs sind. Die letzten der Simulation bekannten Positionen der Beine sind damit nicht mehr aktuell und müssen vor dem nächsten Simulationsschritt zum nächsten Frame $n+1$ aus dem

vorangehenden automatisch durch 3D-Studio-MAX veränderten Szenenzustand n ausgelesen werden, wenn die Positionen der Beine SPs sind.

Die Zustandsübergänge sowie Lese- und Schreibvorgänge für einen Simulationsschritt werden in der folgenden Abbildung 70 schematisch dargestellt. Die oben angesprochene indirekte Änderung von SPs erfordert die Einführung neuer Zwischenzustände. Dies sind ein neuer MaxControl-Zustand σ_n^{pre} sowie ein neuer vorläufiger Szenenzustand δ_n^{pre} .

MaxControl arbeitet neben den MaxControl-Zuständen auch mit internen Daten. Dazu gehören unter anderem auch die im Voraus gelesenen MP-Werte aus Frame $n+1$, welche die für Frame n gültigen MP-Werte nicht überschreiben dürfen und somit gesondert gespeichert werden müssen, ebenso wie die aus technischen Gründen gelesenen Keys für SPs, die in Frame n gar nicht manuell kontrolliert sind. Die gesonderte Speicherung der MP-Werte aus Frame $n+1$ wird in der folgenden Abbildung 70 als Erweiterung der MaxControl-Zustände durch die Zustände μ_{n-1}, μ_n und μ_{n+1} gezeigt.

In der nun folgenden Abbildung 70 entsprechen die Pfeile im Wesentlichen denen in Abbildung 28 aus Kapitel 6.7 auf Seite 93. Die Pfeilnotationen $\text{---} \blacktriangleright$ und $\text{.....} \blacktriangleright$ weisen darauf hin, dass es sich um Datenübertragungen zwischen MaxControl und 3D-Studio-MAX bzw. um Übergänge mittels Key-Techniken handelt. Es wird hier das Simulationsintervall zu Frame n für $a < n < b$ betrachtet. Für den Fall $n = a + 1$ gilt $\mu_a = \emptyset$.

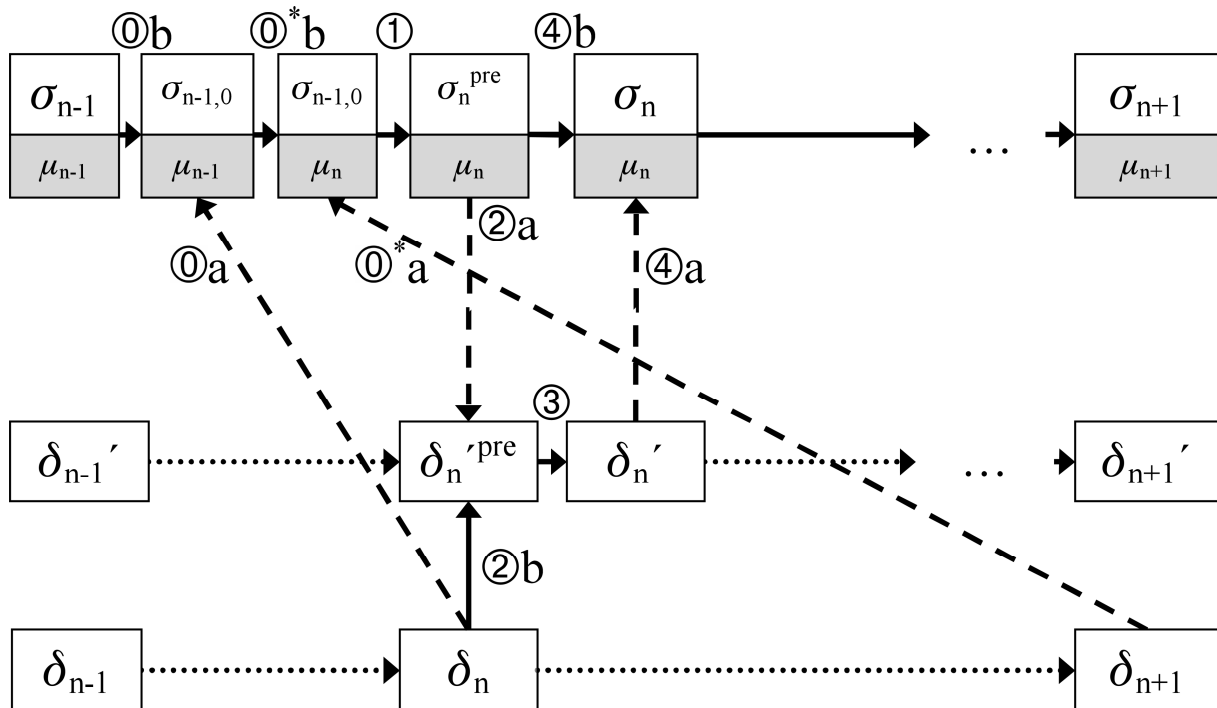


Abbildung 70: Zustandsübergänge mit Kommunikation

Es werden die folgenden Schritte in der angegebenen Reihenfolge durchgeführt, in Abbildung 70 sind auch die Pfeile entsprechend bezeichnet:

①a/b: Hier werden die Werte der in Frame n manuell kontrollierten SPs aus δ_n gelesen. Sie werden in Zustand σ_{n-1} eingelesen und überschreiben die entsprechenden Werte in σ_{n-1} . Es entsteht $\sigma_{n-1,0}$. Auch für nicht manuell kontrollierte SPs werden ggf. neue Keys gelesen und intern in MaxControl gespeichert.

- ①* a/b: Es werden die Werte der MPs aus δ_{n+1} gelesen, noch bevor die Simulation zu σ_n stattfindet. Diese Werte überschreiben dabei **nicht** die in σ_n gültigen Werte für die MPs, sondern sie werden gesondert in MaxControl im Erweiterungszustand μ_n zwischengespeichert, um dann sofort im nächsten Simulationsschritt zu Frame n+1 zur Verfügung zu stehen. Dadurch wird μ_{n-1} zu μ_n .
- ①: Hier wird der Simulationsschritt zu Frame n durchgeführt. Der Schritt endet nicht wie bisher in dem MaxControl-Zustand σ_n , sondern in dem vorläufigen MaxControl-Zustand σ_n^{pre} , der durch die Schritte ③ und ④ noch nachbearbeitet wird und dann erst zu σ_n wird.
- ②a/b: Der vorläufige MaxControl-Zustand σ_n^{pre} wird in den vorläufigen Szenenzustand δ_n übertragen und überschreibt dort vorhandene SP-Werte. Manuell kontrollierte Werte für SPs aus δ_n bleiben dabei erhalten. Dadurch wird δ_n zu dem weiterhin vorläufigen Zustand δ_n^{pre} .
- ③: Hier führt 3D-Studio-MAX gemäß inverser Kinematik oder ähnlicher Abhängigkeiten Zustandsänderungen durch.
- ④a/b: Die in ③ durchgeführten Änderungen werden nach σ_n^{pre} übertragen, dadurch entsteht der MaxControl-Zustand σ_n .

In der technischen Umsetzung ist der zusätzliche Lesevorgang ④a/b eigentlich Teil des **folgenden** Simulationsschrittes zu Frame n+1. So kann er in den weiter oben genannten zweiteiligen Leseschritt (siehe Seite 210) integriert werden, wobei er allerdings in den Simulationsschritt zu Frame n+1 integriert wird, indem dieser Lesevorgang dort vor allen anderen Lesevorgängen ausgeführt wird.

Für den Simulationsschritt zu Frame n+1 können auf diese Weise sowohl das Lesen einiger SPs aus dem vorangehenden Frame n, danach das Lesen der im aktuellen Frame n+1 manuell kontrollierten SPs sowie danach das Lesen der MP-Werte aus dem folgenden Frame n+2 in dieser Reihenfolge in einem Leseschritt und damit in **einem** Kommunikationsschritt durchgeführt werden.

Die obige Abbildung 70 zeigt diesen zusätzlichen Lesevorgang dennoch als letzten Schritt des zugehörigen Simulationsschrittes, weil die Zustandsübergänge eines Simulationsschrittes so vollständig und leichter verständlich dargestellt werden können.

10.3.1.3 Algorithmen zur Kommunikationsoptimierung mittels Key-Technik

Um nun die optimierte Kommunikation zu erklären, wird vereinfachend angenommen, dass die aktuellen Werte der MPs zu jedem Zeitpunkt bereits bekannt sind. Die Kommunikationsoptimierungen beim Lesen der MPs sind einfacher als die Optimierungen beim Lesen der SPs und basieren auf denselben Prinzipien, weshalb hier nur das Lesen und Schreiben der SPs genauer erläutert werden soll.

Es wird weiterhin angenommen, dass auch keine Nachbearbeitung im Sinne der Schritte ③ und ④a/b aus Abbildung 70 des vorangehenden Kapitels nötig ist. Die in solchen Fällen stattfindende Kommunikation beim erneuten Lesen ist nicht optimiert.

Der Simulationszyklus wurde grundsätzlich bereits in den Kapiteln 9.3 und 10.2.2 erläutert. Zur Erklärung der optimierten Kommunikation wird das Flussdiagramm in Abbildung 51 aus Kapitel 9.3 in der folgenden Abbildung 71 in den gestrichelt umrahmt dargestellten Bereichen bezüglich der Kommunikation genauer spezifiziert und erläutert. Dabei wird der Aufruf von `readManualValues(simFrame)` aus Abbildung 51 ersetzt durch die Aufrufe `askAtFrame(simFrame)`, `sendCommand()`, `readAnswer(simFrame)`. Der Aufruf `putValues(simFrame)` wird jetzt unter Beachtung der optimierten Kommunikation geändert.

Es werden nur die Kommunikationsschritte ① und ② gezeigt, die Kommunikationsschritte ③* und ④ werden in Abbildung 71 und auch im Folgenden nicht behandelt.

Wie bereits in Kap. 9.3 erläutert wurde, orientieren sich die Darstellungen von Variablen, Bedingungen und Anweisungen in den Flussdiagrammen an Java-Syntax. Insbesondere wird ab hier auch der NOT-Operator in Bedingungen als „!“ dargestellt und der Wert einer unbelegten Variablen für Objekte wird als „NULL“ bezeichnet, wobei dieses Schlüsselwort entsprechend auch in Bedingungen verwendet werden kann. Die Verknüpfung „&&“ bezeichnet in Bedingungen den UND-Operator, während „||“ für ODER steht.

Die bezüglich der Änderungen gegenüber Abbildung 51 relevanten Methoden, auf die im Folgenden Bezug genommen wird, werden direkt nach Abbildung 71 erläutert. In Abbildung 71 fällt auf, dass die Anweisung `sendCommand()` lediglich direkt nach `askAtFrame()` und am Ende der Simulation ausgeführt wird. Dies wurde aus Gründen der Effizienz absichtlich so gewählt. Wie in Kapitel 10.3.1.2 erläutert wurde, werden Skriptbefehle schneller ausgeführt, wenn möglichst viele von ihnen in einem Schritt gebündelt an 3D-Studio-MAX übermittelt und dort ausgeführt werden, statt sie befehlsweise zu übermitteln und auszuführen. Deshalb wird `sendCommand()` nur dann ausgeführt, wenn es wirklich erforderlich ist. Dies ist einmal am Ende der Simulation erforderlich, weil die letzten Befehle, die eventuell einige neue SP-Werte für die Animation in die Szenenzustandsfolge schreiben sollen, noch ausgeführt werden müssen. Weiterhin ist `sendCommand()` zwischen `askAtFrame` und `readAnswer` nötig, weil die in `askAtFrame` vorbereiteten Befehle, die aus der Szenenzustandsfolge gelesene SP-Werte in eine Datei schreiben, vollständig ausgeführt sein müssen, bevor diese Werte durch `readAnswer` vollständig aus der Datei ausgelesen werden können.

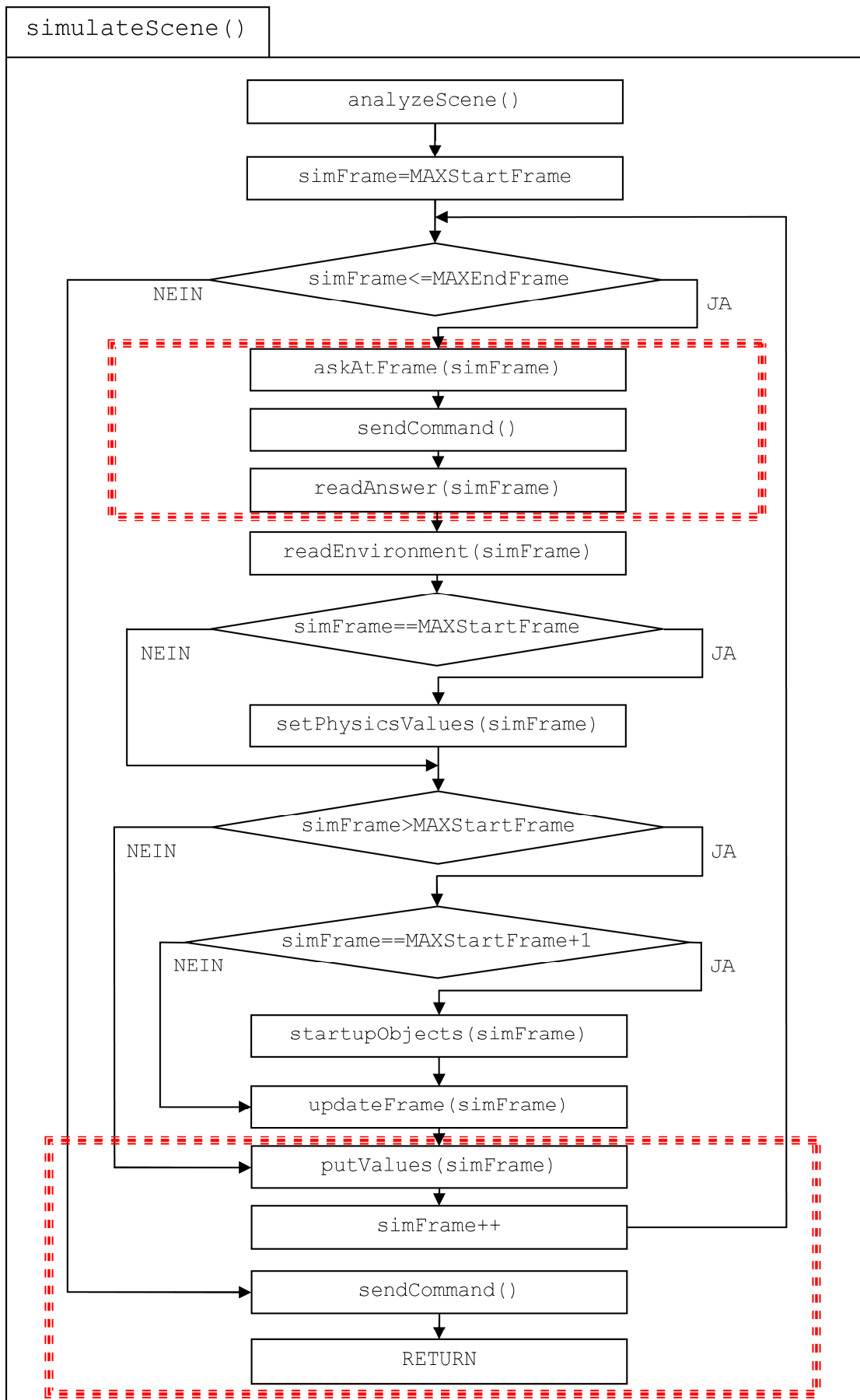


Abbildung 71: Simulationszyklus mit optimierter Kommunikation

Relevante Methoden aus Abbildung 71:

`askAtFrame(simFrame):`

Diese Methode führt indirekt bei jedem Simulationsobjekt in jeder seiner SPs den Aufruf `askValue(simFrame)` (siehe Abbildung 74) aus. Dadurch werden die MAXScript-Befehle vorbereitet, welche die Werte der SPs aus dem aktuellen Frame `simFrame` der Szenenzustandsfolge auslesen und sie in eine Datei schreiben.

`sendCommand():`

Sendet alle bisher vorbereiteten und noch nicht gesendeten MAXScript-Befehle an 3D-Studio-MAX und führt sie dort aus

`readAnswer(simFrame):`

Führt indirekt bei jedem Simulationsobjekt in jeder seiner SPs den Aufruf `readValue(simFrame)` aus. Dadurch werden die Werte der SPs aus der Datei gelesen, die vorher durch die oben genannten MAXScript-Befehle erzeugt wurde

`putValues(simFrame):`

Führt indirekt bei jedem Simulationsobjekt in jeder seiner SPs den Aufruf `putAtFrame(simFrame)` (siehe Abbildung 75) aus. Dadurch werden MAXScript-Befehle erzeugt, welche die Werte der SPs des neuen MaxControl-Zustandes in den Szenenzustand `simFrame` schreiben. Sie werden beim nächsten Aufruf von `sendCommand()` ausgeführt.

Ebenso fällt im Vergleich zu Abbildung 51 aus Kapitel 9.3 (erläutert in Kap. 10.2.2) auf, dass der Befehl `putValues(simFrame)` hier auch für den ersten Frame `a` (`=MAXstartFrame`) des Simulationsintervalls aufgerufen wird und nicht erst ab `simFrame > MAXstartFrame`. Dies wird für die Kommunikationsmethode `CommOptMPs` benötigt, worauf weiter unten im Zusammenhang mit „Left Anchoring“ auf Seite 226 näher eingegangen wird.

Die oben genannten indirekt aufgerufenen Methoden `askValue`, `readValue` und `putAtFrame` sind Teile jeder SP, sie werden im Folgenden genauer erklärt.

Dabei gehen wir von der in `CommOptKeys` geforderten Repräsentation der SP-Tracks durch Keys mit linearer Interpolation und mit Extrapolation durch konstante Fortsetzung aus. Jeder Track `t` besteht damit aus einer Folge ($K_1, K_2, \dots, K_{r(t)}$) von Keys. Die Keys sind Tripel mit den Komponenten `index`, `frame` und `value` für den Index bzgl. der Zählung der Keys von links nach rechts, für die Nummer des Frames, der den Key enthält bzw. für den Wert des Keys. Dabei kann `r(t)` auch 0 sein, da ein Track mit konstantem Wert auch ohne jeden Key repräsentiert werden kann (siehe Kapitel 3.5, Seite 47). Der Wert `K.frame` für einen Key `K` kann außerhalb des Simulationsintervalls liegen.

Letzteres ist dadurch gegeben, dass das Animations- und Simulationsintervall auch nachträglich beliebig wählbar ist, wie in Kapitel 6.5 erläutert wurde. So kann zunächst eine Simulation von Frame 0 bis Frame 200 durchgeführt werden und in der so animierten Szene kann danach noch eine Simulation von Frame 150 bis Frame 300 berechnet werden. Für diese zweite Simulation wäre die Nummer des ersten Frames im Simulationsintervall dann nicht 0 sondern 150. Daraus ergibt sich im Hinblick auf die erste Simulation auch, dass die zweite Simulation **vor** ihrem ersten Frame 150 bereits Keys finden kann.

In dem zu diskutierenden Verfahren `CommOptKeys` ist für eine betrachtete SP die Lage einiger Keys bzgl. des aktuell zu simulierenden Frames n wichtig. Wir sprechen, falls vorhanden, vom „momentan nächsten Key“ (der erste rechtsseitige Key K mit $K.frame > n$), vom „momentan letzten Key“ (der in der Key-Folge letzte linksseitige Key K mit $K.frame \leq n$) und vom „momentan vorletzten Key“ (der in der Key-Folge letzte linksseitige Key K' mit $K'.frame < K.frame$, wobei K der „momentan letzte Key“ ist). Diese Keys werden im folgenden „momentan relevante Keys“ genannt. Die entsprechenden Felder in mit `CommOptKeys` optimierten SPs sind `nextKey`, `lastKey1` bzw. `lastKey2`. Jedes dieser Felder kann den Java-Nullwert `NUL`L tragen, wenn für den aktuellen Frame n kein entsprechender Key existiert.

Zur Verdeutlichung werden zwei Situationen eines Beispieltracks dargestellt:

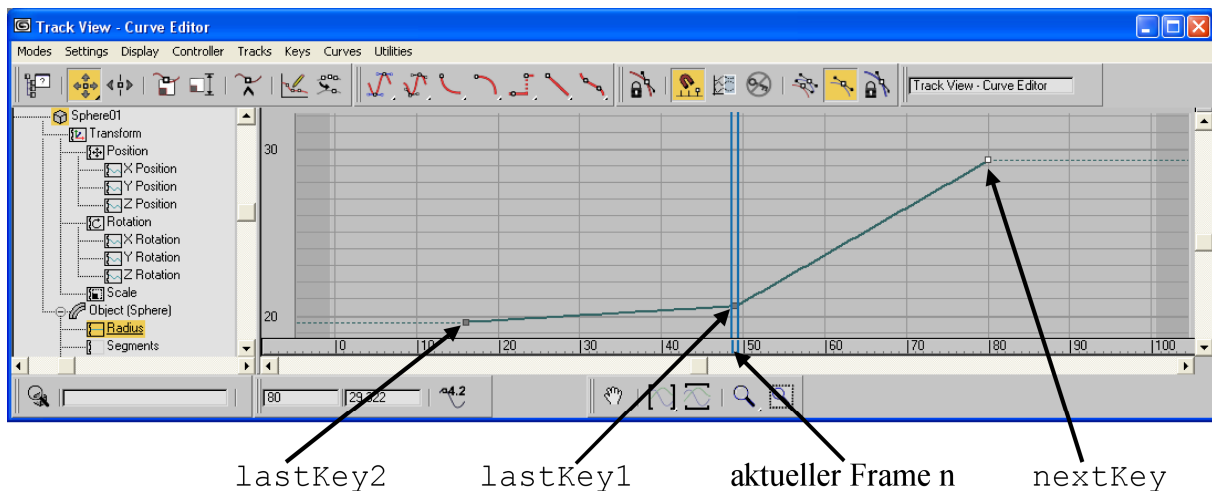


Abbildung 72: Beispiel für momentan relevante Keys, $lastKey1.frame = n$

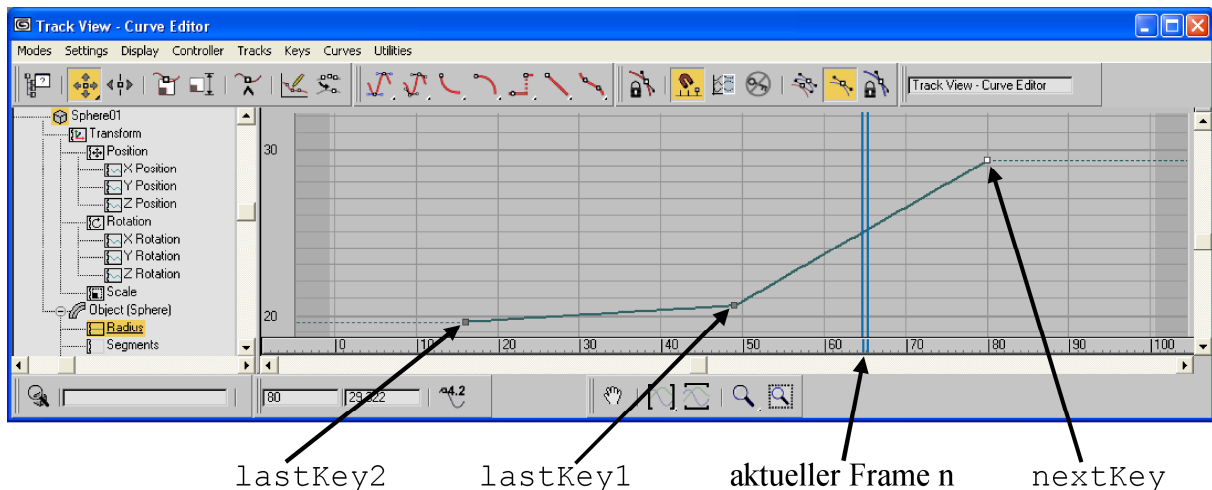


Abbildung 73: Beispiel für momentan relevante Keys, $lastKey1.frame < n$

Es fällt auf, dass die Werte zu `nextKey`, `lastKey1` und `lastKey2` in beiden Situationen jeweils gleich sind, obwohl sich die Nummer des aktuellen Frames geändert hat. Für wachsendes n ändern sich die momentan relevanten Keys erst bei $n = nextKey.frame$. Auf diesen Sachverhalt kommen wir auf Seite 219 zurück. Der aktuelle Wert der SP ergibt sich jeweils durch Interpolation zwischen den Werten von `nextKey` und `lastKey1`. Ist einer dieser Keys nicht vorhanden, wird zur Bestimmung des Wertes die vorher festgelegte Extrapolation durch konstante Fortsetzung verwendet. Fehlen beide Keys, enthält der Track

keine Keys und der aktuelle Wert der damit konstanten SP ist direkt ohne Keys festgelegt. Es ist zu beachten, dass bei fehlendem `lastKey1` auch kein `lastKey2` existiert.

Bei Betrachtung der Zustandsübergänge in `CommOptKeys` muss man zwischen den aktuell (aus der vorläufigen Szenenzustandsfolge) gelesenen und dem aktuell (aus dem `MaxControl`-Zustand in die Szenenzustandsfolge) zu schreibenden Wert einer gegebenen SP unterscheiden. Der erstgenannte Wert ergibt sich wie oben erläutert durch Interpolation oder Extrapolation aus den momentan relevanten Keys bzw. durch direkte Bestimmung, falls kein Key im betreffenden Track vorhanden ist. Die direkte Bestimmung sei im Folgenden im Begriff Extrapolation enthalten. Der zweitgenannte Wert ist bei aktueller manueller Kontrolle gleich dem erstgenannten Wert, ohne manuelle Kontrolle wird er durch Simulation bestimmt.

In den Lese- und Schreibverfahren werden für die oben genannten Werte die Felder `value` und `lastValue` genutzt. Das Feld `value` speichert den aktuell zu schreibenden Wert der SP. Das Feld `lastValue` trägt genau dann den aktuell gelesenen Wert der SP, wenn diese in Frame `n` **nicht** manuell kontrolliert ist. In diesem Fall kann der Wert von `lastValue` von `value` abweichen, da `value` dann durch die Simulation bestimmt wird. So kann vor dem Schreibvorgang festgestellt werden, ob der durch die Simulation neu bestimmte Wert `value` von `lastValue` abweicht und in die Szenenzustandsfolge geschrieben werden muss, oder ob dort bereits der gleiche Wert gültig ist und der Schreibvorgang somit entfallen kann. Bei manueller Kontrolle bleibt der gelesene Wert in Szenenzustandsfolge erhalten, so dass der Schreibvorgang entfallen kann, weshalb kein Vergleichswert in `lastValue` benötigt wird.

Die im Folgenden erklärten Lese- und Schreibverfahren in `CommOptKeys` speichern den momentan relevanten Ausschnitt des Tracks der betrachteten SP in den Feldern `lastKey2`, `lastKey1`, `nextKey`, `value` und `lastValue`.

Wir kommen zur Erklärung dieser Lese- und Schreibverfahren. Zum Verfahren `CommOptKeys` gehört die Unterklasse `MC_PropertyCommOptimized` von `MC_Property`. Eine SP, die `CommOptKeys` nutzen soll, ist daher stets eine Instanz einer Unterklasse von `MC_PropertyCommOptimized`.

Zu den Leseverfahren:

Jede SP erbt aus `MC_Property` unter anderem eine Methode der Signatur `void askValue(long f)`. In der Unterklasse `MC_PropertyCommOptimized` ist diese Methode so implementiert, dass sie das Verfahren `CommOptKeys` nutzt. Die folgende Abbildung 74 stellt den Ablauf der Methode `askValue(long f)` dar, die in der Erläuterung zu `askAtFrame` nach Abbildung 71 eingeführt wurde. Dabei sind die folgenden Bedingungen und Anweisungen näher zu erläutern:

`!valueRead:`

Überprüft, ob noch kein Wert für diese SP gelesen wurde. Dies ist genau zu Beginn der Simulation der Fall, wenn also der aktuelle Frame der erste Frame `a` des Simulationsintervalls ist.

`askFirstValueAndFirstKey(f):`

Liest den Wert der SP im Frame Nr. `f(=a)`, auch wenn sich dort kein Key befindet. Dies ist möglich, weil der Wert einer SP in jedem Frame auch direkt gelesen werden kann, ohne auf Keys zuzugreifen. Zusätzlich werden ein „momentan nächster Key“ und ein „momentan letzter Key“ gesucht. Nur die unten beschriebenen Anchoring-Verfahren können einen

„momentan vorletzten Key“ benötigen. Da diese Anchoring-Verfahren jedoch niemals auf Daten aus Frames vor dem ersten Frame *a* der Simulation zugreifen, wird ein „momentan vorletzter Key“ noch nicht benötigt.

Wurde die Methode `askFirstValueAndFirstKey` aufgerufen, so bewirkt die später in `simulateScene()` nach `sendCommand` im Rahmen von `readAnswer` ausgeführte Methode `readValue`, dass die Variablen `value`, `nextKey` und `lastKey1` mit den Ergebnissen der obigen Suche besetzt werden, `nextKey` und `lastKey1` ggf. mit `NULL`.

`nextKey!=null && nextKey.frame==f:`

Diese Bedingung überprüft die im Zusammenhang mit Abbildung 72 und Abbildung 73 geschilderte Situation beim Verschieben des aktuellen Frames nach rechts und bestimmt, ob ein weiterer Key für die SP gesucht werden muss. Nur, wenn die Bedingung erfüllt ist, muss eine Neubestimmung der momentan relevanten Keys vorbereitet werden.

`askNextKey()` :

Sucht einen Key, der auf `nextKey` folgt und liest ggf. seinen Wert.

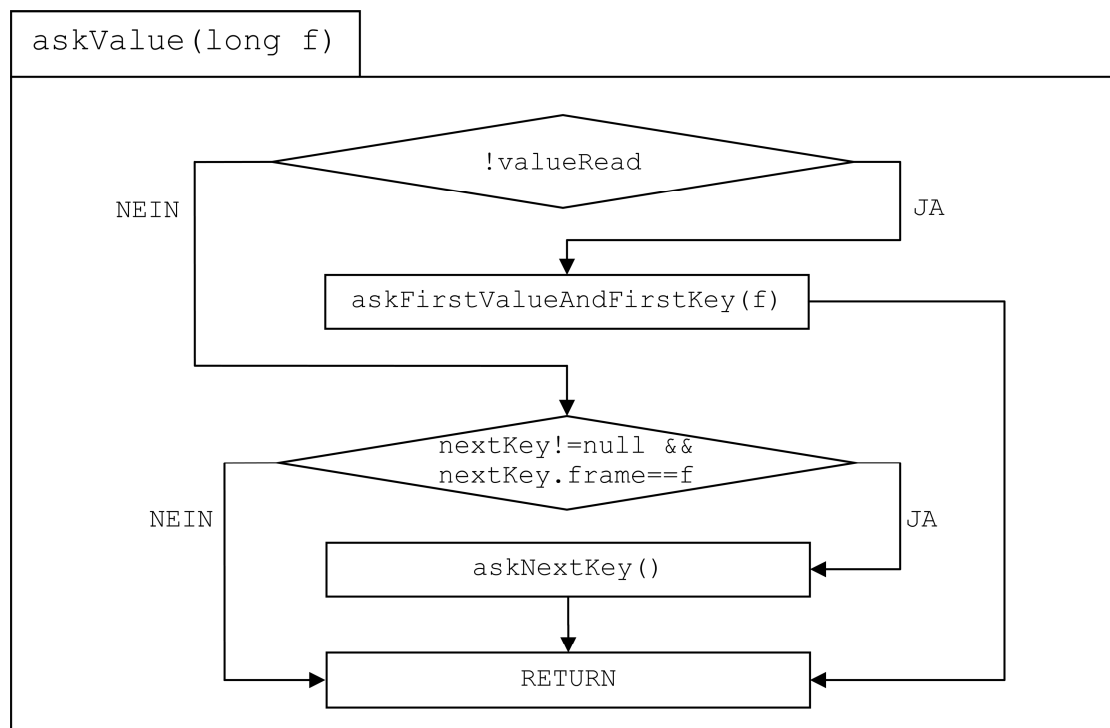


Abbildung 74: Ablaufdiagramm zu `askValue`

Wurde `askNextKey()` aufgerufen, so führt die oben genannte Methode `readValue` der Reihe nach die folgenden Ersetzungen für die betrachtete SP aus:

`lastKey2` ← `lastKey1`

`lastKey1` ← `nextKey`

`value` ← `nextKey.value`, falls aktuell manuelle Kontrolle vorliegt

`lastValue` ← `nextKey.value`, falls aktuell keine manuelle Kontrolle vorliegt

`nextKey` ← neuer gefundener Key bzw. `NULL`

Bei Nichterfüllung der Bedingung `nextKey!=null && nextKey.frame==f` sind die momentan relevanten Keys weiterhin gültig. Dann wird der Wert aus dem aktuellen

Szenenzustand durch Interpolation bzw. Extrapolation bestimmt. Dieser Wert sei v . Die Methode `readValue` führt dann die folgende Ersetzung aus:

```
value      ← v, falls aktuell manuelle Kontrolle vorliegt
lastValue ← v, falls aktuell keine manuelle Kontrolle vorliegt
```

Nach diesen für $f \neq a$ von `readValue` durchgeführten Ersetzungen trägt `value` den korrekten Wert für den nächsten Szenenzustand, falls die betrachtete SP in Frame f manuell kontrolliert ist. Anderenfalls wird der Wert von `value` anschließend im Rahmen von `updateFrame` durch Simulation bestimmt und kann mit dem in `lastValue` gespeicherten Wert verglichen werden.

Für $f=a$ gelten alle SPs als manuell kontrolliert und der Aufrufe von `startupObjects` und `updateFrame` werden übersprungen.

Ein Diagramm zu `readValue` kann nach den obigen Erläuterungen entfallen, zumal sein Aufbau dem von `askValue` sehr ähnlich ist.

Zu den Schreibverfahren:

Jede SP besitzt eine Methode der Signatur `void putAtFrame(long f)`, die in der Erläuterung zu `putValues` nach Abbildung 71 eingeführt wurde. In der Klasse `MC_PropertyCommOptimized` ist diese Methode so implementiert, dass sie das Verfahren `CommOptKeys` nutzt. Die folgende Abbildung 75 stellt den Ablauf der Methode dar.

Es werden nun Bedingungen und Methodenaufrufe aus Abbildung 75 einzeln erläutert.

```
!manual && valueChanged() :
```

Fall $f=a$:

In diesem Fall sind alle SPs manuell kontrolliert, so dass diese Methode aufgrund der Bedingung `!manual` sofort verlassen wird.

Fall $f \neq a$:

Das Schreiben eines neuen Wertes in den aktuellen Frame der Szenenzustandsfolge soll nur dann stattfinden, wenn der durch die Simulation bestimmte Wert `value` von dem in `lastValue` gespeicherten Wert abweicht, der zuvor aus dem aktuellen Frame der vorläufigen Szenenzustandsfolge gelesen worden ist. Eine solche Abweichung kann nur dann auftreten, wenn die SP im aktuellen Frame nicht manuell kontrolliert ist. Daher wird zuerst die Bedingung `!manual` überprüft, wobei `manual` den aktuellen Wert der zur SP gehörigen MP speichert. Dieser einfache Test kann schneller durchgeführt werden als die aufwändigere Methode `valueChanged()`, die auch die Ungenauigkeiten bei der Speicherung von Werten in 3D-Studio-MAX berücksichtigt.

Diese oben erläuterte Bedingung `!manual && valueChanged()` leistet einen der Hauptbeiträge zur Kommunikationsoptimierung. Sie verhindert das unnötige Schreiben eines Keys, wenn dessen Wert an der zugehörigen Stelle in der Szenenzustandsfolge bereits gültig ist. Es liege die in Abbildung 76 folgende Situation in einem Track vor.

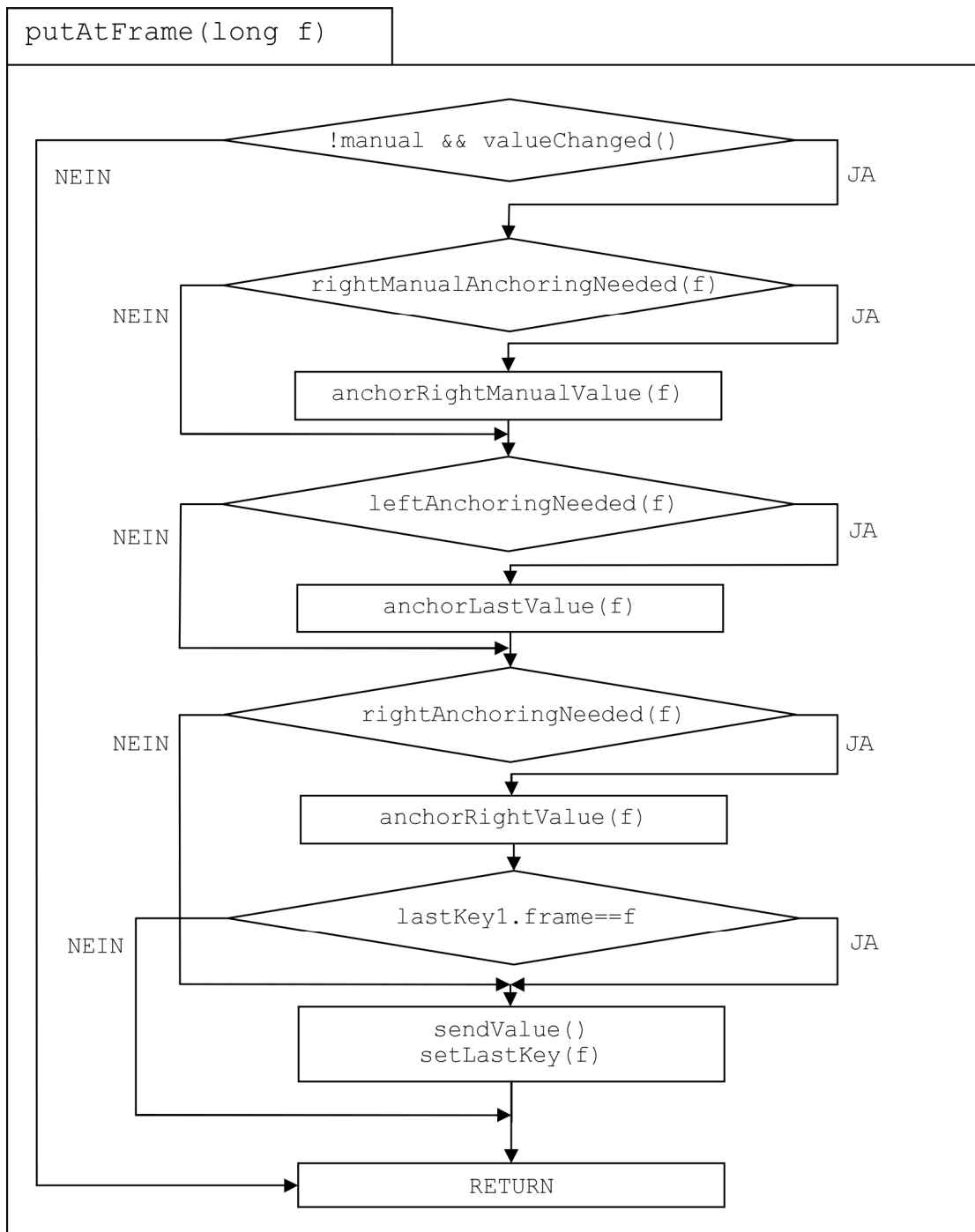


Abbildung 75: Ablaufdiagramm zu putAtFrame

Dann ist das Setzen des in Abbildung 77 folgenden Keys unnötig, da sein Wert an dieser Stelle allein durch die lineare Interpolation der beiden bereits vorhandenen Keys gültig ist. Fälle dieser Art werden durch die Bedingung erkannt und es wird dann kein neuer Key hinzugefügt.

Das Auftreten solcher Fälle anzunehmen ist realistisch. Häufig werden z.B. bei einer erneuten Simulation im relevanten Bereich wieder die gleichen Werte durch die Simulation erzeugt, die auch durch die vorangehende Simulation bestimmt wurden. Damit tritt der Fall ein, dass keine von den bisher im Track gültigen Werten abweichende Werte geschrieben werden sollen. Auch die Annahme, dass sich vor der erneuten Simulation nur relativ wenige Keys im relevanten Bereich des Tracks befinden, ist plausibel. Die im Beispiel gezeigte Grade kann

durch nachträgliches Anwenden der Key-Reduktion entstanden sein. Die aktuelle Version von CommOptKeys kann für Double-SPs solche linearen Zusammenhänge auch unmittelbar erkennen und erzeugt dann direkt weniger redundante Keys, so dass ein vergleichbares Beispiel auch so entstanden sein kann. Eine weitere Möglichkeit für ein passendes Beispiel ist die konstante Fortsetzung des Wertes des letzten Keys im Track. Wenn sich nach diesem letzten Key der SP-Wert im weiteren Verlauf der aktuellen Simulation nicht ändert, dann werden auch keine weiteren redundanten Keys gesetzt. Dieser Fall tritt oft bei SPs auf, die eher Parameter der Simulation darstellen, deren Werte sich im Verlauf der Simulation nicht ändern.

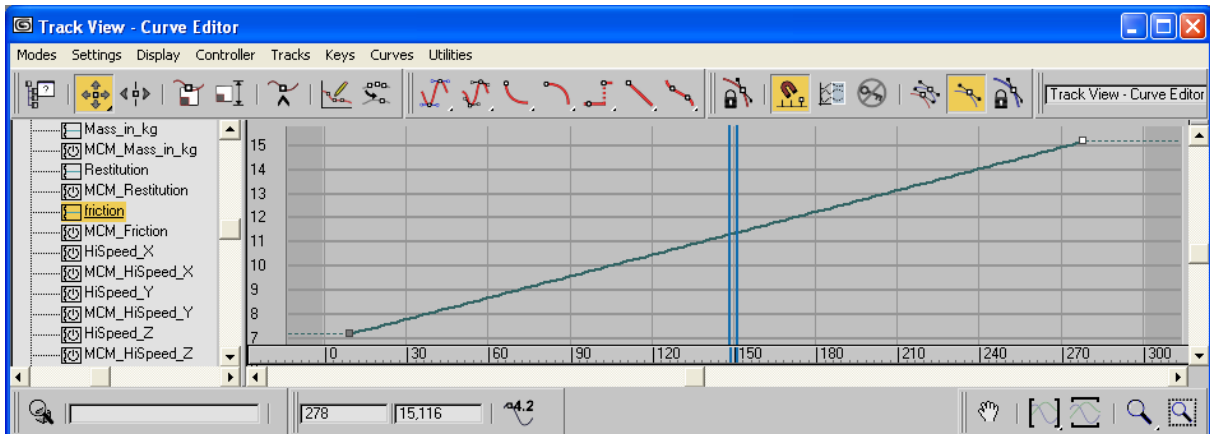


Abbildung 76: Beispielsituation vor dem Schreiben eines redundanten Keys

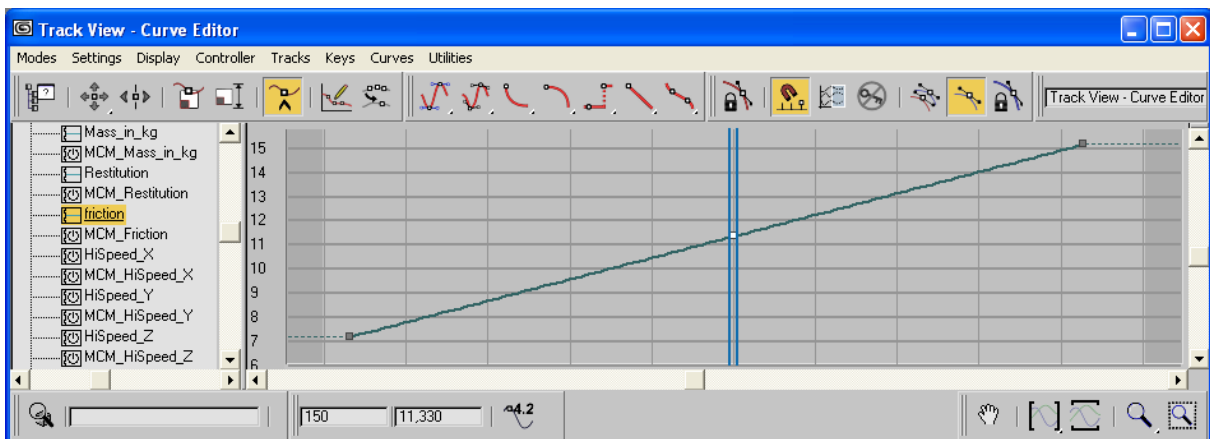


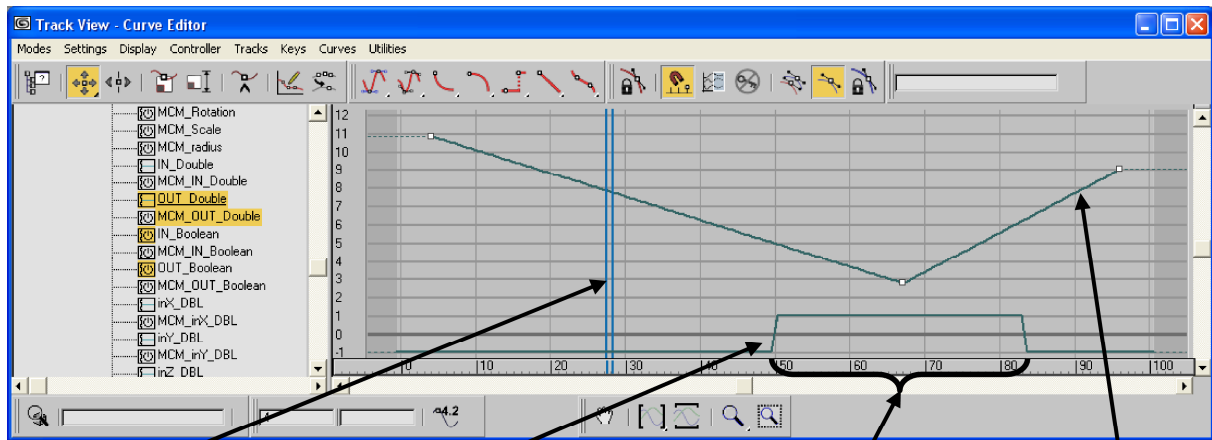
Abbildung 77: Setzen eines redundanten Keys

Die meisten Bedingungen und Anweisungen der Methode `putAtFrame(long f)` sind Teil eines Mechanismus, der hier „Anchoring“ genannt wird. Dabei wird durch zusätzliche Keys dafür gesorgt, dass das Setzen eines Keys im aktuellen Frame nicht zu unerwünschten Seiteneffekten auf die umliegenden Werte führt. Das Anchoring kann auch das Setzen eines neuen Keys im aktuellen Frame unnötig machen. Die auftretenden Fälle und die entsprechenden Anchoring-Maßnahmen werden im Folgenden erläutert.

Right Manual Anchoring:

Das so genannte „Right Manual Anchoring“ soll manuell kontrollierte Intervalle, die im Track einer SP hinter dem aktuellen Frame liegen, vor ungewollten Veränderungen schützen, die durch Setzen eines neuen Keys entstehen können.

Das folgende Beispiel illustriert das Right Manual Anchoring, wobei der Track einer SP und der Track der zugehörigen MP in einer kombinierten Darstellung von 3D-Studio-MAX angegeben sind. Es handelt sich bei der ersten Abbildung 78 um eine Situation unmittelbar vor dem Schreibvorgang im aktuellen Frame:



aktueller Frame Werte der zugehörigen MP manuell kontrolliertes Intervall Werte der SP
Abbildung 78: Beispielsituation für Right Manual Anchoring vor dem Schreiben

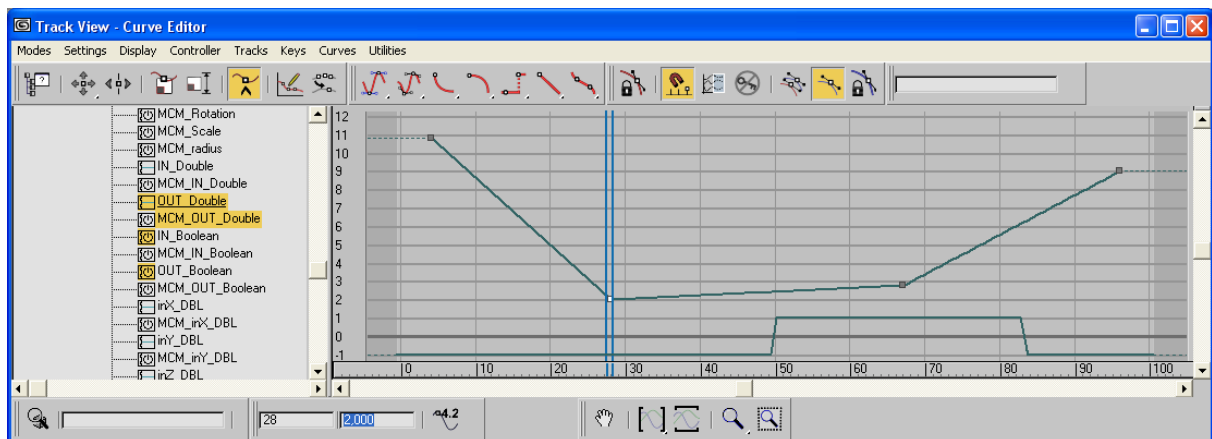


Abbildung 79: Beispielsituation für Right Manual Anchoring nach dem Schreiben ohne Anchoring

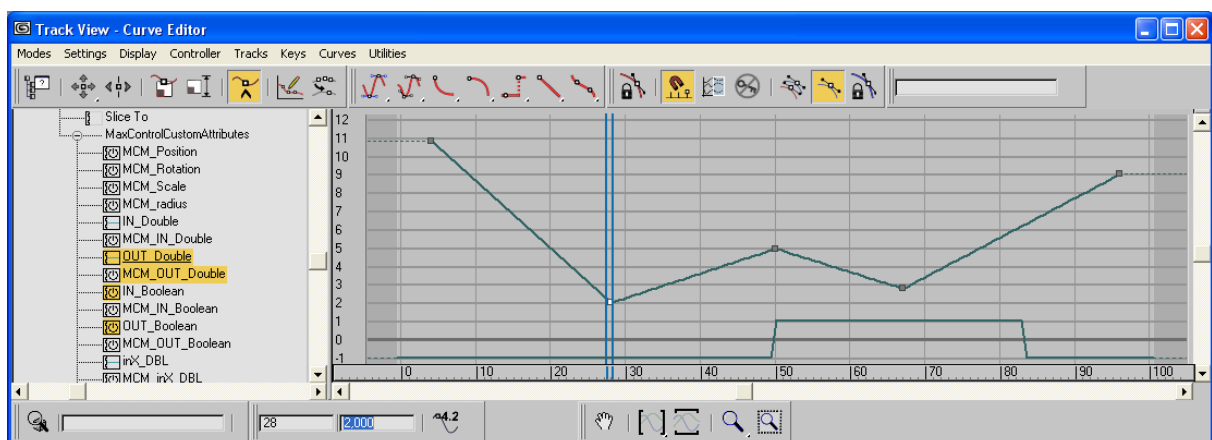


Abbildung 80: Beispielsituation für Right Manual Anchoring nach dem Schreiben mit Anchoring

Wird in der in Abbildung 78 gezeigten Situation im aktuellen Frame ein neuer Wert als Key geschrieben, dessen Wert vom vorher in diesem Frame gültigen Wert abweicht, so würden

durch die lineare Interpolation zwischen den Keys auch Werte im manuell kontrollierten Intervall verändert werden, obwohl das Simulationsprogramm diese Werte nicht verändern darf. Abbildung 79 zeigt einen solchen unerwünschten Fall. Fügt man nun stattdessen vorher am Anfang des manuell kontrollierten Intervalls einen zusätzlichen Key mit dem dort gültigen Wert ein und erzeugt erst dann den neuen Key im aktuellen Frame, so bleiben die Werte im manuell kontrollierten Intervall unverändert, wie die Abbildung 80 zeigt.

Ein solches Vorgehen wird hier als Anchoring bezeichnet, weil der erste Wert im manuell kontrollierten Intervall durch den zusätzlich gesetzten Key verankert wird und so das Intervall vor Veränderungen durch neue vorangehende Keys geschützt wird.

`rightManualAnchoringNeeded(f)`, `anchorRightManualValue(f)` :

Der Methodenaufruf `rightManualAnchoringNeeded(f)` bestimmt, ob der erste Wert in einem folgenden manuell kontrollierten Intervall durch Anchoring geschützt werden muss, und der Aufruf `anchorRightManualValue(f)` führt dieses Anchoring ggf. aus.

Die weniger stark optimierte Kommunikationsmethode `CommOptMPs` führt dagegen kein „Right Manual Anchoring“ durch. Bei Verwendung dieser Kommunikationsmethode durch einen SP-Typ muss der Benutzer den linken Rand eines manuell kontrollierten Intervalls selbst durch einen Key schützen. In der gegenwärtigen Version von `MaxControl` verwenden jedoch alle SP-Typen beim Schreiben die Kommunikationsmethode `CommOptKeys`.

Left Anchoring:

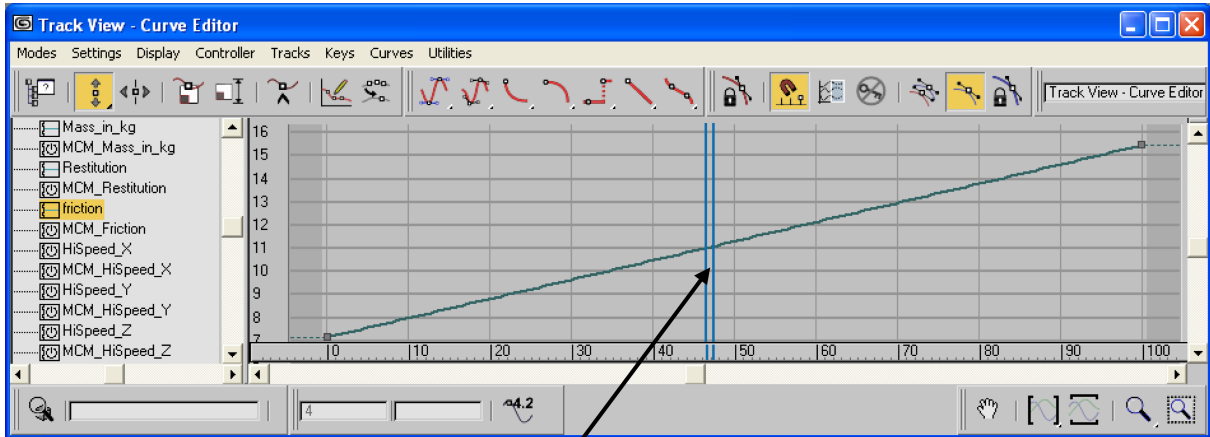
Es sei wieder die obige Situation aus Abbildung 78 gegeben. Wird nun der in Abbildung 79 gezeigte neue Key eingefügt, so hat dies auch sichtbare Seiteneffekte auf den Bereich vor dem aktuellen Frame.

Die Veränderung der Werte **vor** dem neuen Key ist ebenfalls unerwünscht, weil diese Werte Simulationsergebnisse darstellen, die auch weiterhin gültig bleiben sollen. Es wird daher eine entsprechende Gegenmaßnahme durchgeführt. Die Veränderung der Werte **nach** dem neuen Key außerhalb von manuell kontrollierten Intervallen wird jedoch vernachlässigt, da diese Werte aus der vorläufigen Szenenzustandsfolge stammen und nicht als manuell kontrolliert definiert wurden. Solche Werte müssen nicht erhalten bleiben und können durch die folgenden Simulationsschritte überschrieben werden.

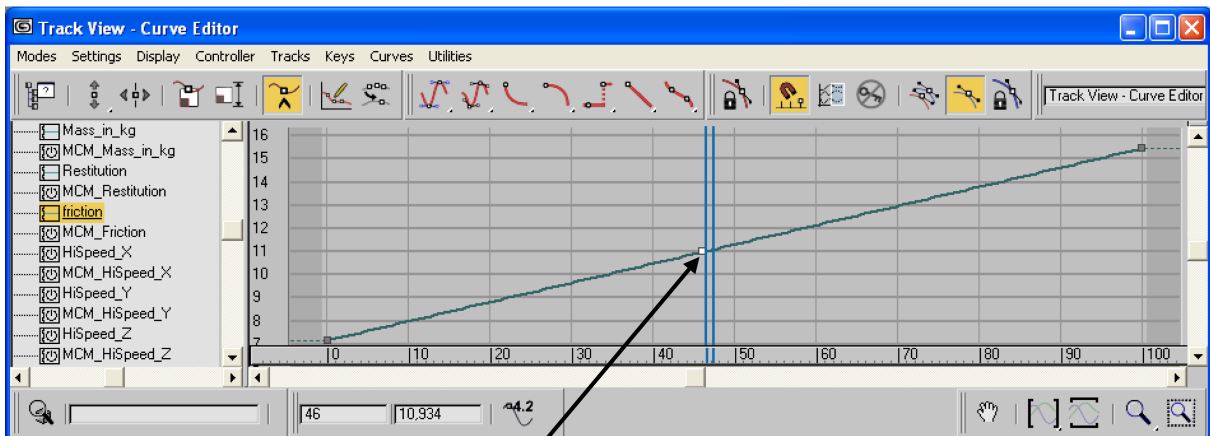
Um die Veränderung der Werte zu verhindern, die **vor** dem neu gesetzten Key liegen, wird überprüft, ob sich direkt vor dem neu zu setzenden Key bereits ein Key befindet. Ist dies nicht der Fall, so wird dort ein neuer Key mit dem bisher dort gültigen Wert gesetzt. Erst dann wird der neu zu setzende Key eingefügt. Die folgenden Abbildungen 81 sollen dieses Vorgehen erläutern. Durch dieses Vorgehen bleibt der vor dem neuen Key liegende Teil des Tracks unbeeinflusst, während der folgende Teil jedoch wie erwartet verändert wird. Dieses Anchoring schützt auch ein manuell kontrolliertes Intervall vor Veränderungen, das vor dem aktuellen Frame liegt. Der Methodenaufruf `leftAnchoringNeeded(f)` überprüft, ob das hier beschriebene Anchoring erforderlich ist, und der Methodenaufruf `anchorLastValue(f)` führt dieses Anchoring dann ggf. aus.

Insgesamt ist im Beispiel (Abbildungen 81) so erreicht worden, dass bei vielen Frames kein neuer Key gesetzt werden musste, bis es zu einer Abweichung von den bisher in der Szenenzustandsfolge gültigen Werten kam. Um dies zu erkennen, war im Beispiel aus den Abbildungen 81 auch nur ein Lesen der vorher existierenden zwei Keys aus der Szenenzustandsfolge nötig, da die Werte dazwischen durch Interpolation innerhalb von `MaxControl` bestimmt werden. Insgesamt wurden also viele Lese- und Schreibbefehle für die

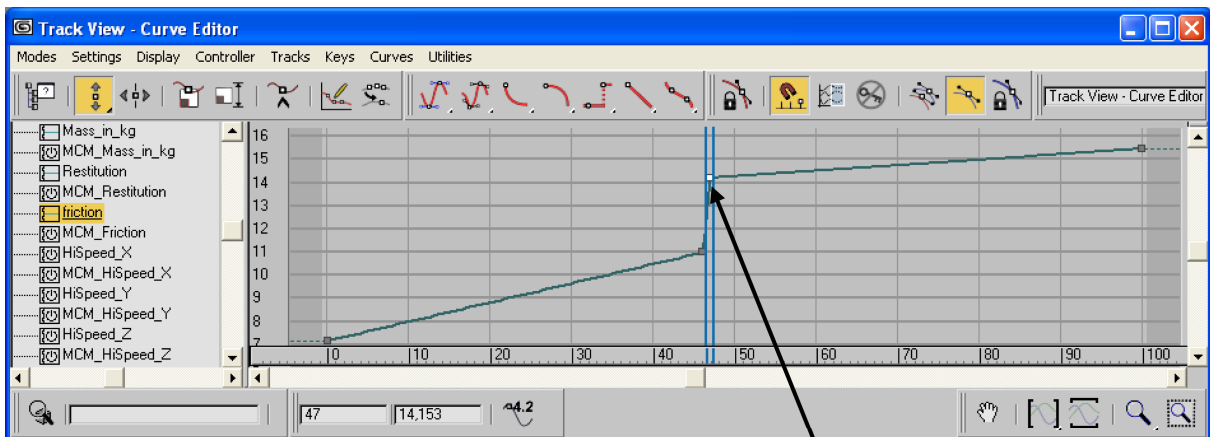
Kommunikation mit 3D-Studio-MAX eingespart, was sich positiv auf die Kommunikationsgeschwindigkeit auswirkt. Ferner wurden nur relativ wenige Keys in der Szenenzustandsfolge für diesen Track erzeugt, was zu einer effizienteren Nutzung der Speicherressourcen führt und die optimierte Kommunikation bei nachfolgenden Simulationen der aktuellen Szene unterstützt.



aktueller Frame



vorher zusätzlich eingefügter Key



neuer Key im aktuellen Frame

Abbildungen 81: Durchführung des Left Anchoring in einem Beispiel

Die weniger stark optimierte Kommunikationsmethode CommOptMPs erzeugt bei jedem Schreibvorgang einen neuen Key, es wird also von Frame zu Frame eine Folge von Keys erzeugt. Dadurch ist bei dieser Kommunikationsmethode ein „Left Anchoring“ für $f=n>a+1=MAXStartFrame+1$ nicht erforderlich, weil beim Setzen eines neuen Keys in Frame n das links davon liegende Intervall bereits vor Seiteneffekten dieses neuen Keys geschützt ist, da nach dem vorangehenden Simulationsschritt zu Frame $n-1$ bereits ein neuer Key in Frame $n-1$ gesetzt wurde. Dies ist jedoch nicht zwingend der Fall für $n=a+1$, da die Simulation alle SPs in Frame a als manuell kontrolliert ansieht und auch keinen Simulationsschritt zu Frame a durchführt. Daher ist das Schreiben der unveränderten aus Frame a gelesenen Werte zurück in diesen Frame a eigentlich unnötig. Damit könnte in Frame a dann aber ein Key fehlen, der den Wert einer SP in Frame a vor Veränderungen schützt, die auftreten könnten, wenn für diese SP ein Key in Frame $n=a+1$ gesetzt wird. Daher wird auch für Frame a die Methode `putValues(simFrame)` aufgerufen, damit die Kommunikationsmethode CommOptMPs auch in Frame a einen Key erzeugt.

Right Anchoring:

Die im Folgenden beschriebene Methode zur Optimierung der Kommunikation führt ein Anchoring durch, das auf der Annahme basiert, dass es sich positiv auf nachfolgende Kommunikationsschritte auswirkt. Dabei wird aufgrund empirischer Beobachtungen und Schätzungen davon ausgegangen, dass ausreichend häufig Situationen auftreten, in denen der Wert einer durch die Simulation beeinflussten SP für mehrere Frames konstant bleibt. Zudem wird angenommen, dass durch vorherige Simulationen nicht bereits zu viele Keys in diesen bei der aktuellen Simulation konstanten Bereichen gesetzt wurden. Dies kann durch nachträgliche Key-Reduktion für die vorangehenden Simulationen begünstigt werden.

Der simulierte Helligkeitsverlauf eines Blinklichtes kann beispielsweise so aussehen:

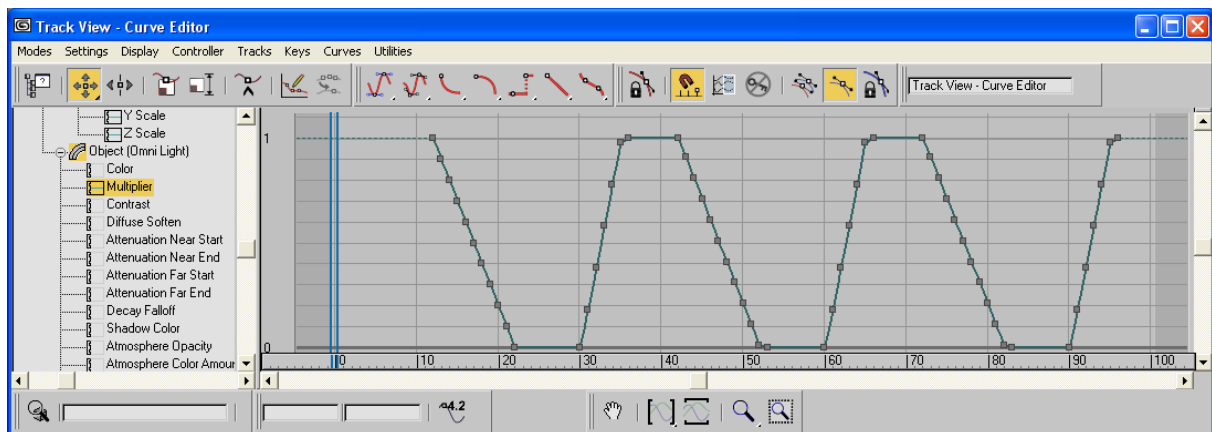
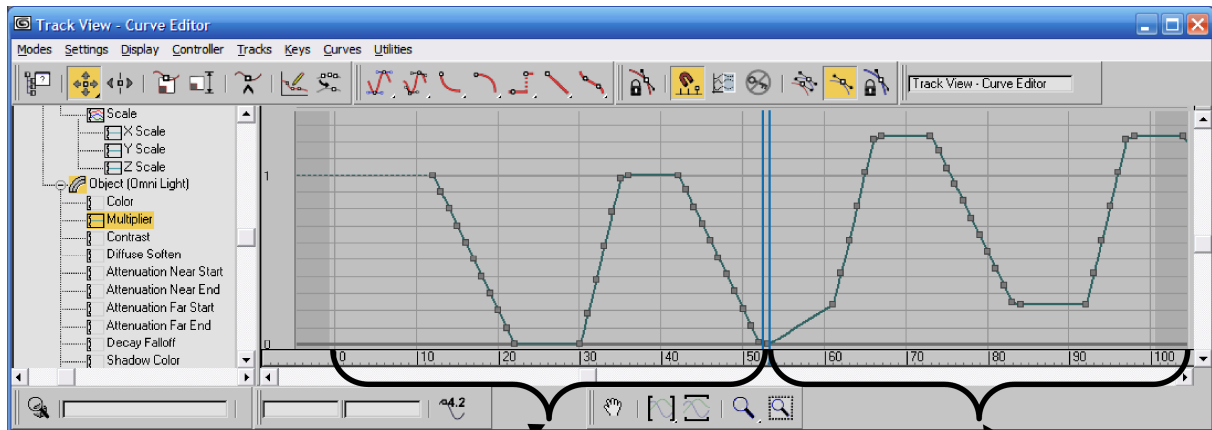


Abbildung 82: Helligkeitsverlauf eines Blinklichtes

Je nach Einstellung des Blinkintervalls werden sowohl die maximale Helligkeit als auch die minimale Helligkeit der Lampe über mehrere Frames konstant gehalten. Die Situation, dass ein Wert über mehrere Frames hinweg konstant bleibt, tritt hier also regelmäßig auf. Diese Situation ist besonders augenfällig, wenn das Blinklicht ausgeschaltet wird. Dann wird seine minimale Helligkeit über so viele Frames gehalten, wie das Blinklicht ausgeschaltet bleibt.

Es soll nun ein Mechanismus entwickelt werden, der das Setzen neuer Keys für solche Situationen optimiert, indem in den konstanten Bereichen so wenige Keys wie möglich erzeugt werden.

Für das in der folgenden Abbildung 83 gezeigte Beispiel gehen wir davon aus, dass sich bereits Keys einer vorangehenden Simulation derselben Lichtquelle in der Szene befinden. Allerdings sollen bei der vorangehenden Simulation die maximale und die minimale Helligkeit sowie die anfängliche Helligkeit der Lichtquelle höhere Werte gehabt haben. Dies wird in der Abbildung 83 eines entsprechenden Tracks illustriert:



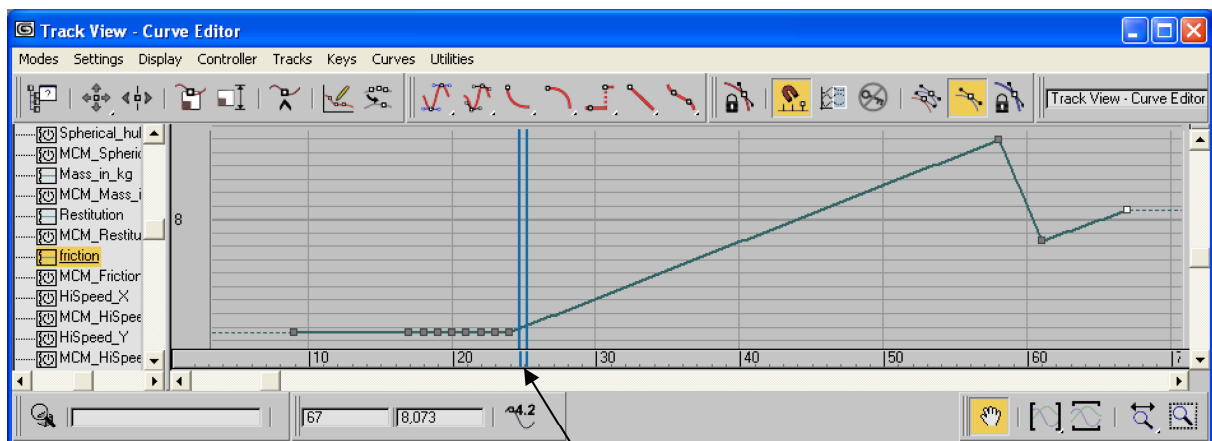
Keys aus aktueller Simulation

Keys aus vorangehender Simulation

Abbildung 83: Zwischenstand einer Simulation mit vorangehender Simulation

Es soll nun ermöglicht werden, das ab dem aktuellen Frame n durch die Simulation über mehrere Frames konstant gehaltene Minimum der Lichthelligkeit entsprechend in die Szenenzustandsfolge zu schreiben, ohne dabei in jedem Frame einen neuen Key setzen zu müssen.

Zur Erläuterung der für solche Fälle entwickelten Optimierung sei nun folgende Situation gegeben:



aktueller Frame

Abbildung 84: Beispielsituation für Right Anchoring vor dem Schreiben

Hier wurden bis vor dem aktuellen Frame Werte in die Szenenzustandsfolge geschrieben, die untereinander zwar alle gleich sind, die jedoch in den folgenden Frames vom bisher jeweils dort gültigen Wert abweichen werden, wenn weiterhin derselbe Wert wie in den vorangehenden Frames in die Szenenzustandsfolge geschrieben werden soll. Dann wird die Bedingung `valueChanged()` in allen Frames, die den gleichen Wert wie die vorangehenden Frames erhalten sollen, entscheiden, dass ein neuer Key gesetzt werden muss.

Würde man nun aber vorher den nach dem aktuellen Frame momentan nächsten Key bereits auf diesen Wert setzen, so entstünde folgende Situation:

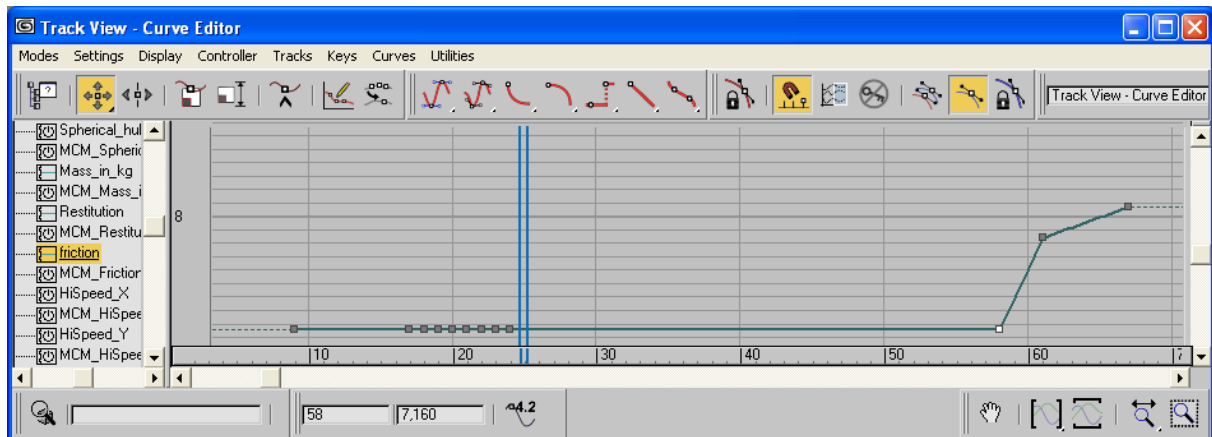


Abbildung 85: Beispielsituation nach der Durchführung des Right Anchoring

Unter der Voraussetzung, dass sich die Werte der SP für längere Zeit nicht ändern, müsste bis maximal zu dem veränderten Key kein neuer Key gesetzt werden, was für diesen Bereich des Tracks die Kommunikation vollständig einsparen würde.

Bei dem Beispiel aus Abbildung 84 führt allein das Anchoring schon dazu, dass nun auch der Wert im aktuellen Frame bereits der gewünschte Wert ist. Dies ist jedoch nicht zwingend der Fall, wie das folgende durch einen zusätzlichen Key veränderte Beispiel mit gleicher Zielsetzung zeigt:

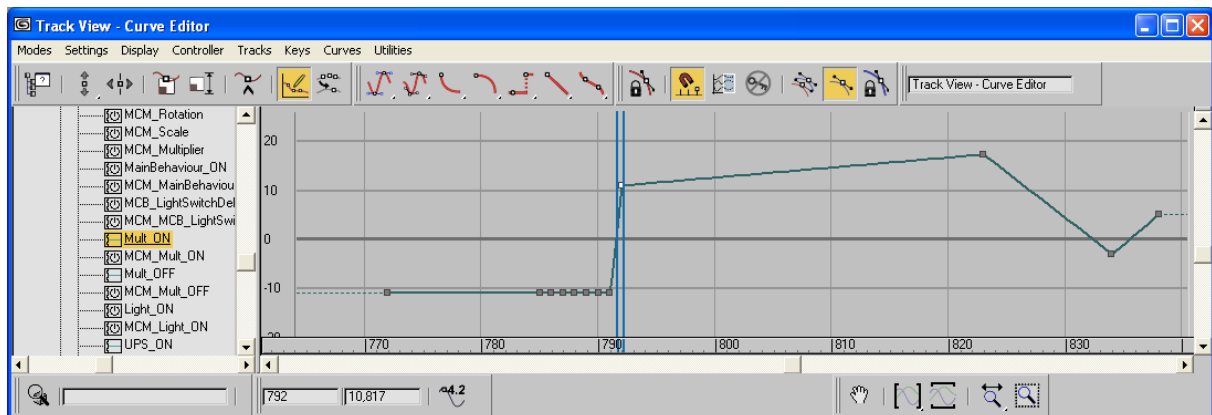


Abbildung 86: Key im aktuellen Frame bei Right Anchoring, Situation vor dem Schreiben

Befindet sich im aktuellen Frame bereits ein Key, so weicht sein aktueller Wert vom gewünschten neuen Wert ab, denn die Bedingung `valueChanged()` wurde vorher als `true` ausgewertet.

Führt man nun zunächst die Anpassung des momentan nächsten Keys durch und setzt auch für den Key im aktuellen Frame den gewünschten neuen Wert so ergibt sich die folgende Situation:

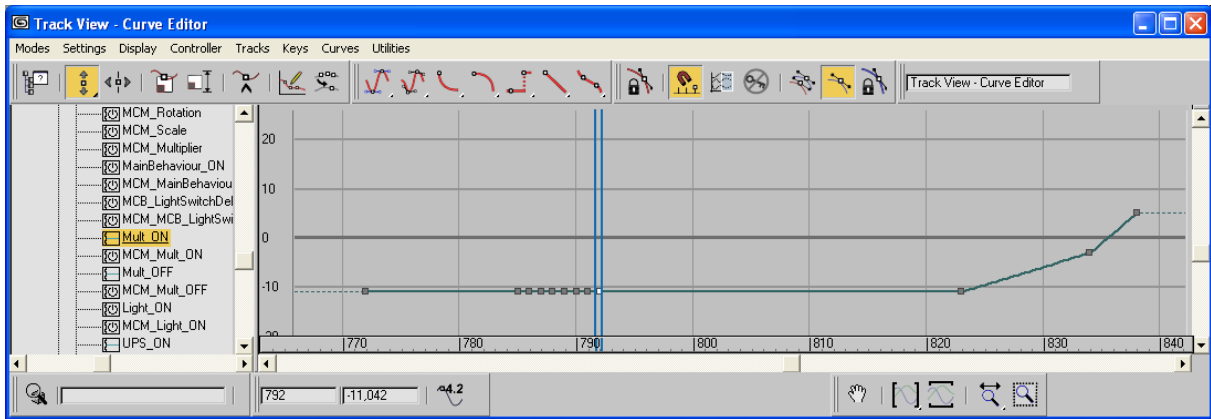


Abbildung 87: Key im aktuellen Frame bei Right Anchoring, Situation nach der Korrektur

Das Right Anchoring wird unter bestimmten Bedingungen ausgeführt, die auf praxisbezogenen Annahmen beruhen. Diese Bedingungen werden von `rightAnchoringNeeded(f)` getestet.

Zunächst müssen mindestens zwei Werte der SP hintereinander denselben Wert haben. Erst dann kann man annehmen, dass dieser Wert auch in einigen folgenden Frames beibehalten wird, so dass sich das Anchoring positiv auswirkt.

Zusätzlich muss der momentan nächste Key **mehr** als einen Frame vom aktuellen Frame entfernt sein, da dieses Anchoring sonst zu häufig ausgeführt werden könnte, ohne dass es einen günstigen Effekt hätte. Dies soll durch die folgende Darstellung eines Beispiels erläutert werden, bei dem angenommen wird, dass bei der aktuellen Simulation genau derselbe Werteverlauf wie bei der vorangehenden Simulation erzeugt wird:

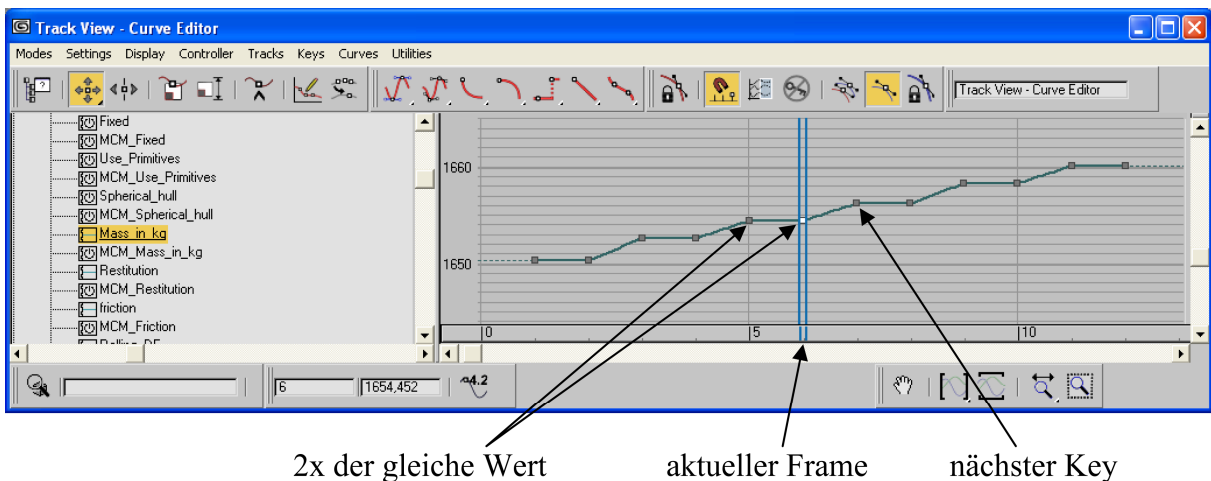


Abbildung 88: Bedingung für Right Anchoring, Situation vor dem Schreiben

Würde diese Form des Anchoring im Beispiel ohne Beachtung der letztgenannten Bedingung durchgeführt, so würde hier alle 2 Frames ein unnötiges Anchoring entstehen:



unnötiges Anchoring

Abbildung 89: Bedingung für Right Anchoring, Situation nach unnötigem Anchoring

Dieses unnötige Anchoring würde durch das Setzen des korrekten Wertes im nächsten Frame sofort wieder rückgängig gemacht werden.

Weiterhin wird überprüft, ob sich der Wert des momentan nächsten Keys von dem im Frame f zu setzenden Wert unterscheidet. Ist dies **nicht** der Fall, so hat der momentan nächste Key schon den für dieses Anchoring erforderlichen Wert und der MAXScript-Befehl zum Ändern dieses Keys in dem Track der betrachteten SP kann dann eingespart werden. In diesem Fall wird dieses Anchoring übersprungen.

Das Anchoring selbst wird nun ggf. von `anchorRightValue()` durchgeführt.

Ist die Bedingung `lastkey1.frame=f` erfüllt, d.h. befindet sich im aktuellen Frame ein Key, müssen sich die Aufrufe `sendValue()` und `setLastKey(f)` anschließen, um diesen Key wie beim Wechsel von Abbildung 86 auf Abbildung 87 aus den dort genannten Gründen auf den gewünschten Wert zu setzen. Anderenfalls kann das Setzen eines Keys in Frame f wie in Abbildung 85 entfallen, so dass `sendValue()` und `setLastKey(f)` nicht aufgerufen werden müssen.

Es folgt die genaue Erläuterung von `sendValue()` und `setLastKey(f)`.

Durch die Methodenaufrufe `sendValue()` und `setLastKey(f)` wird im aktuellen Frame nach den eventuell vorangehenden Anchoring-Maßnahmen der neue Wert der SP in den aktuellen Frame der Szenenzustandsfolge geschrieben. `sendValue()` erzeugt an der aktuellen Stelle im Track einen Key mit dem in `value` gespeicherten Wert. Der folgende Methodenaufwurf `setLastKey(f)` sorgt dafür, dass die Felder `lastKey1`, `lastKey2` und `nextKey` entsprechende, eventuell veränderte Werte erhalten. Hat vorher eines der oben genannten Anchoring-Verfahren neue Keys in den Track der betrachteten SP eingefügt, so hat dieses bereits selbst die Variablen `lastKey1`, `lastKey2` und `nextKey`, soweit betroffen, auf entsprechende Werte gesetzt, so dass `setLastKey(f)` jetzt nur noch das eventuelle Setzen eines neuen Keys durch `sendValue()` berücksichtigen muss.

Die in diesem Kapitel bezüglich der Kommunikationsoptimierung nicht näher dokumentierte Klasse `MCP_Double`, die eine Unterklasse der hier dokumentierten Klasse `MC_PropertyCommOptimized` ist und durch die Double-SPs repräsentiert werden, kann auf ähnliche Weise auch nicht-waagerechte lineare Zusammenhänge zwischen bisher gesetzten Keys erkennen. Sie führt dann ggf. analog ein Anchoring durch, welches für

möglichst viele folgende Frames ein Setzen neuer Keys unnötig macht, wenn der erkannte lineare Zusammenhang beibehalten wird.

Insgesamt hat sich die Kommunikationsoptimierung CommOptKeys als sehr erfolgreich erwiesen, da im Durchschnitt eine Verzehnfachung der Ausführungsgeschwindigkeit gemessen werden konnte und die Szenendateien signifikant kleiner wurden. Besonders die letzte beschriebene Methode „Right Anchoring“ führt zu günstigen Ergebnissen, wenn sich möglichst wenig Keys für die betrachtete SP in der Szenenzustandsfolge befinden. Ein zusätzliches Durchführen einer Key-Reduktion vor einer erneuten Simulation kann daher positive Auswirkungen auf die optimierte Kommunikation haben, denn die hier beschriebene optimierte Kommunikation kann überflüssige Keys nicht direkt löschen, dies ist nur über eine nachfolgende Key-Reduktion möglich.

10.3.2 Kommunikationsmethoden zur Sicherung der Skalierbarkeit

Der in Kapitel 10.3.1.2 erläuterte Umweg über Dateien sowohl bei der Erstellung der auszuführenden MAXScript-Befehle als auch bei der Zwischenspeicherung der dabei eventuell gelesenen Daten, die dann von MaxControl gelesen werden, wird durch Instabilitäten der beteiligten Programmkomponenten nötig, die bei einer direkten Kommunikation über Zeichenketten zu Programmabstürzen führen, wenn die so in einem Kommunikationsschritt ausgetauschte Datenmenge zu groß wird. Vergleichbare Probleme gelten auch für die Kommunikation mit der in Shockwave implementierten Komponente zur Simulation von Mechanik. Da diese Komponente zusammen mit dem Hauptprogramm auf einer Internet-Seite läuft, fehlen ihr aus Sicherheitsgründen die Rechte für lesenden und schreibenden Zugriff auf Dateien. Hier können also keine Dateien als Alternative verwendet werden.


Die von Shockwave zurück an MaxControl gesendete Datenmenge führte bisher zu keinen Problemen, es musste lediglich die in einem Kommunikationsschritt von MaxControl an Shockwave in Form von Zeichenketten gesendete Datenmenge reduziert werden. Der Länge der an 3D-Studio-MAX gesendeten MAXScript-Programme sind auch dann Grenzen durch Programminstabilitäten gesetzt, wenn sie bereits in Form von Dateien und nicht als Strings an 3D-Studio-MAX übergeben werden. Lange Skripte müssen daher unter Umständen in mehrere kleinere Dateien aufgeteilt werden, die dann in mehreren Kommunikationsschritten an 3D-Studio-MAX gesendet werden. Dagegen konnten bisher beliebig lange Datensätze in Form von Dateien in einem Kommunikationsschritt zurück an MaxControl übermittelt werden.

Sowohl bei der Kommunikation mit Shockwave als auch mit 3D-Studio-MAX stellte sich also das Problem, in jedem Kommunikationsschritt, in dem MaxControl Daten an eine dieser Komponenten sendet, eine bestimmte Datenmengengrenze nicht zu überschreiten, da es sonst zu Programmabstürzen kommt. Andererseits darf die Datenmenge auch nicht zu klein werden, da sich in der Praxis wie in Kapitel 10.3.1.2 zu 3D-Studio-MAX erläutert gezeigt hat, dass die Kommunikation einer bestimmten Datenmenge erheblich schneller abläuft, wenn nicht kleine Teile dieser Datenmenge in vielen Kommunikationsschritten sondern möglichst große Teile dieser Datenmenge in entsprechend weniger Kommunikationsschritten übermittelt werden. Dies gilt auch für die Kommunikation mit Shockwave. Die Instabilitäten, welche solche Vorgehensweisen nötig machen, liegen dabei nicht in dem in Java implementierten Hauptprogramm MaxControl, sondern in einer oder mehrerer der Komponenten, die an der Kommunikation beteiligt sind. Dies sind 3D-Studio-MAX, die OLE-Kommunikation des verwendeten Betriebssystems Windows, der Internet-Explorer, sowie Shockwave.

Sowohl für die Kommunikation an Shockwave als auch an 3D-Studio-MAX wurde in Tests empirisch eine maximale Länge bestimmt, die Befehlsgruppen in einem Kommunikationsschritt haben durften, ohne dass es zu Instabilitäten kam. Diese Längen wurden dann aus Sicherheitsgründen mindestens halbiert. So wurde für 3D-Studio-MAX eine maximale Befehlsdateilänge von 1 MB festgelegt, für Shockwave 128 KB. Es werden nun jeweils so lange zu übermittelnde Befehle gesammelt, bis diese aus Gründen des Programmablaufs gesendet werden müssen oder die jeweilige Maximallänge überschritten ist. Dann wird die Befehlsgruppe gesendet, im Zielprogramm ausgeführt und die von ihr eventuell zurückgegebenen Daten werden gelesen und für eine spätere Verarbeitung gesammelt. Das Überschreiten der Maximallänge wird nur nach dem Zusammenstellen syntaktisch nicht trennbarer Befehlsgruppen überprüft, da z.B. Schleifen nicht in der Mitte geteilt werden dürfen um diese Teile einzeln auszuführen.

Die genannten Stabilitätsprobleme konnten so vollständig ausgeschlossen werden, wobei gleichzeitig eine effiziente Kommunikation beibehalten werden konnte.

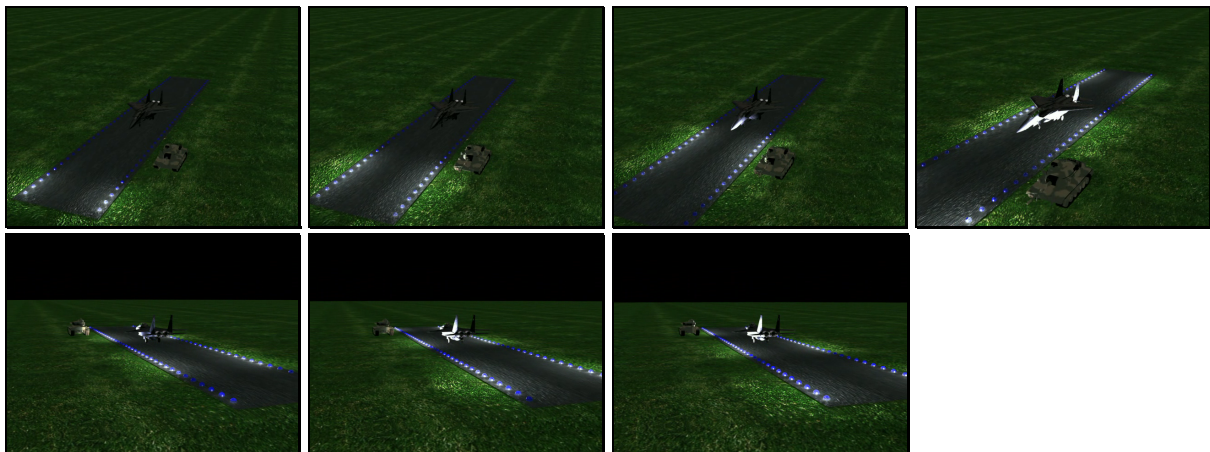
10.4 Praktische Anwendungen des entwickelten Werkzeugs

Um die Verwendbarkeit des entwickelten Werkzeugs in der Praxis und dessen Flexibilität bei der Lösung verschiedener Probleme aus unterschiedlichen Themengebieten zu testen, wurden mehrere Szenen zu verschiedenen Themen mit zugehörigen Objekttypen und Verhaltensweisen entwickelt. Diese Szenen wurden dann durch Simulation mit MaxControl animiert. Es folgen einige Beispiele hierfür in kommentierten Bildern aus den so produzierten Filmen, die auch auf der beiliegenden DVD zu finden sind. Die angegebenen Pfade zu den entsprechenden Filmdateien sind hier mit einem -Symbol gekennzeichnet.


10.4.1 Animationsfilme zum Thema „Visuelle Effekte in Spielfilmen“

In diesem Kapitel behandelte Filme wurden zu Themen und Problemstellungen produziert, die sich häufig bei der Herstellung visueller Effekte in Spielfilmen stellen. Die entwickelten Lösungen eignen sich auch für Spielfilme, die vollständig auf Computeranimationen basieren. Da diese Bereiche aktive Forschungsschwerpunkte im Bereich der Computeranimation sind, sollte die Tauglichkeit von MaxControl für entsprechende Aufgabenstellungen getestet werden.

10.4.1.1 Automatisierte Ansteuerung vieler Lichtquellen

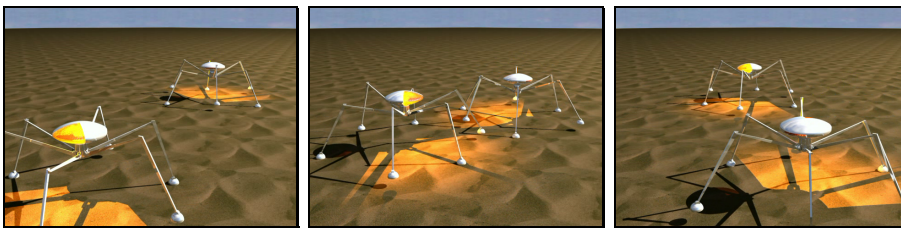


Abbildungen 90: Bilder zu „Automatisierte Ansteuerung vieler Lichtquellen“


 Movies\RunwayTest_2.avi


Die Animation zeigt eine Landebahn mit animierten Lichtern, welche die Landebahn markieren und durch periodisches Blinken einen Lauflichteffekt erzeugen, der entlang dieser Landebahn verläuft. Jeder Lichtquelle wurde derselbe Objekttyp `MCD_LightBlinkingDelayed` zugeordnet. Seine Verhaltensweisen `MCB_LightSwitchDelayed` und `MCB_IntervalSwitcher` können die zugehörige Lichtquelle in periodischen Abständen blinken lassen, wobei sich die Helligkeit der Lichtquelle in wählbaren Geschwindigkeiten ändert. Die Landebahn selbst hat den Objekttyp `MCD_Runway`. Seine Verhaltensweise `MCB_RunwayLights` steuert die Verhaltensweisen aller Lichtquellen, die sich auf dieser Landebahn befinden. So lässt sie das Blinken aller Lichtquellen entlang ihrer Position auf der Landebahn jeweils mit einer Zeitverzögerung gestaffelt beginnen, so dass sich ein Lauflichteffekt ergibt. Auf diese Weise konnte der Objekttyp für die Lichtquellen zusammen mit seinen Verhaltensweisen mehrfach wieder verwendet werden und es mussten insgesamt nur zwei Objekttypen und zugehörige Verhaltensweisen entwickelt werden. Die Problemstellung dieser Animation wirkt zunächst einfach, jedoch wäre schon diese Animation mit herkömmlicher Keyframe-Technik nur schwer durchführbar, weil man dann für den zeitlichen Verlauf der Helligkeit jeder einzelnen Lichtquelle eine eigene Folge von Animations-Keys erzeugen müsste (siehe dazu auch Kapitel 3.5).

10.4.1.2 Animation sechsbeiniger Roboter



Abbildungen 91: Bilder zu „Animation sechsbeiniger Roboter“

 Movies\Walker_15_newAudioV3.avi

 Movies\Walker_many_2_plusSFX_V2.avi

Hier wurden Roboter mit jeweils sechs Beinen simuliert. Sie bestehen aus mehreren Einzelteilen. Der Kopf, der Rumpf und die Füße haben jeweils eigene OTs, wobei alle Füße denselben OT verwenden. Dies zeigt die Vorteile kompositioneller Wiederverwendung auf der Ebene der 3D-Objekte. Das grundsätzliche Verhalten eines Fußes musste nur einmal spezifiziert werden und konnte dann für alle Füße verwendet werden. Die Füße erhalten Befehle vom Rumpf des jeweiligen Roboters.

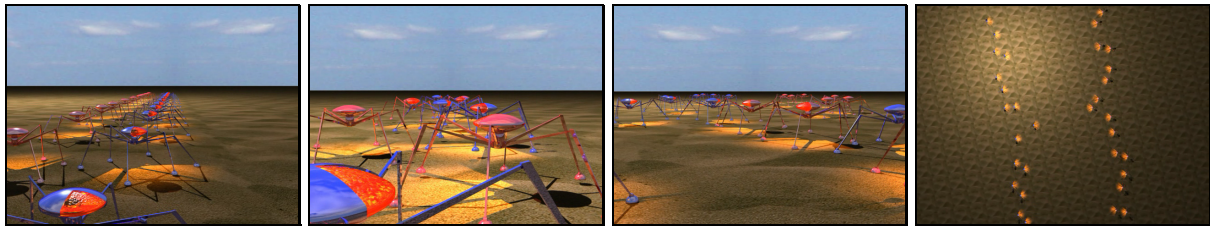
An den Köpfen der Roboter wurde als optischer Effekt eine Art Scannerlicht realisiert. Die Java-Klassenvererbung wurde hier genutzt, um das Verhalten der Landebahn aus dem ersten Beispiel in einer neuen erfindenden Klasse anzupassen. Es werden wieder mehrere Lichtquellen so angesteuert, dass ein Lauflichteffekt entsteht, wobei diesmal jedoch der Laufeffekt abwechselnd von rechts nach links und dann von links nach rechts geht.

Durch die Implementation des Verhaltens passender tonerzeugender Objekte wurden auch Toneffekte für die Bewegungen des Roboters und für das Scannerlicht automatisch erzeugt.


Die Bewegungen der Roboter basieren hier nicht auf der Simulation von Mechanik, deren Simulation in MaxControl zwar möglich aber nicht erzwungen ist. Durch die enge Integration in das Animationswerkzeug 3D-Studio-MAX konnte dessen Automatik zur Berechnung inverser Kinematik für die Bewegung der Beine genutzt werden, wobei als Basis die von MaxControl gesteuerten Bewegungen der Füße und des Rumpfes verwendet wurden.


Die so spezifizierten Roboter können miteinander interagieren. Sie können auf Zielpunkte in der Szene automatisch zulaufen und sich dabei gegenseitig ausweichen.

10.4.1.3 Simulation einer größeren Anzahl von Robotern



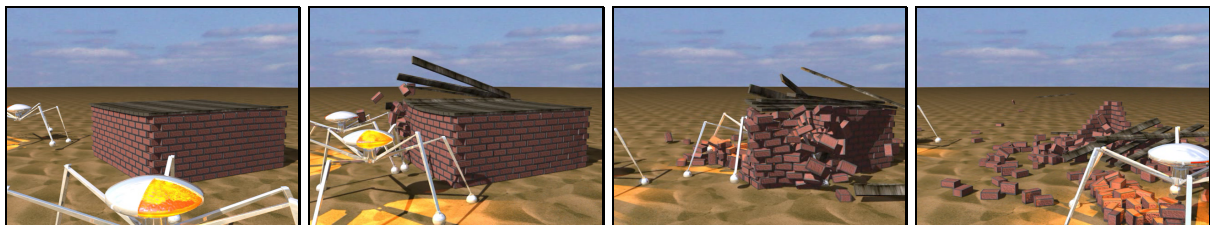
Abbildungen 92: Bilder zu „Simulation einer größeren Anzahl von Robotern“

 Movies\Walker_many_6_AudioV2.avi


 Movies\Walker_many_4.avi

Da MaxControl insbesondere die Animation von Szenen mit sehr vielen Objekten erleichtern soll, wurden in dieser Simulation viele der Roboter aus dem vorangehenden Beispiel in einer Szene simuliert. Sie stehen sich zunächst in zwei Reihen gegenüber und sollen Zielpunkte hinter der jeweils gegenüberliegenden Reihe von Robotern erreichen. Dabei müssen sie korrekt aneinander vorbeilaufen, ohne sich gegenseitig zu berühren. Die erste für dieses Beispiel angegebene Filmdatei zeigt die Roboter aus verschiedenen Kameraperspektiven, während sie im zweiten angegebenen Film nur sehr einfach gerendert sind und aus nur einer Perspektive gezeigt werden, wodurch die genauen Ausweichbewegungen der Roboter klarer zu erkennen sind.

10.4.1.4 Simulation von Mechanik auf der Basis der Animation sechsbeiniger Roboter



Abbildungen 93: Bilder zu „Simulation von Mechanik auf der Basis der Animation sechsbeiniger Roboter“

 Movies\Walker_plusPhysics_1_Sound_V3.avi

Diese Animation basiert auf der Simulation von zwei sechsbeinigen Robotern aus den vorangehenden Beispielen. Hier sollte gezeigt werden, wie sehr die Simulation von Mechanik zum Realismus und zur Glaubwürdigkeit einer Computeranimation beitragen kann. Da zum Zeitpunkt der Erstellung dieser Animation die Mechanik-Simulationskomponente von

MaxControl noch nicht implementiert war, wurde für eine nachträgliche Simulation von Mechanik das Plugin „Reactor“ für 3D-Studio-MAX verwendet. Entsprechend den vorher durch MaxControl simulierten Bewegungen der zwei Roboter konnte dieses Plugin die Bewegungen der Steine und Bretter eines Gebäudes durch Simulation von Mechanik automatisch erzeugen, so dass die Mauern und die Decke des Gebäudes realistisch einstürzen. Die Toneffekte für das Laufen der Roboter wurden wieder durch MaxControl automatisch erzeugt, während die Toneffekte für das Einstürzen der Mauern nachträglich in einem Videoschnittprogramm hinzugefügt wurden. Dies zeigt, dass eine mit MaxControl erstellte Animation anschließend mit denselben Werkzeugen und Techniken nachbearbeitet werden kann wie eine herkömmliche manuell durch Keyframes erstellte 3D-Animation. Da sich eine von MaxControl erstellte Animation technisch nicht von manuell erstellten Animationen unterscheidet, kann sie durch weitere Simulationswerkzeuge wie Reactor oder z.B. durch Werkzeuge zur Simulation von Textilien oder Haaren sowie durch manuelle Animationstechniken nachbearbeitet werden. Ebenso kann die von MaxControl automatisch erstellte Tonspur wie jede andere Tonspur nachbearbeitet werden. Man kann also durch MaxControl zwar endgültige Animationsergebnisse erzielen, sie können jedoch auch nachträglich auf verschiedenste Weise nachbearbeitet werden, was dem Benutzer weitere Freiheiten bietet.

10.4.1.5 Simulation eines Autorennens



Abbildungen 94: Autorennen: Kameraperspektiven




Abbildungen 95: Autorennen: Lichteffekte



Abbildungen 96: Autorennen: Mechaniksimulation

Um die Eignung von MaxControl für die Animation eines Autorennens zu testen, wurden mehrere Simulationen entwickelt. Einige der daraus produzierten Filme sind im folgenden Verzeichnis der beigelegten DVD zu finden:

 Movies\CarRace

Bei diesen Simulationen fahren mehrere Fahrzeuge selbstständig ein Rennen und können dabei mechanisch miteinander und mit der Umgebung interagieren (siehe Abbildungen 96). So können die Fahrzeuge über Sprungschancen fahren oder Gegenstände umwerfen. Die Fahrzeugmodelle wurden von einem Künstler⁶⁷ entworfen, ihre Einzelteile mit MaxControl-Verhaltensweisen ausgestattet und so animiert. Die Fahrzeuge haben komplexe Verhaltensweisen, die ihnen zum einen die Fähigkeit geben, erfolgreich entlang einer vorgegebenen Strecke zu fahren, die in den Filmen durch grüne oder hellgraue rechteckige Felder markiert ist, die transparent über dem Asphalt erscheinen. Die Verhaltensweisen ermöglichen den Fahrzeugen weiterhin, auf die jeweils anderen Fahrzeuge zu reagieren um ihnen bei Gefahr auszuweichen oder sie, wenn möglich, zu überholen. Damit das Rennen spannend bleibt, dürfen nur Fahrzeuge, die zurzeit nicht den ersten Platz im Rennen belegen, ihre Düsentriebwerke einsetzen. Die Schubkraft dieser Triebwerke darf bis zu einem Maximalwert umso stärker sein, je weiter das Fahrzeug in seiner Platzierung zurückliegt. So können zurückliegende Fahrzeuge die besser platzierten Fahrzeuge trotz gleicher Motorleistungen überholen.


Diese Aufgaben sind nicht trivial, da sich die Autos in einer mechanisch realistisch simulierten Umgebung befinden und mit entsprechenden Problemen umgehen müssen. So müssen die Fahrzeuge selbst darauf achten, dass ihre Räder nicht durch zu starke Drehmomente des Motors, zu starkes Bremsen oder zu hohe Kurvengeschwindigkeiten die Bodenhaftung verlieren. Vor Kurven müssen sie ggf. rechtzeitig mit dem Bremsmanöver beginnen, um zu Beginn der Kurve eine angemessene Geschwindigkeit zu haben, die aus der Kurvenkrümmung und der Bodenhaftung abgeleitet werden muss. Dies zeigt, dass die erforderlichen Verhaltensweisen von Objekten relativ komplex werden müssen, wenn die 3D-Objekte ihre Aufgaben in einer mechanisch realistisch simulierten Welt erfüllen müssen. Jedoch entstehen durch die Simulation von Mechanik komplexe realistische Bewegungen, die diesen Aufwand rechtfertigen. Fahren so simulierte Autos z.B. in einem Rennen mehrere Runden auf einer Strecke, so entstehen im Allgemeinen dennoch keine Wiederholungen von Zuständen der Szenenzustandsfolge, da schon kleinste Abweichungen aufgrund der komplexen Simulation von Mechanik den Verlauf des Rennens gravierend beeinflussen können.

Auch die Toneffekte wurden wieder durch MaxControl automatisch erzeugt. Dabei wurden auch Daten der Simulation von Mechanik verwendet, um z.B. Aufprall- und Roll- und Schleifgeräusche objekt- und materialabhängig erzeugen zu können. Das Quietschen von Reifen wurde unter anderem so generiert.

Die Fahrzeuge verfügen auch über eine automatische Lichtsteuerung, so dass die angesteuerten Lichtquellen z.B. ihre Blinker, Bremsleuchten, Rückfahrleuchten, volumes Licht bei den Düsentriebwerken und bei Bremsscheiben, die beim Abbremsen von hohen Geschwindigkeiten rot glühen, sowie ein Lauflicht vorne am Auto bilden können. Diese Lichtquellen reagieren automatisch auf die gegebenen Situationen. Das Lauflicht wird über denselben Objekttyp gesteuert wie das Lauflicht bei den Robotern aus Kap. 10.4.1.2. Da es

⁶⁷ 21st Century Solutions Ltd., 3D Special Service, Suite 31, Don House, 30-38 Main Street, Gibraltar, <http://www.3-d-models.com/images/104m301.htm>, 04.07.2007

sich bei diesen Lichteffekten nicht um Textur- oder Materialveränderungen sondern um „echte“ Lichtquellen handelt, können diese auch einen Lichtschein mit Schatten auf die Umgebung werfen (siehe Abbildungen 95). Der im Folgenden genannte Film zeigt die automatische Aktivierung der Scheinwerfer eines Fahrzeugs, sobald es in einen Tunnel fährt:

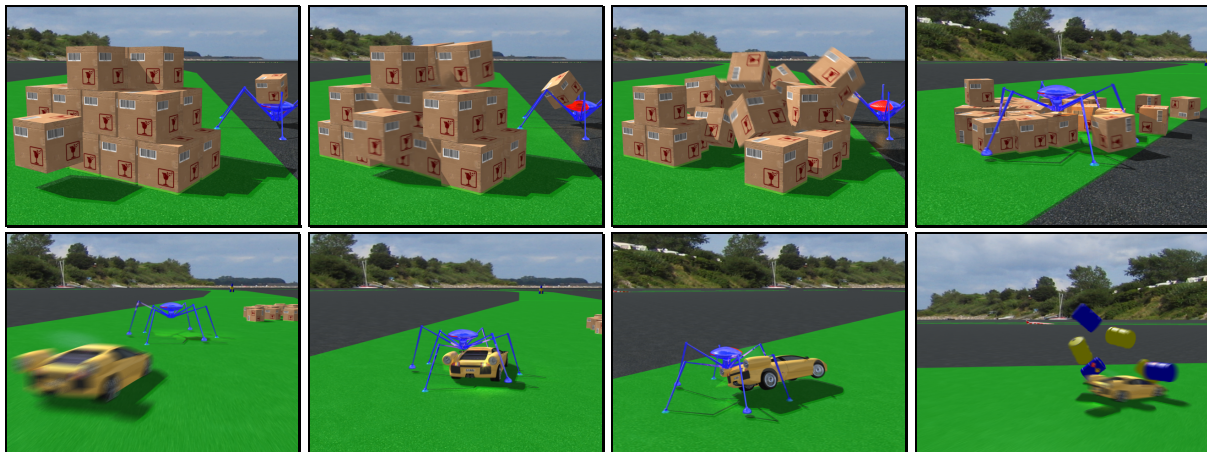
: Movies\CarRace\TunnelLightTest_V2_softwareRendering.avi

Auch die Kamera wird durch Verhaltensweisen automatisch gesteuert. So kann sie sich wie eine Handkamera verhalten, inklusive eines Bildwackelns je nach Stärke der gewählten Vergrößerung, oder wie eine fest stehende Kamera, oder sie kann an festen Punkten der Karosserie der Fahrzeuge angebracht sein. Welche der oben genannten Verhaltensweisen genutzt wird, wird mit Hilfe von Zufallswerten und situationsabhängig automatisch von der Kamera bestimmt, und wechselt in der Regel mehrfach während des Rennens. Werden die Fahrzeuge von außen aus der Entfernung gezeigt, so wählt die Kamera automatisch aus vorgegebenen „Kamerapunkten“ ihren Standort aus, von dem aus sie das Fahrzeug aufnimmt. Sie verfolgt automatisch immer das Fahrzeug, das im Rennen zurzeit den ersten Platz einnimmt (siehe Abbildungen 94).

Der im Folgenden genannte Film zeigt den Einsatz von Zeitlupe und Zeitraffer über die Veränderung des in Kap. 8.1.2 und 10.2.2 erläuterten Parameters `timePerSecond`. Allein die manuell kontrollierte Animation dieses Parameters beeinflusst die Ablaufgeschwindigkeiten der Verhaltensweisen, der Mechaniksimulation und der Tonerzeugung:

: Movies\CarRace\SlowMoFastMoTest_2_SFX_V2.avi

10.4.1.6 Fahrzeug und Roboter in einer Simulation



Abbildungen 97: Bilder zu „Fahrzeug und Roboter in einer Simulation“

: Movies\ManualPSimControlTest_2_SFX_V2.avi

Der Roboter aus Kap. 10.4.1.2 bis 10.4.1.4 wurde mit seinen OTs und VHs entwickelt, bevor MaxControl über eine eigene Komponente zur Simulation von Mechanik verfügte und bevor die Autorennsimulationen aus Kap. 10.4.1.5 entwickelt wurden. Durch die flexible und erweiterbare Struktur von MaxControl war es dennoch möglich, beide Simulationen zu vereinen. Damit der Roboter teil der Mechaniksimulation wird, wurde seinen passiven Teilen, die bisher keine eigenen VHs hatten, jeweils der OT „MCD_PhysicalBody“ zugewiesen. Aktiven Teilen wie dem Rumpf, dem Kopf und den Füßen, die sich über direkte Positions-

und Rotationsveränderungen selbst bewegen, wurden Unterobjekte mit der gleichen dreidimensionalen Form hinzugefügt, denen ebenfalls der OT „MCD_PhysicalBody“ zugewiesen wurde. In beiden Fällen wurde deren Unter-VH „MCB_Physics“ ausgeschaltet, so dass die Bewegungen des Roboters nun für die Mechaniksimulation erzwungen werden, wie in den Kapiteln 9.5 und 10.2.2 erläutert wurde.

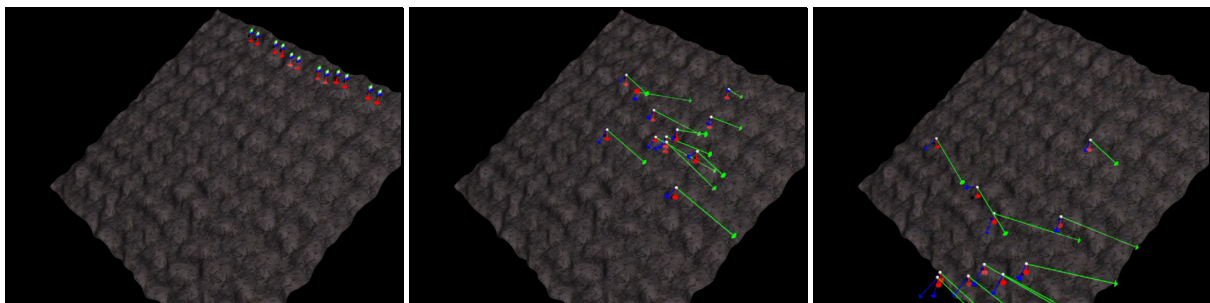
In der hier gezeigten Simulation wird auch die manuelle Kontrolle der Mechaniksimulation demonstriert, indem der Karton im Vordergrund zu Beginn der Animation manuell so kontrolliert wird, dass er die dahinter gestapelten Kartons durch eine langsame manuell festgelegte Bewegung umwirft. Danach wird die manuelle Kontrolle abgeschaltet und der Karton fällt zwischen den anderen Kartons zu Boden. Der Roboter läuft nun durch die Kartons und wirft sie bei Kollision um oder drückt sie zur Seite. Das Fahrzeug fährt nun entlang der grün markierten Strecke und trifft dabei auf den Roboter. Dieser hält das Fahrzeug fest und schleift es mit, während das Fahrzeug weiterhin versucht, der Strecke zu folgen. Dabei erkennt es auch, wenn es rückwärts geschoben wird, obwohl es vorwärts fahren will, und lenkt entsprechend in die seiner Zielrichtung entgegengesetzte Richtung. Schließlich kommt es frei, folgt wieder der Strecke und trifft auf einen Stapel Tonnen, den es umwirft.

Hier konnten also zwei unabhängig voneinander entwickelte Simulationen zu unterschiedlichen Themen kombiniert werden, ohne dass auch nur ein OT oder eine VH angepasst werden musste. Durch entsprechende Anpassungen der VHs könnten das Fahrzeug und der Roboter auch interagieren, indem sie sich z.B. gegenseitig ausweichen. Eine große Schwäche vergleichbarer Lösungen ist die schlechte Verbindbarkeit unterschiedlicher Simulationen (siehe Kap. 2.1 ab S. 29).

10.4.2 Animationsfilme zum Thema „Didaktik der Physik“

Computeranimationen eignen sich auch zur Verdeutlichung von Lerninhalten. Um die Flexibilität von MaxControl in Bezug auf die Erweiterung der Anwendungsgebiete zu überprüfen, wurden verschiedene Filme produziert, die sich für die Didaktik verschiedener Inhalte der Physik eignen. Entwickelt wurden diese Filme in Zusammenarbeit mit Herrn Dr. Härtel vom „Institut für Theoretische Physik und Astrophysik der Universität Kiel“, der bis 2001 am „IPN - Leibniz-Institut für die Pädagogik der Naturwissenschaften an der Universität Kiel“ tätig war.

10.4.2.1 Automatische Animation datenrepräsentierender Objekte



Abbildungen 98: Bilder zu „Automatische Animation datenrepräsentierender Objekte“



Movies\VectorTest_3_V2_noMotionBlur.avi

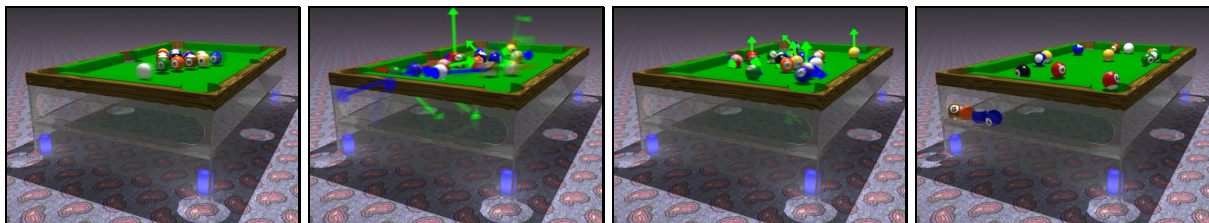


Movies\VectorTest_3_V3.avi


Bei Animationen aus diesem Fachgebiet „Didaktik der Physik“ ist es oft hilfreich, sie mit Darstellungen zugehöriger Daten anzureichern. Bei der Darstellung des Verhaltens von mechanisch simulierten Körpern kann es z.B. sehr nützlich sein, Vektoren zu zeigen, welche die wirkenden Kräfte und Geschwindigkeiten repräsentieren. Schon eine physikalisch korrekte Animation von starren Körpern in nicht-trivialen Bewegungen ist mit manuellen Techniken sehr schwierig, hier kann die Simulation von Mechanik in MaxControl eingesetzt werden. Jedoch ist auch die Animation passender Vektoren, die zu jedem Zeitpunkt der Animation korrekte Größen darstellen, manuell kaum realisierbar. Die offene Architektur von MaxControl ermöglichte nun die Implementation spezieller Klassen, die das Verhalten solcher Vektoren korrekt spezifizieren. Die hier implementierten Verhaltensweisen für Vektoren kommunizieren mit den jeweils zugehörigen mechanisch simulierten Körpern in der Simulation und lesen Werte wie die aktuelle Geschwindigkeit und Bewegungsrichtung oder die aktuell auf den Körper wirkende Gesamtkraft aus diesen Objekten aus und passen ihre Position, Länge und Ausrichtung im Raum entsprechend an. Solche Vektoren können nun mit jedem beliebigen mechanisch simulierten Körper in der Szene verknüpft werden, so dass korrekte Werte für diese Objekte dargestellt werden können. Die Verhaltensweisen der Vektoren wurden so allgemein gehalten, dass sie beliebige Eigenschaften des passenden Datentyps aus jedem Simulationsobjekt verwenden können, sie sind also nicht auf mechanisch simulierte Körper festgelegt und könnten auch die Helligkeit einer Lichtquelle oder die Tonhöhe einer Tonquelle darstellen.


Die vorliegende Animation zeigt mehrere mechanisch simulierte Körper, die einen Abhang hinunterrollen und dabei mehrfach vom Boden abprallen. Die Simulation von Mechanik wurde hier mit der entsprechenden in MaxControl integrierten Shockwave-Komponente durchgeführt. Die dreidimensionalen Vektoren zeigen nun verschiedene Werte an, die der Physiksimation bekannt sind und an MaxControl zurückgegeben werden. Die Roten Vektoren geben die wirkende Gesamtkraft an, die blauen Vektoren repräsentieren die aktuelle lineare Geschwindigkeit, während die grünen Vektoren die aktuelle Rotationsachse und über ihre Länge die Winkelgeschwindigkeit zeigen. Diese Farbverteilung gilt auch für die folgenden Beispiele, sofern sie solche Vektoren zeigen. Die zweite hier angegebene Filmdatei enthält ein Rendering der Animation, in dem zusätzlich Bewegungsunschärfe (Motion-Blur, siehe Kap. 10.2.3.2) berechnet wurde. Dadurch wirkt die Animation weicher, nur für kurze Zeit stabile Zustände der Vektoren, die in der ersten Animation wie ein Aufblitzen dieser Vektoren wirken, werden dadurch allerdings weitgehend verschluckt.

10.4.2.2 Simulation von Billardkugeln



Abbildungen 99: Bilder zu „Simulation von Billardkugeln“

 Movies\Billard_3.avi

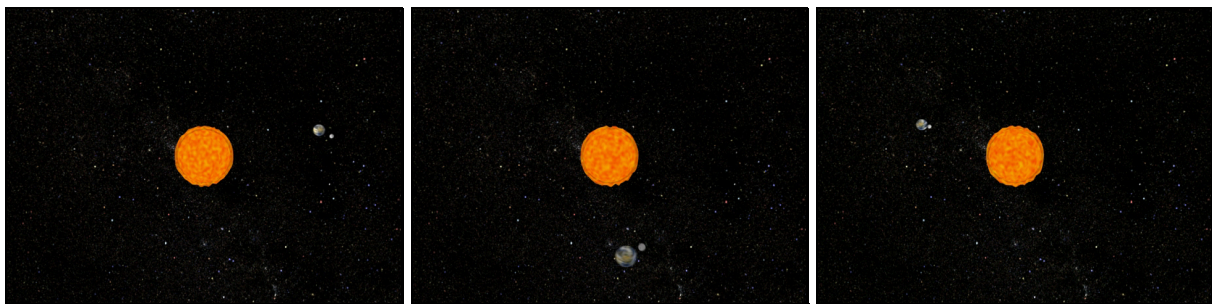
 Movies\Billard_5.avi

Diese Animation zeigt die Bewegungen von Billardkugeln auf einem Billardtisch. Zur Simulation der Bewegungen der Kugeln wurde wie im vorangehenden Beispiel die


Simulation von Mechanik in MaxControl verwendet, ebenso finden die oben erläuterten Vektoren wieder Verwendung, die nun entsprechende Daten über die Kugeln zeigen. Dass sich Kugeln, die in ein Loch gefallen sind, automatisch in einer Ecke der unteren Ebene des Billardtisches anordnen ist allein durch die geometrische Konstruktion des Tisches bedingt, die ebenfalls an die Mechaniksimulation übergeben wird. Dies zeigt das Potential der Spezifikation von Bewegungsabläufen durch die Simulation von Mechanik.

Da Shockwave keinen Rollwiderstand für mechanisch simulierte Körper unterstützt, wurde dieser Widerstand als Unter-VH `MCB_PSimRollingDragForce` einer global wirkenden Verhaltensweise `MCBU_Physics` in MaxControl implementiert, deren weitere Unter-VH `MCB_MassGravity` beispielsweise auch die Anziehungskräfte zwischen Körpern großer Masse wie Planeten simulieren kann. Nur im zweiten hier angegebenen Film wird der Rollwiderstand berücksichtigt. Im ersten Film verlieren die Kugeln nur dann an Geschwindigkeit, wenn sie gegen einen anderen Körper stoßen, erst im zweiten Film verlieren sie beim Rollen über die Oberfläche des Tisches ständig an Geschwindigkeit. Dadurch bedingt werden die Kugeln aber auch insgesamt schneller langsamer und infolgedessen sind die Vektoren nur relativ kurz zu sehen, weil sie schnell zu klein werden und dann von den Kugeln verdeckt werden.

10.4.2.3 Simulation von Sonne, Erde und Mond



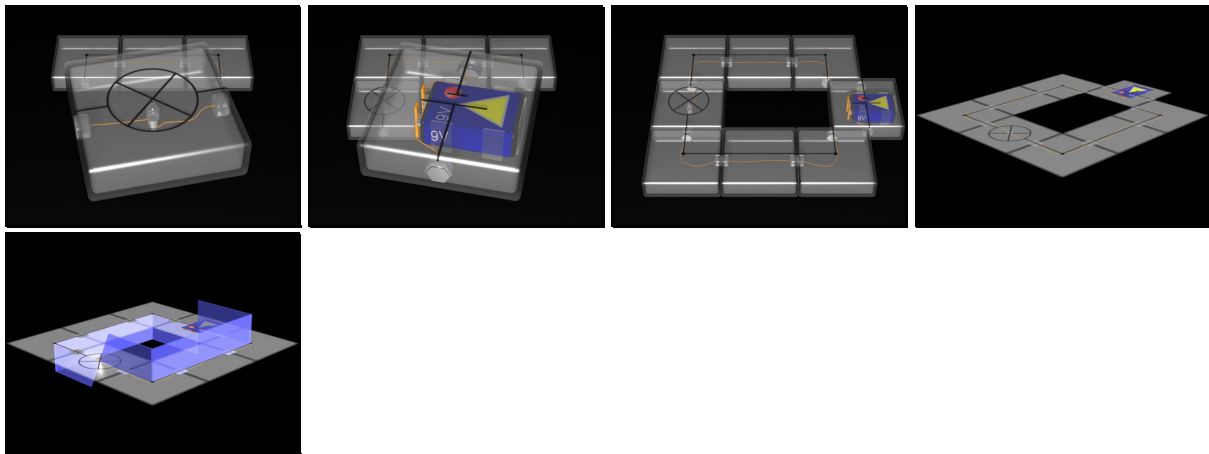
Abbildungen 100: Bilder zu „Simulation von Sonne, Erde und Mond“

 `Movies\SunEarthMoon_2.avi`


Die im vorangehenden Beispiel genannte Verhaltensweise `MCB_MassGravity` wurde hier verwendet, um die Bahnen von Erde und Mond um die Sonne zu zeigen. Neben Anfangsgeschwindigkeiten und Massen für Erde und Mond wurde diese Animation allein durch die Simulation von Massengravitation erstellt. Dabei wird ersichtlich, dass die Erde keiner perfekt elliptischen Bahn folgt, sondern auf ihrer Bahn vom Mond zu leichten Abweichungen gezogen wird.

Auch hier kamen wie in Kap. 10.4.1.5 Zeitlupe und Zeitraffer zum Einsatz. Da die Massengravitation durch eine VH gesteuert wird, die wie jede VH nur einmal in jedem Simulationszwischen schritt handelt, werden die von ihr bestimmten Anziehungskräfte zwischen den Massen während eines Mechaniksimulationsschrittes als konstant angenommen, obwohl sie sich bei jeder Positionsveränderung der Massen ändert. Um näherungsweise korrekte und damit stabile Bahnen zu erhalten, mussten deshalb relativ viele Simulationszwischen schritte für jeden Frame der Animation berechnet werden. Hier war es sehr wichtig, dass diese Anzahl an Simulationszwischen schritten bei Zeitlupe und Zeitraffer durch MaxControl automatisch wie in Kap. 10.2.2 erläutert angepasst wurde.

10.4.2.4 Darstellung von Potentialverläufen in Schaltungen



Abbildungen 101: Bilder zu „Darstellung von Potentialverläufen in Schaltungen“

 Movies\TICS_Scene1.avi

Ziel dieses Projektes war die Darstellung der zeitlichen Entwicklung von Potentialverläufen in verschiedenen aus Bausteinen aufgebauten elektronischen Schaltungen.

Zu Beginn der Animation sollte sich die Schaltung selbst aus den einzelnen Bausteinen zusammensetzen. Dabei sollten Bausteine, deren Typ noch nicht genauer gezeigt wurde, dicht vor der Kamera um 180° rotieren, bevor sie in die Schaltung eingesetzt werden. Für verschieden große Schaltungen war es erforderlich, den Kamerablickwinkel stets so einzustellen, dass die gesamte entstehende Schaltung überblickt werden kann. Die Schaltung sollte dann auf die zweidimensionale Ebene des Untergrundes zusammengedrückt werden und die Kamera sollte so schwenken, dass die Schaltung schräg aus einem 45° -Winkel gezeigt wird. Dann sollte die Batterie in die Schaltung eingeschoben werden und es sollte sich darauf folgend der Potentialverlauf entlang der Schaltung entwickeln. Danach war eine weitere Rotation der Kamera um 360° um die Szene gefordert.

Um dies bei beliebigen Schaltungen effizient auf diese Art animieren zu können, wurde MaxControl eingesetzt. Das wichtigste Problem bestand hier darin, die Schaltung als Ganzes zu überblicken, die Bausteine in eine Animationsreihenfolge für den Aufbau der Schaltung zu ordnen, darauf zu achten, dass kein Bausteintyp mehrfach dicht vor der Kamera gezeigt wird, und die Kamera passend einzustellen. Diese Aufgaben konnten besser von einer global auf die Szene wirkenden VH gelöst werden als von VHs, die direkt den einzelnen Bausteinen zugeordnet sind und diese animieren. Obwohl dies für MaxControl ungewöhnlich ist, steuern sich die Bausteine hier also nicht selbst, sondern sie werden von der VH `MCBU_TICS_SceneController` des OTs `MCU_TICS_SceneController` zentral animiert. Auch ein solcher Lösungsansatz ist durch SOs des Typs `MCO_Universe` also möglich (siehe Kap. 7.3).

Ein SO des Typs `MCU_TICS_SceneController` übernimmt somit fast die gesamte Animation der Szene. Nur das Einschalten der Glühbirne und die Animation des Potentialverlaufs wurden manuell animiert, ohne MaxControl zu verwenden. Hier zeigt sich der Vorteil der engen Integration von MaxControl in 3D-Studio-MAX, der dazu führt, dass eine mit MaxControl erstellte Animation problemlos durch weitere Szenenobjekte und Animationen ergänzt werden kann, bevor die Animation durch Rendering in einen Film umgewandelt wird.

10.5 Geplante Weiterentwicklungen von MaxControl

MaxControl ist durch seine modulare und offene Architektur gut erweiterbar. Im Folgenden werden einige geplante Weiterentwicklungen von MaxControl erläutert, die bereits im aktuellen Konzept von MaxControl bedacht sind und daher keine grundsätzlichen Änderungen an der Programmarchitektur erfordern.

Eine wichtige Neuerung wäre die Möglichkeit, direkt auf die Geometriedaten eines SOs zuzugreifen, so dass dessen genaue Form durch die VHs berücksichtigt, aber auch verändert werden kann. Dies wäre über eine entsprechende neue SP-Klasse möglich, z.B. „MCP_Geometry“, die durch die offene Architektur von MaxControl auch ein Benutzer definieren könnte, ohne das Hauptprogramm zu verändern. Die Formveränderung eines 3D-Objektes im Verlauf einer Animation wäre z.B. für die Animation einer Wasseroberfläche erforderlich. Bisher kann MaxControl dazu „Modifikatoren“ aus 3D-Studio-MAX nutzen, indem nur deren Parameter über SPs animiert werden. Zwei solcher Modifikatoren wurden in der Animation aus Kap. 10.4.2.3 verwendet, um die Oberfläche der Sonne mit zwei Welleneffekten zu animieren. Da MaxControl diese Modifikatoren steuert, folgt die Wellenbewegung auch der verwendeten Zeitlupe und dem Zeitraffer.

Viele vergleichbare Automatisierungswerkzeuge unterstützen eine bequeme Animation von Figuren mit Gliedmaßen, wie Menschen oder Tieren, indem sie aus vorgefertigten Bewegungsabläufen und Posen neue Bewegungsabläufe zusammensetzen. Solche Werkzeuge verwenden oft in externen Dateien gespeicherte Bewegungsabläufe, um daraus neue komplexe Bewegungsabläufe abzuleiten und zusammensetzen. Da bei MaxControl die in Java implementierten Verhaltensweisen auch auf Dateien zugreifen können, wäre ein solches Vorgehen auch hier möglich, es müsste jedoch vom Anwender in Form entsprechender Verhaltensweisen implementiert werden. Es ist geplant, MaxControl so zu erweitern, dass diese vordefinierten Bewegungsabläufe in „Vorbildobjekten“ in der Szene als Animationen gespeichert werden können, um diese dann bei der Animation anderer 3D-Objekte zu verwenden. Dazu müssten spezielle neue Arten von SPs in der Lage sein, zu Beginn der Animation alle Bewegungsabläufe der „Vorbildobjekte“ zu lesen und zu speichern, damit sie den zu animierenden Objekten als Vorgaben zur Verfügung stehen.

Ferner unterstützt MaxControl bisher noch keine dynamische Erzeugung neuer 3D-Objekte in der Szene während der Simulation. Dies ist eine Fähigkeit von Partikelsystemen, die dynamisch 3D-Objekte erzeugen und wieder aus der Szene entfernen können, um so z.B. einen Wasserstrahl aus Wassertöpfen darzustellen, die am Ende eines Wasserschlauches entstehen und beim Kontakt mit dem Boden wieder verschwinden. Es wurden bereits Konzepte entwickelt, mit denen MaxControl diese Fähigkeiten erlangen kann. Das bisherige Simulationskonzept und die Datenstrukturen sind bereits dafür vorbereitet. Da Partikelsysteme in dem von MaxControl verwendeten 3D-Animationsprogramm 3D-Studio-MAX im Allgemeinen eigenständige 3D-Objekte sind, die ihre dynamisch erzeugten 3D-Objekte, die so genannten Partikel, selbst verwalten, kann MaxControl als vorläufige Lösung solche Partikelsysteme wie alle anderen 3D-Objekte steuern. MaxControl kann z.B. die Position eines Partikelsystems verändern, was den Entstehungsort neuer Partikel mit verschiebt, wenn das Partikelsystem im 3D-Raum gleichzeitig den so genannten Partikelemitter darstellt, der die neuen Partikel in seiner aktuellen Position oder einer daraus abgeleiteten Umgebung entstehen lässt. Die Ausrichtung des Partikelemitters, die MaxControl ebenfalls steuern kann, bestimmt zudem meistens bis auf gewünschte einstellbare Abweichungen die Bewegungsrichtung der neu entstehenden Partikel. Auch Parameter, die das Verhalten des Partikelsystems kontrollieren, wie z.B. die Anzahl der pro Einzelbild neu erzeugten Partikel oder deren Geschwindigkeit, können von MaxControl gesteuert werden. So

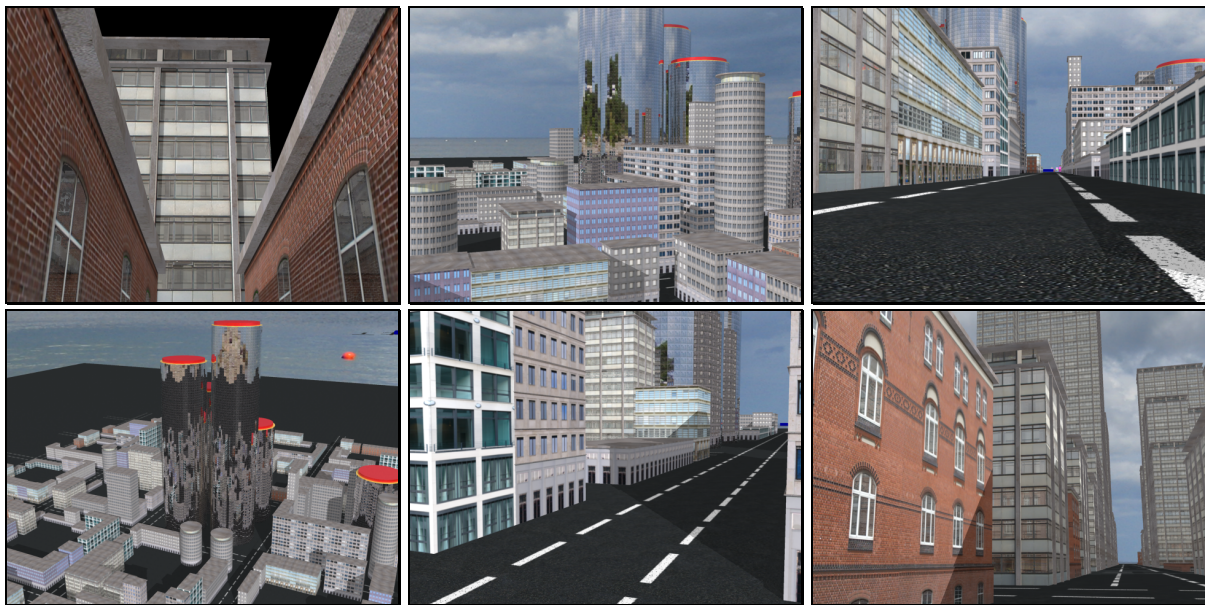
kann MaxControl Partikelsysteme steuern und diese die entsprechenden Aufgaben des dynamischen Hinzufügens und Entfernens von 3D-Objekten übernehmen lassen. Im Allgemeinen kann jedoch nicht die volle Kontrolle über jedes einzelne Partikel erlangt werden und die Partikel sind in ihrer geometrischen und hierarchischen Komplexität bei den meisten Partikelsystemen eingeschränkt. Deshalb bleibt weiterhin eine Erweiterung wünschenswert, mit der MaxControl selbst neue 3D-Objekte mit beliebiger geometrischer und hierarchischer Komplexität dynamisch erzeugen und steuern kann. Dabei stellt sich das Problem, dass die Existenz eines 3D-Objektes in der Szenenzustandsfolge nicht animiert werden kann. Ein 3D-Objekt ist entweder in einer Szene enthalten oder nicht. Man kann jedoch die Sichtbarkeit von 3D-Objekten steuern, so dass ein Objekt erst bei seiner „Entstehung“ sichtbar wird und bei seinem „Verschwinden“ unsichtbar wird. Danach bliebe es jedoch Teil der Szene. Bei den meisten Anwendungen eines Partikelsystems entstehen jedoch sehr schnell viele neue Objekte, so dass ein triviales Erstellen dieser neuen Objekte, die erst bei ihrer Entstehung sichtbar werden, die Szene mit neu erstellten Objekten überfüllen würde, da sie bei ihrem „Verschwinden“ nur unsichtbar werden. Daher ist ein „Objekt-Reinkarnations-System“ geplant, das „verschundene“ unsichtbare Objekte wieder verwendet, wenn ein gleiches Objekt neu entstehen soll. Damit würde beim Beispiel des Wasserstrahls von oben relativ schnell eine maximale Anzahl gleichzeitig bewegter Objekte entstehen, die dann konstant bleiben würde, sobald für jedes neu „entstehende“ Objekt ein „verschundenes“ Objekt zur Wiederverwendung zur Verfügung steht.

In Kapitel 9.2 wurden bereits die beiden vorbereiteten Klassen `MCO_Rendering` und `MCO_Generating` genannt, deren geplante Fähigkeiten werden im Folgenden näher erläutert:

`MCO_Rendering` soll das Rendering einer Szene steuern können, um so beispielsweise auf der Basis einer animierten Szene mehrere Filmdateien zu erstellen. Dies könnte sinnvoll werden, wenn 3D-Objekte in einer Szene „A“, die durch MaxControl animiert wurde, Bildschirme darstellen sollen, deren Inhalt ebenfalls von der Simulation gesteuert wird. Werden beispielsweise mehrere Flugzeuge von MaxControl animiert, wäre es wünschenswert, wenn jeder Radarschirm jedes Flugzeugs die jeweils anderen Flugzeuge anzeigen könnte. Dazu könnten die VHs der Flugzeuge entsprechende Radardaten zunächst in Dateien speichern. Um nun die Radarschirme über animierte Texturen zu realisieren, könnten diese Texturen auf der Basis einer speziellen Szene „S“ erzeugt werden. In dieser Szene könnten VHs diese Radardaten lesen und so nacheinander die Anzeigen der Radarschirme der Flugzeuge animieren, so dass die Szene „S“ beispielsweise von Bild 1-100 den Radarschirm von Flugzeug 1 zeigt, von Bild 101-200 den Radarschirm von Flugzeug 2 und so weiter. Durch Rendering dieser Bildsequenzen in eigene Filmdateien, die jeweils als Texturen den Radarschirmen der Flugzeuge aus der ursprünglichen Szene „A“ zugeordnet sein müssen, könnten entsprechende Filmsequenzen entstehen, systematisch gesteuert durch ein SO des Typs `MCO_Rendering` in der Szene „S“.

MaxControl kann komplexe Szenen mit vielen 3D-Objekten animieren, doch auch die Erstellung einer solchen Szene ist nicht trivial. SOs des Typs `MCO_Generating` sollen genau diese Aufgabe übernehmen, indem sie in einem rekursiven Ansatz komplexe Szenen automatisch erzeugen. Als Basis sollen neben algorithmisch erzeugten Objekten dabei auch „Vorbildobjekte“ zum Einsatz kommen, die mehrfach wieder verwendet werden können. Um eine Stadt zu erzeugen, könnte zunächst ein Quader „A“ erstellt werden, dessen Breite und Länge die Ausmaße der Stadt angeben und dessen Höhe die maximale Gebäudehöhe angibt. Seine VHs würden bei ihrem Aufruf nun Häuserblöcke unterschiedlicher Höhe erzeugen, wobei die Höhen zwar zufällig gewählt werden, die mittlere Höhe jedoch von der Stadtmitte

nach außen abnimmt. Der Quader „A“ würde nun inaktiv und unsichtbar werden und in im folgenden Simulationsschritt würden stattdessen die neu erzeugten Häuserblöcke Häuser und Straßen erzeugen, die auf vorbereiteten „Vorbildobjekten“ basieren, und danach selbst unsichtbar und inaktiv werden. Dieser Vorgang würde sich wiederholen, bis kein aktives Objekt mehr in der Szene vorliegt und die gewünschte Szene erstellt wurde. Um dieses Konzept auf seine Praxistauglichkeit hin zu testen, wurden vom Hamburger Architekturbüro „Scoop“⁶⁸ modulare Gebäudeteile entwickelt, die zu verschiedenen Gebäuden zusammengesetzt werden können. In MAXScript wurde dann ein Programm entwickelt, das dem oben erläuterten Prinzip folgt, dem jedoch die klaren Strukturierungsmöglichkeiten von Java fehlen und das auch nicht modular erweiterbar ist, so dass eine Umsetzung solcher Prinzipien in Java im Rahmen von MaxControl ein umzusetzendes Ziel bleibt. Mit dem MAXScript-Programm konnte automatisch eine Stadt erzeugt werden, die sich beispielsweise für die Autorennscenen aus Kapitel 10.4.1.5 eignen würde und auch für diese konzipiert wurde. Es folgen einige Bilder der so erzeugten Stadt:




Abbildungen 102: Bilder einer automatisch generierten Stadt

Es gibt verschiedenste Methoden der automatischen Erstellung von Szenen, die ähnlich den Systemen zur automatischen Animation von Szenen meistens jeweils auf eine bestimmte Problemstellung spezialisiert sind. Analog zur Animation soll in MaxControl ein erweiterbares System integriert werden, das automatisch Szenen mit verschiedensten Zielsetzungen erzeugen kann. Beispiele für Systeme zur automatischen Erstellung von Gebäuden und Städten sind [Parish01, Sun02, Wonk03]. Vergleichbare Systeme zur automatischen Erstellung von Szenen mit anderen oder ähnlichen Zielsetzungen sind [Reeves85, Smith02, Cutler02, ChenY02], wobei [Cutler02] bereits ein sehr flexibles System ist.

⁶⁸ <http://www.scoopstudio.de/>, 04.06.2007; Scoop Hamburg: Schulterblatt 58, 20357 Hamburg

11 Diskussion

Dieses Kapitel beurteilt die Ergebnisse der Entwicklung und Anwendung von MaxControl, auch im Hinblick auf die zugrundeliegenden Thesen aus Kapitel 1.3.

Wie die Beispiele aus Kapitel 10.4 zeigen, konnten mit MaxControl sehr lange wiederholungsfreie Animationen erzeugt werden, deren Realisierung mit Keyframe-Techniken einen unvermeidbar höheren Aufwand bedeutet hätten. Die mit MaxControl produzierten Animationen haben viele verschiedene Themen zum Inhalt, die nur schwer alle mit einem einzelnen anderen gängigen Werkzeug wie Partikelsystemen oder Crowd-Systemen umsetzbar gewesen wären, da diese sich nicht flexibel genug für die verschiedenen Aufgabenstellungen anpassen lassen. MaxControl zeigte dagegen die erforderliche Erweiterbarkeit durch seine modulare offene Architektur. Auch jede einzelne Animation aus Kapitel 10.4 hätte nur schwer mit jeweils einem dieser Werkzeuge umgesetzt werden können, da sie nicht unter die typischen Anwendungsgebiete dieser Werkzeuge fallen. Die Roboter aus den Kapiteln 10.4.1.2-10.4.1.4 wären in der gezeigten Form nicht, wie nahezuliegen scheint, mit einem Crowd-System animierbar gewesen, da Crowd-Systeme in der Regel eine endliche Zahl vorgegebener Bewegungsabläufe als Basis verwenden. Dagegen passt der Roboter, um in eine neue Richtung zu gehen, nur seine Art zu gehen an, indem er beispielsweise seitwärts oder schräg geht, ohne dabei seinen Unterkörper mit den Beinen in die neue Richtung zu drehen. Er dreht nur seinen Kopf in die entsprechende Richtung. Damit ergeben sich durch die möglichen Navigationsrichtungen bei entsprechend feiner Winkeleinteilung unendlich viele Arten des Gehens, die in MaxControl über einen entsprechenden Algorithmus in den Verhaltensweisen des Roboters erzeugt werden. Die Fahrzeuge aus Kapitel 10.4.1.5 hätten nicht allein mit einem Werkzeug zur Simulation von Mechanik animiert werden können, wie sie inzwischen Teil vieler 3D-Animationsprogramme sind, da auch das autonome Verhalten der Fahrzeuge auf der Basis einer Mechaniksimulation umgesetzt werden musste. (Eine Ausnahme ist der frühe Testfilm „: Movies\CarRace\Car_Test_9.avi“, weil hier das autonome Verhalten für die Fahrzeuge noch nicht implementiert war.) Durch die Mechaniksimulation erhalten die Fahrzeuge zudem wieder so viele Bewegungsmöglichkeiten mit subtilen Feinheiten (Radfederung), dass Crowd-Systeme mit vordefinierten Bewegungsabläufen diese Animationen ebenfalls nicht in der gezeigten Form hätten umsetzen können. Partikelsysteme kämen mit ihren Fähigkeiten, Partikel z.B. Pfaden und Oberflächen folgen zu lassen, für diese Animationen in Frage, jedoch unterstützen Partikelsysteme in der Regel nicht die Simulation von Mechanik für komplexe mechanische Konstrukte wie über federnde Radachsen mit der Karosserie verbundene Räder. Kein bekanntes System hätte für die Animationen aus Kapitel 10.4.1 automatisch so vollständig Toneffekte generieren können.

MaxControl hat also die Herstellung von Animationen ermöglicht, die thematisch zwar praxisnah sind, die aber mit anderen Werkzeugen nicht so effizient, vollständig und ohne aufwändige manuelle Schritte bei der Animation realisierbar gewesen wären.

Es folgt eine Überprüfung der Thesen aus Kapitel 1.3:

Zu These T-1:

MaxControl kann auf zwei Arten genutzt werden. Einerseits können eigene OTs und VHs implementiert werden, um 3D-Objekten das gewünschte Verhalten zu geben. Diese Arbeit sollte dem Programmierer zugeteilt sein. Andererseits kann man sich auch auf die Verwendung der bisher implementierten OTs beschränken. Diese können direkt in der grafischen Benutzeroberfläche von 3D-Studio-MAX den 3D-Objekten zugewiesen werden. Sobald MaxControl dann die entsprechenden nicht-standard SPs und die MPs erzeugt hat,

können diese wie alle anderen Eigenschaften der 3D-Objekte durch einen Benutzer mit Werten belegt und ggf. animiert werden. Statt nun aber z.B. wie bisher nötig bei den Blinklichtern der Landebahn aus Kap. 10.4.1.1 die Helligkeit jeder einzelnen Lichtquelle zu animieren, wird nur der über SPs der Landebahn animiert, wann diese sich einschalten und damit ihre Lichtquellen blinken lassen soll oder wann sie sich wieder ausschalten soll. Die eigentliche Animation der Lichtquellenhelligkeiten übernimmt dann MaxControl. Diese Einstellungen sind alle über die Benutzeroberflächen von 3D-Studio-MAX und MaxControl möglich und damit auch für Künstler ohne Programmierkenntnisse geeignet. Damit ist die geforderte Arbeitsteilung möglich, bei der Künstler wie gewohnt 3D-Objekte entwerfen, erstellen und, diesmal auf höherem Kontrollniveau, ihre Eigenschaften animieren, während Programmierer das Verhalten der Objekte spezifizieren. Dabei ist eine erfolgreiche Kommunikation zwischen Künstler und Programmierer erforderlich, was jedoch eine erprobte Methodik aus der Entwicklung von Computer- und Videospiele ist. Das Fahrzeug aus Kapitel 10.4.1.5 wurde von einem Künstler erstellt und gekauft (Quelle siehe dort), der keinerlei Kenntnis von der geplanten Animation oder der dabei eingesetzten Technik hatte. Anpassungen dieses Fahrzeugs in 3D-Studio-MAX waren vor allem bezüglich der Lichtquellen und der Einstellung der SPs erforderlich, die optischen Eigenschaften konnten jedoch weitgehend unverändert übernommen werden. Ein weiteres Beispiel für eine direkte Zusammenarbeit mit Künstlern sind die modularen Gebäudekomponenten aus Kapitel 10.5, die Architekten nach genauen Vorgaben erstellten (Quelle siehe dort), die für den Algorithmus zur automatischen Erstellung einer Stadt eingehalten werden mussten, wobei die Zielsetzung hier jedoch nicht die automatische Animation sondern die automatische Erstellung einer Szene war.

Zu These T-2:

Die Simulation von Mechanik hat für MaxControl den Realismus und die Glaubwürdigkeit der Animationen stark erhöht. Während die nicht auf Mechanik basierende Animation der Roboter aus den Kapiteln 10.4.1.2-10.4.1.4 in ihren Bewegungen noch künstlich wirken, weil die Bewegungen zu störungsfrei sind, da die Roboter beispielsweise durch kein anderes Objekt festgehalten, bewegt oder umgeworfen werden können. Dagegen wirken auf Mechanik basierenden Bewegungen der Fahrzeuge aus Kapitel 10.4.1.5 weitaus realistischer, da diese nicht nur andere Gegenstände umwerfen können, sondern auch von ihnen aus der Bahn geworfen werden können. Sie können über Sprungschancen fahren und sich überschlagen, die Radfederung bringt subtile Feinheiten in die Animation. Da die Fahrzeuge in der so nach den Gesetzen der Mechanik simulierten Welt handeln müssen, handeln sie auch realistischer, da sie nicht einfach ihre Position ändern können, sondern wie echte Fahrzeuge nur Kräfte erzeugen können, um ihre Position und Rotation im Raum ihren Zielvorgaben entsprechend zu ändern. Die Objektorientierung erlaubte die effiziente Erstellung, Wiederverwendung und Erweiterung von Verhaltensweisen, welche die erforderlichen Kräfte bestimmen und auch weitere Animationsaufgaben übernehmen können wie z.B. die automatische situationsabhängige Steuerung der Lichtanlage der Fahrzeuge.

Zu These T-3:

Die in Kapitel 10.4 gezeigten Animationen wurden fast ausschließlich vollständig automatisiert durch MaxControl erzeugt. Das Autorennen aus Kapitel 10.4.1.5 hätte theoretisch ohne manuelles Eingreifen beliebig lange weitersimuliert werden können, ohne dass sich eine Animationsschleife ergibt, indem sich ein Szenenzustand wiederholt. Die Vorgaben, denen die Animation dabei folgt, finden sich in den Verhaltensweisen, der Einstellung ihrer SPs im Anfangszustand, den Festkörpern in der Szene (Sprungschanze, Tonnen, Boden, Fahrzeuge etc.) sowie dem durch entsprechende SOs festgelegten Streckenverlauf. Ein manuelles Eingreifen zeigt die Animation aus Kapitel 10.4.1.6, in der ein manuell animierter Karton zu Beginn der Simulation andere Kartons umwirft. Durch die MPs

ist eine manuelle Kontrolle aller animierbaren SPs möglich, die manuelle Kontrolle nicht-animierbarer SPs besteht in der Wahl eines Anfangswertes, der auch für alle animierbaren SPs manuell festgelegt wird, damit ein Startzustand für die Simulation entsteht. Da sowohl die Standard-SPs wie die Position eines SOs, als auch die SPs, die sich auf das Verhalten eines SOs beziehen, manuell kontrolliert werden können, kann das Verhalten eines SOs auf verschiedenen Ebenen kontrolliert werden. Entweder sehr nah an herkömmlicher manueller Animation durch direkte Manipulation der Zielparameter wie Position oder Rotation oder auf höherer Ebene durch Kontrolle der verhaltenssteuernden SPs, wie beispielsweise von Navigationsrichtungen, Maximalgeschwindigkeiten oder Grenzwerten, nach denen in VHs Entscheidungen getroffen werden.

Zu These T-4:

Die SOs werden zunächst über ihre VHs selbst gesteuert, aber auch durch die Werte ihrer SPs. Ebenso kann z.B. die Position anderer SOs ihr Verhalten steuern. So wurde in den Animationen aus Kap. 10.4 das Verhalten der 3D-Objekte so festgelegt, dass zwar klar definiert wurde, welche Ziele die SOs erreichen sollen und welche Vorgaben sie dabei einhalten sollen, jedoch nicht, welche kleineren Handlungen sie dabei exakt ausführen sollen. So gehen die Roboter aus Kap. 10.4.1.2-10.4.1.4 auf Zielpunkte zu und weichen sich dabei gegenseitig aus. Die Fahrzeuge folgen der vorgegebenen Strecke und versuchen dabei, sich gegenseitig zu überholen. Solche Anweisungen würde man auch an Statisten in einer Massenszene bzw. Stunt-Fahrern, die ein Autorennen darstellen sollen, geben. Sowohl bei den Simulationen als auch bei Schauspielern verlässt man sich bei den feineren Einzelhandlungen dabei auf das individuelle Verhalten der Objekte bzw. Schauspieler. In der Regel gibt auch ein Regisseur nicht jedem Statisten in einer Massenszene vor, zu welchem Zeitpunkt sie einen einzelnen Schritt mit den Füßen ausführen sollen oder wann ein Stunt-Fahrer die Kupplung betätigen soll. In beiden Fällen hat man also nur die Kontrolle über das grundsätzliche Verhalten der Objekte bzw. Schauspieler, ohne dabei subtile Feinheiten jeder Bewegung vorgeben zu können. Gerade hier liegt aber in beiden Fällen natürlich auch eine erhebliche Einsparung an Arbeit, da diese genauen Vorgaben auch nicht nötig sind. In beiden Fällen können solche genauen Vorgaben jedoch gemacht werden, wenn sie erforderlich sind. Einem Stunt-Fahrer kann auch gesagt werden, dass er auf ein Signal hin aus der Fahrbahn ausbrechen soll oder sichtbar das Steuer abrupt in einer andere Richtung drehen soll. Durch die MPs ist eine solche genauere Kontrolle zu wählbaren Zeitpunkten auch bei SOs möglich, indem man beispielsweise in einem Intervall die SP eines Fahrzeugs, die den Steuerwinkel der Vorderräder festlegt, entsprechend manuell kontrolliert.

Zu These T-5:

Wie Kapitel 10.2.4 erläutert wurde, können SOs auch tonerzeugende Objekte sein, die wie jedes andere SO durch Verhaltensweisen kontrolliert werden können. So wurde eine automatische Vertonung der meisten Animationen aus Kapitel 10.4.1 möglich, die als manueller Nachbearbeitungsschritt einen erheblichen zusätzlichen Aufwand erfordert hätte. Da die Simulation Daten über alle Geschehnisse im Simulationsverlauf hat, auch über 3D-Objekte außerhalb des Kamerasichtbereichs und über Kollisionsereignisse der Mechaniksimulation, kann sie viel direkter und präziser Toneffekte erzeugen, als dies in einer manuellen Nachbearbeitung möglich wäre. Da auch die Objektpositionen der Simulation bekannt sind, können Surround-Tonspuren entsprechend exakt erzeugt werden.

Zu These T-6:

Wie bereits zu Beginn dieses Kapitels erörtert wurde, konnten die mit MaxControl erstellen Animationen aus Kapitel 10.4 nicht nur aus verschiedensten Themengebieten gewählt werden, sondern sie wären in den gezeigten Formen auch nicht mit der gleichen Effizienz über andere bekannte Werkzeuge realisierbar gewesen. Die Möglichkeiten von MaxControl

sind durch diese Beispiele jedoch bei weitem nicht erschöpft, da es durch seine Architektur sehr leicht erweitert werden kann und sich somit flexibel verschiedensten Anforderungen anpassen kann. Dabei bleiben alle Lösungen Teil desselben Simulationssystems, wodurch sie miteinander interagieren können, was bei vielen Speziallösungen nicht möglich ist, die in der Regel nur nacheinander aber nicht miteinander interagierend angewendet werden können (siehe dazu Kap. 2.1 ab S. 29).

12 Abbildungsverzeichnis

| | |
|---|-----|
| Abbildung 1: Stop-Motion-Film | 8 |
| Abbildung 2: Schatten auf Schauspieler | 15 |
| Abbildung 3: Silhouette in Lichtkegel | 15 |
| Abbildung 4: Pilot in Cockpit 1 | 15 |
| Abbildung 5: Pilot in Cockpit 2 | 15 |
| Abbildung 6: Schauspieler in 3D-Welt 1 | 15 |
| Abbildung 7: Schauspieler in 3D-Welt 2 | 15 |
| Abbildung 8: Schauspieler in 3D-Welt 3 | 15 |
| Abbildung 9: Spiegelung | 15 |
| Abbildungen 10: Integration von Computeranimationen in Live-Action-Filme | 16 |
| Abbildung 11: Architektur von MaxControl | 34 |
| Abbildung 12: Darstellung einer Pyramide durch Dreiecksflächen | 38 |
| Abbildung 13: Pivot-Punkt der Pyramide aus Kapitel 3.1 | 39 |
| Abbildung 14: Hand aus hierarchisch verknüpften Einzelobjekten | 40 |
| Abbildung 15: Baumdarstellung von hierarchisch verknüpften Einzelobjekten | 40 |
| Abbildung 16: Szenenstruktur in 3D-Studio-MAX | 43 |
| Abbildung 17: Hierarchie von 3D-Objekten in 3D-Studio-MAX | 43 |
| Abbildungen 18: Drei Phasen einer Beispielanimation | 45 |
| Abbildung 19: Darstellung von Keys und Werteverläufen in 3D-Studio-MAX | 45 |
| Abbildung 20: Darstellung von Animations-Tracks in 3D-Studio-MAX | 46 |
| Abbildung 21: Interpolatoren (Controller) in 3D-Studio-MAX | 46 |
| Abbildung 22: Mögliche Fortsetzungsarten für Animations-Tracks in 3D-Studio-MAX | 47 |
| Abbildung 23: Diagramm zur Anfangssituation vor der Simulation | 84 |
| Abbildung 24: Diagramm zu den Zustandsübergängen während der Simulation | 86 |
| Abbildung 25: GUI-Repräsentation einiger SPs | 89 |
| Abbildung 26: Simulationsintervall zu Frame n | 89 |
| Abbildung 27: Zeitlicher Wirkungsbereich der manuellen Kontrolle | 90 |
| Abbildung 28: Zustandsübergänge unter Berücksichtigung der MPs | 93 |
| Abbildung 29: Klassenzusammenhänge zu „MCD_LightBlinking“ | 96 |
| Abbildung 30: OT-Hierarchie zu „OT _{Robot} “ | 102 |
| Abbildung 31: OT-Hierarchie zu „OT _{RobotCommunicating} “ | 103 |
| Abbildung 32: OT-Hierarchie zu „OT _{RobotCommunicating} ² “ | 105 |
| Abbildung 33: OT-Hierarchie zu „OT _{MCD_Vector} “ | 109 |
| Abbildung 34: Überblick über die Hierarchie der wichtigsten Basisklassen in MaxControl | 113 |
| Abbildung 35: Teil-OT-Hierarchie zu MCD_Light | 118 |
| Abbildung 36: Ausschnitt einer OT-Hierarchie zu MCB_LightSwitch | 119 |
| Abbildung 37: Beispiel einer Baumsuche nach einer VH | 128 |
| Abbildung 38: Beispiel einer Baumsuche nach einer SP | 131 |
| Abbildung 39: Beispiel einer SO-Hierarchie zur Illustration von Zugriffspfaden | 135 |
| Abbildung 40: Beispiel zur Nutzung semantischer Zusammenhänge für Zugriffspfade | 135 |
| Abbildung 41: Unempfindlichkeit relativer Zugriffspfade gegen Änderungen an der SO-Hierarchie | 137 |
| Abbildung 42: OT-Hierarchie zu MCD_Car ohne SPs | 148 |
| Abbildung 43: Beispielanimation zu „MCD_SimpleMotion“ | 154 |
| Abbildung 44: Beispieleinstellungen zu „MCD_SimpleMotion“ | 155 |
| Abbildung 45: Auszug aus der Klasse MCB_2DNavigation | 158 |
| Abbildung 46: Illustration des Navigationswinkels für „MCB_2DNavigation“ | 159 |
| Abbildung 47: Szenen-Analysealgorithmus | 163 |
| Abbildung 48: Herstellung der Objektliste | 164 |
| Abbildung 49: Ausführungsreihenfolge der VHs in der OT-Hierarchie zu „OT _{Robot} “ | 167 |
| Abbildung 50: Ausführungsreihenfolge der startup-Methoden in „OT _{Robot} “ | 168 |
| Abbildung 51: Simulationszyklus ohne genaue Kommunikation | 169 |
| Abbildung 52: Ablaufdiagramm zu StartupObjects | 171 |
| Abbildung 53: Ablaufdiagramm zu updateFrame | 172 |
| Abbildung 54: Quellcode zu MCB_SimpleFollowerMover | 176 |
| Abbildung 55: Synchrone Bewegung ohne Verzögerung | 177 |
| Abbildung 56: Synchrone Bewegung mit Verzögerung | 178 |
| Abbildung 57: Synchrone Bewegung mit verringerter Verzögerung | 179 |
| Abbildung 58: Positionskommunikation zwischen Eltern- und Kindobjekten, Teil 1 | 179 |

| | |
|--|-----|
| Abbildung 59: Positionskommunikation zwischen Eltern- und Kindobjekten, Teil 2 | 180 |
| Abbildung 60: Ablauf der Mechaniksimulation in einem Simulationszwischenstadium | 197 |
| Abbildung 61: Gestapelte Kartons | 200 |
| Abbildung 62: Gestapelte Tonnen | 200 |
| Abbildung 63: Reifen ohne Motion-Blur | 201 |
| Abbildung 64: Reifen mit Motion-Blur | 201 |
| Abbildung 65: Radachsen durch Federn | 203 |
| Abbildung 66: Track: Abzuflachende Kurve mit 3 Keys | 207 |
| Abbildung 67: Track: Abgeflachte Kurve mit 3 Keys | 208 |
| Abbildung 68: Track: Abzuflachende Kurve mit redundanten Keys | 208 |
| Abbildung 69: Track: Abgeflachte Kurve mit redundanten Keys | 208 |
| Abbildung 70: Zustandsübergänge mit Kommunikation | 212 |
| Abbildung 71: Simulationszyklus mit optimierter Kommunikation | 215 |
| Abbildung 72: Beispiel für momentan relevante Keys, <code>lastKey1.frame=n</code> | 217 |
| Abbildung 73: Beispiel für momentan relevante Keys, <code>lastKey1.frame<n</code> | 217 |
| Abbildung 74: Ablaufdiagramm zu <code>askValue</code> | 219 |
| Abbildung 75: Ablaufdiagramm zu <code>putAtFrame</code> | 221 |
| Abbildung 76: Beispielsituation vor dem Schreiben eines redundanten Keys | 222 |
| Abbildung 77: Setzen eines redundanten Keys | 222 |
| Abbildung 78: Beispielsituation für Right Manual Anchoring vor dem Schreiben | 223 |
| Abbildung 79: Beispielsituation für Right Manual Anchoring nach dem Schreiben ohne Anchoring | 223 |
| Abbildung 80: Beispielsituation für Right Manual Anchoring nach dem Schreiben mit Anchoring | 223 |
| Abbildungen 81: Durchführung des Left Anchoring in einem Beispiel | 225 |
| Abbildung 82: Helligkeitsverlauf eines Blinklichtes | 226 |
| Abbildung 83: Zwischenstand einer Simulation mit vorangehender Simulation | 227 |
| Abbildung 84: Beispielsituation für Right Anchoring vor dem Schreiben | 227 |
| Abbildung 85: Beispielsituation nach der Durchführung des Right Anchoring | 228 |
| Abbildung 86: Key im aktuellen Frame bei Right Anchoring, Situation vor dem Schreiben | 228 |
| Abbildung 87: Key im aktuellen Frame bei Right Anchoring, Situation nach der Korrektur | 229 |
| Abbildung 88: Bedingung für Right Anchoring, Situation vor dem Schreiben | 229 |
| Abbildung 89: Bedingung für Right Anchoring, Situation nach unnötigem Anchoring | 230 |
| Abbildungen 90: Bilder zu „Automatisierte Ansteuerung vieler Lichtquellen“ | 232 |
| Abbildungen 91: Bilder zu „Animation sechsbeiniger Roboter“ | 233 |
| Abbildungen 92: Bilder zu „Simulation einer größeren Anzahl von Robotern“ | 234 |
| Abbildungen 93: Bilder zu „Simulation von Mechanik auf der Basis der Animation sechsbeiniger Roboter“ .. | 234 |
| Abbildungen 94: Autorennen: Kameraperspektiven | 235 |
| Abbildungen 95: Autorennen: Lichteffekte | 236 |
| Abbildungen 96: Autorennen: Mechaniksimulation | 236 |
| Abbildungen 97: Bilder zu „Fahrzeug und Roboter in einer Simulation“ | 238 |
| Abbildungen 98: Bilder zu „Automatische Animation datenrepräsentierender Objekte“ | 239 |
| Abbildungen 99: Bilder zu „Simulation von Billardkugeln“ | 240 |
| Abbildungen 100: Bilder zu „Simulation von Sonne, Erde und Mond“ | 241 |
| Abbildungen 101: Bilder zu „Darstellung von Potentialverläufen in Schaltungen“ | 242 |
| Abbildungen 102: Bilder einer automatisch generierten Stadt | 245 |
| Abbildungen 103: Codec-Einstellungen | 275 |

13 Stichwortverzeichnis

| | | | |
|--|--|------------------------------------|---|
| Δt | 114 | Java-Laufzeitumgebung | 74 |
| 3D-Animation | 44 | Java-Objekt | 38 |
| 3D-Hardware | 17 | Java-Skript | 75 |
| 3D-Objekt | 38 | Key | 44 |
| 3D-Studio-MAX | 54, 68 | Keyframe | 45 |
| 3D-Szene | 39 | Keyframe-Animation | 44, 45 |
| Afterburn | 71 | Key-Reduktion | 208 |
| Alpha-Kanal | 66 | Kinder | 41 |
| Ambient Light | 42 | Kindobjekte | 41 |
| Anchoring | 222 | Krieg der Sterne | <i>Siehe</i> Star Wars |
| Anfangszustand | 25, 78 | Layer | 65 |
| Animation | 44 | Left Anchoring | 220 |
| Animationsintervall | 44 | Lightwave | 68 |
| Animations-Key | 44 | Live-Action | 11 |
| Animations-Track | 45 | Macromedia Shockwave | 72 |
| animierbar | 44 | Manual Property | <i>Siehe</i> MP |
| animierbare SP | 80 | manuell kontrolliert | 87 |
| ASAS | 23 | Massive | 58 |
| Besitzer | 100 | MaxControl | 16, 23 |
| Bewegungsunschärfe | 201 | MaxControl-Zustand | 82 |
| Biped | 57 | MAXScript | 55 |
| Cebas | 71 | Maya | 54, 68 |
| Cel-Shading | 10 | MCH_AxisAngleDegrees | 190 |
| Character Studio | 57 | MCH_Vector3D | 189, 190 |
| Cinema-4D | 68 | Microsoft | 68, 74, 75 |
| Compositing | 10, 65 | Microsoft Internet Explorer | 75 |
| Computeranimationsfilm | 44 | Microsoft Windows XP | 68 |
| Constraints | 55 | momentan letzter Key | 217 |
| Container-Feld | 101 | momentan nächster Key | 217 |
| Crowd-Animation | 57 | momentan relevante Keys | 217 |
| Crowd-System | 57 | momentan vorletzter Key | 217 |
| Cut Scene | 64 | Morphing | 13 |
| Datenbank | 175 | Motion Capture | 18 |
| Echtzeit | 17 | Motion Capturing | <i>Siehe</i> Motion Capture |
| Einzelbild | 44 | Motion-Blur | 201 |
| Eltern | 41 | Motion-Rides | 7 |
| Elternobjekte | 41 | MP | 87 |
| endgültige Szenenzustandsfolge | 84 | MSO | 184 |
| Euler-Winkel | 146 | Neuronale Netze | 29 |
| Feld | 74 | nicht-animierbar | 44 |
| Final Render | 71 | nicht-animierbare SP | 80 |
| Foley Studio MAX | 70 | Nicht-SP | 80 |
| Frame | 44 | nicht-standard SP | 80 |
| Framerate | 7, 115, 192 | Nurbs | 38 |
| Fuzzy Methoden | 29 | Oberobjekte | 41 |
| Game-Engine | 63 | Object Linking and Embedding | 69 |
| Gauß-Seidel | 85, 173 | Objekt | 38 |
| Global Illumination | 9 | Objektliste | 162 |
| GUI | 80 | Objekttyp | 76, 82, 95 |
| Hauptverhaltensweise | 100 | OLE | <i>Siehe</i> Object Linking and Embedding |
| Haupt-VH | <i>Siehe</i> Hauptverhaltensweise | OT | <i>Siehe</i> Objekttyp |
| Herr der Ringe | 19, 53 | OT-Hierarchie | 101 |
| Herr der Ringe – Die Zwei Türme | 19 | Partikel | 243 |
| hierarchische endliche Automaten | 57 | Pivot-Punkt | 39 |
| Image-Based-Rendering | 71 | Plugin | 59 |
| Internet Explorer | <i>Siehe</i> Microsoft Internet Explorer | Polygon | 38 |
| Inverse Dynamik | 55 | Ragdoll | 54, 58 |
| Java | 60, 74 | Reactor | 54 |
| Javadoc | 107 | Rendern | 47 |

| | | | |
|---------------------------------------|-----------------------------------|-------------------------------------|-----------------------------------|
| Right Anchoring..... | 226 | Szenenstruktur | 42 |
| Right Manual Anchoring | 222 | Szenenzustand..... | 42, 82 |
| Shockwave | <i>Siehe</i> Macromedia Shockwave | Szenenzustand n..... | 82 |
| Signatur | 106 | Szenenzustandsfolge..... | 77, 81 |
| Simulation | 76 | Teminator 3..... | 56 |
| Simulation Property | <i>Siehe</i> SP | Temporal-Aliasing..... | 201 |
| Simulationsintervall | 78 | Thinking Particles | 54 |
| Simulationsintervall zu Frame n | 88 | Track | <i>Siehe</i> Animations-Track |
| Simulationsobjekt..... | 76, 82, 95, 125 | Transformation..... | 38 |
| Simulationsschritt..... | 78 | Troja..... | 12 |
| Simulationsschritt zu Frame n..... | 78 | Unterobjekte..... | 41 |
| Simulationszwischenritte | 85 | Unter-VH | <i>Siehe</i> Unterverhaltensweise |
| Sitni Sati..... | 71 | Ursprungskoordinatensystem..... | 41 |
| SO | <i>Siehe</i> Simulationsobjekt | Vector | 116 |
| Softimage | 68 | Verhaltensweise | 82, 95 |
| SO-Hierarchie | 101, 125 | VH..... | <i>Siehe</i> Verhaltensweise |
| SP | 78 | Visual Basic Script..... | 75 |
| Standard-SP..... | 79 | Volumetric-Rendering | 72 |
| Star Wars..... | 11 | vorläufige Szenenzustandsfolge..... | 83, 84, 86 |
| Stop-Motion | 8 | Welt-Koordinaten | 41 |
| Sub-Surface-Light-Scattering | 9 | Weltkoordinatensystem..... | 41 |
| Sun | 74 | Windows XP | <i>Siehe</i> Microsoft Windows XP |
| Surround-Sound | 31, 65 | Wurzelknoten..... | 40 |
| Szene..... | 39 | Xenosaga..... | 64 |
| Szenengraph..... | 40 | Zone of the Enders..... | 64 |
| Szenenhierarchie | 40 | Z-Puffer..... | 66 |

14 Literaturverzeichnis

- [Adl98] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. M. Parikh und J. M. Stichnoth, "Fast, effective code generation in a just-in-time Java compiler", in Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (Montreal, Quebec, Kanada, 17.-19. Juni 1998), S. 280-290, Hrsg. A. M. Berman, PLDI '98, ACM Press, New York, NY, 1998
- [Aitken04] M. Aitken, G. Butler, D. Lemmon, E. Saindon, D. Peters und G. Williams, The "Lord of the Rings: the visual effects that brought middle earth to the screen", in ACM SIGGRAPH 2004 Course Notes (Los Angeles, CA, 8.-12. August 2004), SIGGRAPH '04, Artikel Nr. 11, ACM Press, New York, NY, 2004
- [Alexa00] M. Alexa, D. Cohen-Or und D. Levin, "As-rigid-as-possible shape interpolation", in Proceedings of the 27th Annual Conference on Computer Graphics and interactive Techniques, International Conference on Computer Graphics and Interactive Techniques, S. 157-164, ACM Press/Addison-Wesley Publishing Co., New York, NY, 2000
- [Alia01] D. G. Aliaga und I. Carlbom "Plenoptic stitching: a scalable method for reconstructing 3D interactive walkthroughs", in Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques, SIGGRAPH '01, S. 443-450, ACM Press, New York, NY, 2001
- [Alur99] R. Alur, S. Kannan und M. Yannakakis, "Communicating Hierarchical State Machines", in Proceedings of the 26th international Colloquium on Automata, Languages and Programming (11.-15. Juli 1999), Hrsg.: J. Wiedermann, P. v. Boas und M. Nielsen, Lecture Notes In Computer Science, Vol. 1644, S. 169-178, Springer-Verlag, London, 1999
- [Angel05] A. Angelidis und F. Neyret, "Simulation of smoke based on vortex filament primitives", in Proceedings of the 2005 ACM Siggraph/Eurographics Symposium on Computer Animation (Los Angeles, Kalifornien, 29.-31. Juli 2005), SCA '05, S. 87-96, ACM Press, New York, NY, 2005
- [Ausl95] J. Auslander, A. Fukunaga, H. Partovi, J. Christensen, L. Hsu, P. Reiss, A. Shuman, J. Marks und J. T. Ngo, "Further experience with controller-based automatic motion synthesis for articulated figures", ACM Trans. Graph. 14, 4, Okt. 1995, S. 311-336, 1995
- [Bara98] D. Baraff und A. Witkin, "Large steps in cloth simulation", in Proceedings of the 25th Annual Conference on Computer Graphics and interactive Techniques, SIGGRAPH '98, S. 43-54, ACM Press, New York, NY, 1998
- [Bara03] D. Baraff, A. Witkin und M. Kass, "Untangling cloth", ACM Trans. Graph. 22, 3 (Jul. 2003), S. 862-870, 2003
- [Beier92] T. Beier und S. Neely, "Feature-based image metamorphosis", SIGGRAPH Comput. Graph. 26, 2 (Jul. 1992), S. 35-42, 1992

- [Bell05] N. Bell, Y. Yu und P. J. Mucha, "Particle-based simulation of granular materials", in Proceedings of the 2005 ACM Siggraph/Eurographics Symposium on Computer Animation (Los Angeles, Kalifornien, 29.-31. Juli 2005), SCA '05, S. 77-86, ACM Press, New York, NY, 2005
- [Blinn82] J. F. Blinn, "A Generalization of Algebraic Surface Drawing", ACM Trans. Graph. 1, 3 (Jul. 1982), S. 235-256, 1982
- [Borstnar02] Nils Borstnar, Eckhard Pabst und Hans Jürgen Wulff, "Einführung in die Film- und Fernsehwissenschaft", UVK Verlagsgesellschaft mbH, Konstanz, 2002
- [Braun05] A. Braun, B. E. Bodmann und S. R. Musse, "Simulating virtual crowds in emergency situations", in Proceedings of the ACM Symposium on Virtual Reality Software and Technology (Monterey, CA, USA, 7.-9. November 2005), VRST '05, S. 244-252, ACM Press, New York, NY, 2005
- [Breen92] D. E. Breen, D. H. House und P. H. Getto, "A physically-based particle model of woven cloth", The Visual Computer, Vol. 8, Nr. 5-6, September 1992, S. 264-277, Springer Berlin, Heidelberg, 1992
- [Brid02] R. Bridson, R. Fedkiw und J. Anderson, "Robust treatment of collisions, contact and friction for cloth animation", ACM Trans. Graph. 21, 3 (Jul. 2002), S. 594-603, 2002
- [Bros01] G. J. Brostow und I. Essa, "Image-based motion blur for stop motion animation", in Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques, SIGGRAPH '01, S. 561-566, ACM Press, New York, NY, 2001
- [Bung02] H.-J. Bungartz, M. Griebel und C. Zenger, „Einführung in die Computergraphik: Grundlagen, geometrische Modellierung, Algorithmen“, Vieweg, Braunschweig, 2002
- [Cardle03] M. Cardle, S. Brooks, Z. Bar-Joseph und P. Robinson, "Sound-by-numbers: motion-driven sound synthesis", in Proceedings of the 2003 ACM Siggraph/Eurographics Symposium on Computer Animation (San Diego, California, 26.-27. Juli 2003), Symposium on Computer Animation, S. 349-356, Eurographics Association, Aire-la-Ville, Schweiz, 2003
- [Carls02] M. Carlson, P. J. Mucha, R. B. Van Horn und G. Turk, "Melting and flowing", in Proceedings of the 2002 ACM Siggraph/Eurographics Symposium on Computer Animation (San Antonio, Texas, 21.-22. Juli 2002), SCA '02, S. 167-174, ACM Press, New York, NY, 2002
- [Carr03] J. Carranza, C. Theobalt, M. A. Magnor und H. Seidel, "Free-viewpoint video of human actors", in ACM SIGGRAPH 2003 Papers (San Diego, Kalifornien, 27.-31. Juli 2003), SIGGRAPH '03, S. 569-577, ACM Press, New York, NY, 2003
- [Chen01] Z. Chen, A. M. Fanelli, G. Castellano und L. C. Jain, "Introduction to computational intelligence paradigms", in N. Baba, L. Jain (Hrsg.), "Computational intelligence in games", Physica-Verl., Heidelberg, 2001

- [Chen02] W. Chen, J. Bouguet, M. H. Chu und R. Grzeszczuk, "Light field mapping: efficient representation and hardware rendering of surface light fields", ACM Trans. Graph. 21, 3 (Jul. 2002), S. 447-456, 2002
- [ChenY02] Y. Chen, Y. Xu, B. Guo und H. Shum, "Modeling and rendering of realistic feathers", ACM Trans. Graph. 21, 3 (Jul. 2002), S. 630-636, 2002
- [Chen05] Y. Chen, L. Xia, T. Wong, X. Tong, H. Bao, B. Guo und H. Shum, "Visual simulation of weathering by γ -ton tracing", ACM Trans. Graph. 24, 3 (Jul. 2005), S. 1127-1133, 2005
- [Choi02] K. Choi und H. Ko, "Stable but responsive cloth", ACM Trans. Graph. 21, 3 (Jul. 2002), S. 604-611, 2002
- [Chuang00] Y. Chuang, D. E. Zongker, J. Hindorff, B. Curless, D. H. Salesin und R. Szeliski, "Environment matting extensions: towards higher accuracy and real-time capture", in Proceedings of the 27th Annual Conference on Computer Graphics and interactive Techniques, International Conference on Computer Graphics and Interactive Techniques, S. 121-130, ACM Press/Addison-Wesley Publishing Co., New York, NY , 2000
- [Chuang02] Y. Chuang, A. Agarwala, B. Curless, D. H. Salesin, und R. Szeliski, "Video matting of complex scenes", ACM Trans. Graph. 21, 3, S. 243-248, Jul. 2002
- [Chuang03] Y. Chuang, D. B. Goldman, B. Curless, D. H. Salesin, und R. Szeliski, "Shadow matting and compositing", ACM Trans. Graph. 22, 3, S. 494-500, Jul. 2003
- [Clav05] S. Clavet, P. Beaudoin und P. Poulin, "Particle-based viscoelastic fluid simulation", in Proceedings of the 2005 ACM Siggraph/Eurographics Symposium on Computer Animation (Los Angeles, California, 29.-31. Juli 2005), SCA '05, S. 219-228, ACM Press, New York, NY, 2005
- [Cohen98] D. Cohen-Or, A. Solomovic und D. Levin, "Three-dimensional distance field metamorphosis", ACM Trans. Graph. 17, 2 (Apr. 1998), S. 116-141, 1998
- [Cord02] F. Cordier, N. Magnenat-Thalmann, "Real-Time Animation of Dressed Virtual Humans", Computer Graphics Forum, 21, 3, September 2002, S. 327-336, 2002
- [Corr98] W. T. Corrêa, R. J. Jensen, C. E. Thayer und A. Finkelstein, "Texture mapping for cel animation", in Proceedings of the 25th Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '98, S. 435-446, ACM Press, New York, NY, 1998
- [Cutler02] B. Cutler, J. Dorsey, L. McMillan, M. Müller und R. Jagnow, "A procedural approach to authoring solid models", ACM Trans. Graph. 21, 3 (Jul. 2002), S. 302-311, 2002
- [Debev02] P. Debevec, A. Wenger, C. Tchou, A. Gardner, J. Waese und T. Hawkins, "A lighting reproduction approach to live-action compositing", ACM Trans. Graph. 21, 3, S. 547-556, Jul. 2002

- [Des99] M. Desbrun, M.-P. Cani, "Space-time adaptive simulation of highly deformable substances", Tech. rep., INRIA, 1999
- [Dob00] Y. Dobashi, K. Kaneda, H. Yamashita, T. Okita und T. Nishita, "A simple, efficient method for realistic animation of clouds", in Proceedings of the 27th Annual Conference on Computer Graphics and interactive Techniques, International Conference on Computer Graphics and Interactive Techniques, S. 19-28, ACM Press/Addison-Wesley Publishing Co., New York, NY, 2000
- [Dob03] Y. Dobashi, T. Yamamoto und T. Nishita, "Real-time rendering of aerodynamic sound using sound textures based on computational fluid dynamics", ACM Trans. Graph. 22, 3, S. 732-740, Jul. 2003
- [Doel01] K. van den Doel, P. G. Kry und D. K. Pai, "FoleyAutomatic: physically-based sound effects for interactive simulation and animation", in Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '01, S. 537-544, ACM Press, New York, NY, 2001
- [Döll97] Jürgen Döllner und Klaus Hinrichs, "Object-Oriented 3D Modeling, Animation and Interaction", Journal of Visualization and Computer Animation, 8(1):33-64, 1997
- [Duff85] T. Duff, "Compositing 3-D rendered images", SIGGRAPH Comput. Graph. 19, 3, S. 41-44, Jul. 1985
- [Ebert90] D. S. Ebert und R. E. Parent, "Rendering and animation of gaseous phenomena by combining fast volume and scanline A-buffer techniques", SIGGRAPH Comput. Graph. 24, 4 (Sept. 1990), S. 357-366, 1990
- [Ell04] T. Ellman, "Specification and Synthesis of Hybrid Automata for Physics-Based Animation", Lecture Notes in Computer Science, Vol. 3018/2004, S. 54-55, Springer, Berlin/Heidelberg, 2004
- [Enr02] D. Enright, S. Marschner und R. Fedkiw, "Animation and rendering of complex water surfaces", in Proceedings of the 29th Annual Conference on Computer Graphics and interactive Techniques (San Antonio, Texas, 23.-26. Juli 2002), SIGGRAPH '02, ACM Press, New York, NY, S. 736-744, 2002
- [Ewald06] R. Ewald, D. Chen, G. K. Theodoropoulos, M. Lees, B. Logan, T. Oguara und A. M. Uhrmacher, "Performance Analysis of Shared Data Access Algorithms for Distributed Simulation of Multi-Agent Systems", in Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation (24.-26. Mai 2006), Workshop on Parallel and Distributed Simulation, S. 29-36, IEEE Computer Society, Washington, DC, 2006
- [Fang03] Fang, A. C. and Pollard, N. S. 2003. Efficient synthesis of physically valid human motion. ACM Trans. Graph. 22, 3 (Jul. 2003), 417-426.
- [Fear00] P. Fearing, "Computer modelling of fallen snow", in Proceedings of the 27th Annual Conference on Computer Graphics and interactive Techniques, International Conference on Computer Graphics and Interactive Techniques, S. 37-46, ACM Press/Addison-Wesley Publishing Co., New York, NY, 2000

- [Fedkiw01] R. Fedkiw, J. Stam und H. W. Jensen, "Visual simulation of smoke", in Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques, SIGGRAPH '01, S. 15-22, ACM Press, New York, NY, 2001
- [Feld03] B. E. Feldman, J. F. O'Brien und O. Arikan, "Animating suspended particle explosions", ACM Trans. Graph. 22, 3 (Jul. 2003), S. 708-715, 2003
- [Fiume87] E. Fiume, D. Tschritzis und L. Dami, "A Temporal Scripting Language for Object-Oriented Animation", in Proc. Eurographics'87, North-Holland, Elsevier Science Publishers, Amsterdam, 1987
- [Foley93] J. D. Foley, "Computer graphics: principles and practice", Addison-Wesley, Reading, Mass., 1993
- [Foley97] J. D. Foley, "Introduction to computer graphics", Addison-Wesley, Reading, Mass., 1997
- [Fourn86] A. Fournier und W. T. Reeves, "A simple model of ocean waves", SIGGRAPH Comput. Graph. 20, 4 (Aug. 1986), S. 75-84, 1986
- [Foster96] N. Foster und D. Metaxas, "Realistic animation of liquids", Graph. Models Image Process., 58, 5 (Sep. 1996), S. 471-483, 1996
- [Foster97] N. Foster und D. Metaxas, "Modeling the motion of a hot, turbulent gas", in Proceedings of the 24th Annual Conference on Computer Graphics and interactive Techniques, International Conference on Computer Graphics and Interactive Techniques, S. 181-188, ACM Press/Addison-Wesley Publishing Co., New York, NY, 1997
- [Foster01] N. Foster und R. Fedkiw, "Practical animation of liquids", in Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques, SIGGRAPH '01, S. 23-30, ACM Press, New York, NY, 2001
- [Funge99] J. Funge, X. Tu und D. Terzopoulos, "Cognitive modeling: knowledge, reasoning and planning for intelligent characters", in Proceedings of the 26th Annual Conference on Computer Graphics and interactive Techniques, International Conference on Computer Graphics and Interactive Techniques, S. 29-38, ACM Press/Addison-Wesley Publishing Co., New York, NY, 1999
- [Funk98] T. Funkhouser, I. Carlbom, G. Elko, G. Pingali, M. Sondhi und J. West, "A beam tracing approach to acoustic modeling for interactive virtual environments", in Proceedings of the 25th Annual Conference on Computer Graphics and interactive Techniques, SIGGRAPH '98, S. 21-32, ACM Press, New York, NY, 1998
- [Funk99] T. Funkhouser, P. Min und I. Carlbom, "Real-time acoustic modeling for distributed virtual environments", in Proceedings of the 26th Annual Conference on Computer Graphics and interactive Techniques, International Conference on Computer Graphics and Interactive Techniques, S. 365-374, ACM Press/Addison-Wesley Publishing Co., New York, NY, 1999

- [Gard84] G. Y. Gardner, "Simulation of natural scenes using textured quadric surfaces", SIGGRAPH Comput. Graph. 18, 3 (Jul. 1984), S. 11-20, 1984
- [Gard85] G. Y. Gardner, "Visual simulation of clouds", SIGGRAPH Comput. Graph. 19, 3 (Jul. 1985), S. 297-304, 1985
- [Gerv94] M. Gervautz und O. Beltcheva, "An Approach for Object-Oriented Animation Design", Institut für Computergraphik, Technische Universität Wien, Österreich, 1994
- [Getto90] P. Getto und D. Breen, "An object-oriented architecture for a computer animation system", the Visual Computer, Vol. 6, Nr. 2 / März 1990, S. 79-92, Springer Berlin / Heidelberg, 1990
- [Giesen00] Rolf Giesen, Claudia Meglin et. al., "Künstliche Welten: Tricks, Special Effects und Computeranimation im Film von den Anfängen bis heute", Europa Verlag GmbH, Hamburg/Wien, Originalausgabe, September 2000
- [Giesen00e] R. Giesen, "Die Entwicklung der Spezialeffekte", in [Giesen00]
- [Giesen00k] R. Giesen, "Künstliche Welten im Film", in [Giesen00]
- [Giesen00s] R. Giesen, "Spezialeffekte made in Germany", in [Giesen00]
- [Giesen00u] F. Petzold, "Ein Unsichtbarer unter Wasser", in [Giesen00]
- [Gora84] C. M. Goral, K. E. Torrance, D. P. Greenberg und B. Battaile, "Modeling the interaction of light between diffuse surfaces", SIGGRAPH Comput. Graph. 18, 3, S. 213-222, Jul. 1984
- [Green04] S. T. Greenwood und D. H. House, "Better with bubbles: enhancing the visual realism of simulated fluid", in Proceedings of the 2004 ACM Siggraph/Eurographics Symposium on Computer Animation (Grenoble, Frankreich, 27.-29. August 2004), SCA '04, S. 287-296, ACM Press, New York, NY, 2004
- [Guend05] E. Guendelman, A. Selle, F. Losasso und R. Fedkiw, "Coupling water and smoke to thin deformable and rigid shells", ACM Trans. Graph. 24, 3 (Jul. 2005), S. 973-981, 2005
- [Guti05] D. Gutierrez, I. Armenteros und B. Frischer, "Predictive crowd simulations for Cultural Heritage applications", in Proceedings of the 3rd international Conference on Computer Graphics and interactive Techniques in Australasia and South East Asia (Dunedin, Neuseeland, 29. November - 2. Dezember 2005), GRAPHITE '05, S. 109-112, ACM Press, New York, NY, 2005
- [Guzdial06] M. Guzdial und B. Ericson, "Problem Solving with Data Structures: A Multimedia Approach", College of Computing/GVU, Georgia Institute of Technology, 15. August 2006
- [Hab02] A. Habibi und A. Luciani, "Dynamic Particle Coating", IEEE Transactions on Visualization and Computer Graphics, S. 383-394, Oktober 2002

- [Hadap01] S. Hadap, N. Magnenat-Thalmann, "Modeling dynamic hair as a continuum", in Eurographics Proceedings, Computer Graphics Forum, Vol.20, Nr. 3, 2001
- [Haev04] W. Van Haevre, F. D. Fiore, P. Bekaert und F. Van Reeth, "A ray density estimation approach to take into account environment illumination in plant growth simulation", in Proceedings of the 20th Spring Conference on Computer Graphics (Budmerice, Slowakei, 22.-24. April 2004), SCCG '04, S. 121-131, ACM Press, New York, NY, 2004
- [Harris03] M. J. Harris, W. V. Baxter, T. Scheuermann, und A. Lastra, "Simulation of cloud dynamics on graphics hardware", in Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (San Diego, Kalifornien, 26.-27. Juli 2003), SIGGRAPH/EUROGRAPHICS Workshop On Graphics Hardware, S. 92-101, Eurographics Association, Aire-la-Ville, Switzerland, 2003
- [Hery04] C. Hery, H. Ono und D. Sutton, "Part 2: Crowd Systems at ILM", April 15, 2004, in [Thal04]
- [Holman00] T. Holman, "5.1 Surround Sound: Up and Running", Von Tomlinson Holman, Focal Press, Boston, MA, 2000
- [Hubb03] S. Hubbard, „Elements of Digital Effects“, in „Handbook of computer animation“, Springer, London, 2003
- [Jack06] D. Jackèl, S. Neunreither, F. Wagner, "Methoden der Computeranimation", Springer, Berlin, 2006
- [Jens02] H. W. Jensen und J. Buhler, "A rapid hierarchical rendering technique for translucent materials", ACM Trans. Graph. 21, 3 (Jul. 2002), S. 576-581, 2002
- [Jens96] H. W. Jensen, "Global Illumination Using Photon Maps“, in Rendering Techniques '96, S. 21-30, Springer-Verlag, Wien, 1996
- [Kacic03] Z. Kacic-Alesic, M. Nordenstam und D. Bullock, "A practical dynamics system", in SCA '03: ACM SIGGRAPH/Eurographics Symposium on Computer animation, S. 7-16, Eurographics Association, 2003
- [Kaji84] J. T. Kajiya und B. P. von Herzen, "Ray tracing volume densities", SIGGRAPH Comput. Graph. 18, 3 (Jul. 1984), S. 165-174, 1984
- [Kalra92] D. Kalra und A. H. Barr, "Modeling with Time and Events in Computer Animation", EUROGRAPHICS '92, Volume 11, (1992), number 3, Blackwell Publishers, © Eurographics Association, 1992
- [KaKl93] P. Kandzia, H.-J. Klein, "Theoretische Grundlagen relationaler Datenbanksysteme", BI-Wiss.-Verl., Mannheim, Leipzig, Wien, Zürich, 1993
- [Kamp04] A. Kamphuis und M. H. Overmars, "Finding paths for coherent groups using clearance", in Proceedings of the 2004 ACM Siggraph/Eurographics Symposium on Computer Animation (Grenoble, Frankreich, 27.-29. August 2004), SCA '04, S. 19-28, ACM Press, New York, NY, 2004

- [Kanev02] K. Kanev, S. Kimura, "Integrating Dynamic Full-Body Motion Devices in Interactive 3D Entertainment", IEEE Computer Graphics and Applications, Vol. 22, Nr. 4, S. 76-86, Juli/August, 2002
- [Kass90] M. Kass und G. Miller, "Rapid, stable fluid dynamics for computer graphics", SIGGRAPH Comput. Graph. 24, 4 (Sept. 1990), S. 49-57, 1990
- [Kim04] T. Kim, M. Henson und M. C. Lin, "A hybrid algorithm for modeling ice formation", in Proceedings of the 2004 ACM Siggraph/Eurographics Symposium on Computer Animation (Grenoble, Frankreich, 27.-29. August 2004), SCA '04, S. 305-314, ACM Press, New York, NY, 2004
- [Kloss98] J. Kloss, R. Rockwell, K. Szabó und M. Duchow, "VRML 97: der internationale Standard für interaktive 3D-Welten im World Wide Web", 1. Auflage, Addison-Wesley-Longman, Bonn, 1998
- [Kov02] L. Kovar, J. Schreiner und M. Gleicher, "Footskate cleanup for motion capture editing", in Proceedings of the 2002 ACM Siggraph/Eurographics Symposium on Computer Animation (San Antonio, Texas, 21.-22. Juli, 2002), SCA '02, S. 97-104, ACM Press, New York, NY, 2002
- [Kow99] M. A. Kowalski, L. Markosian, J. D. Northrup, L. Bourdev, R. Barzel, L. S. Holden und J. F. Hughes, "Art-based rendering of fur, grass, and trees", in Proceedings of the 26th Annual Conference on Computer Graphics and interactive Techniques, International Conference on Computer Graphics and Interactive Techniques, S. 433-438, ACM Press/Addison-Wesley Publishing Co., New York, NY, 1999
- [Lam02] A. Lamorlette und N. Foster, "Structural modeling of flames for a production environment", ACM Trans. Graph. 21, 3 (Jul. 2002), S. 729-735, 2002
- [Lau02] R. W. Lau, O. Chan, M. Luk und F. W. Li, "LARGE a collision detection framework for deformable objects", in Proceedings of the ACM Symposium on Virtual Reality Software and Technology (Hong Kong, China, 11.-13. November 2002) , S. 113-120, VRST '02, ACM Press, New York, NY, 2002
- [Lengy01] J. Lengyel, E. Praun, A. Finkelstein und H. Hoppe, "Real-time fur over arbitrary surfaces", in Proceedings of the 2001 Symposium on interactive 3D Graphics, SI3D '01, S. 227-232, ACM Press, New York, NY, 2001
- [Lin98] M. Lin und S. Gottschalk, "Collision Detection between Geometric Models: A Survey", in Proc. IMA Conf. on Mathematics of Surfaces, S. 37-56, 1998
- [Losa06a] F. Losasso, G. Irving, E. Guendelman und Ron Fedkiw, "Melting and Burning Solids into Liquids and Gases", in IEEE Transactions on Visualization and Computer Graphics, Mai 2006
- [Losa06] F. Losasso, T. Shinar, A. Selle und R. Fedkiw, "Multiple interacting liquids", ACM Trans. Graph. 25, 3 (Jul. 2006), S. 812-819, 2006
- [Luck97] V. Luckas, T. Broll, "CASUS; an object-oriented three-dimensional animation system for event-oriented simulators", Computer Animation '97, S. 144-150, Genf, Schweiz, 5.-6. Jun. 1997

- [Mass03] V. Masselus, P. Peers, P. Dutré und Y. D. Willems, "Relighting with 4D incident light fields", in ACM SIGGRAPH 2003 Papers (San Diego, Kalifornien, 27.-31. Juli 2003), SIGGRAPH '03, ACM Press, New York, NY, S. 613-620, 2003
- [Matus02] W. Matusik, H. Pfister, A. Ngan, P. Beardsley, R. Ziegler und L. McMillan, "Image-based 3D photography using opacity hulls", ACM Trans. Graph. 21, 3, S. 427-437, Jul. 2002
- [McKen90] M. McKenna, S. Pieper und D. Zeltzer, "Control of a virtual actor: the roach", SIGGRAPH Comput. Graph. 24, 2 (März 1990), S. 165-174, 1990
- [Melek05] Z. Melek und J. Keyser, "Multi-representation interaction for physically based modelling", in Proceedings of the 2005 ACM Symposium on Solid and Physical Modeling (Cambridge, Massachusetts, 13.-15. Juni 2005), SPM '05, S. 187-196, ACM Press, New York, NY, 2005
- [Mill89] Gavin Miller und Andrew Pearce, "Globular Dynamics: A Connected Particle System for Animating Viscous Fluids", Comput. & Graphics Vol. 13, No. 3, S. 305-309, 1989
- [Mirt00] B. Mirtich, "Timewarp rigid body simulation", in Proceedings of the 27th Annual Conference on Computer Graphics and interactive Techniques, International Conference on Computer Graphics and Interactive Techniques, S. 193-200, ACM Press/Addison-Wesley Publishing Co., New York, NY, 2000
- [Moriya01] T. Moriya, H. Takeda, "Image Generation for Immersive Multi-Screen Environment with a Motion Ride", S. 297, IEEE Virtual Reality Conference 2001 (VR 2001), 2001
- [Müller98] W. Müller, R. Dörner, V. Luckas und A. Schäfer, "An Efficient Object-Oriented Authoring and Presentation System for Virtual Environments", Proceedings of GraphiCon'98, S. 111-118, Moscow, Russia, 1998
- [Müller03] M. Müller, D. Charypar, M. Gross, "Particle-based fluid simulation for interactive applications", in Proceedings of the 2003 ACM Siggraph/Eurographics Symposium on Computer Animation (San Diego, Kalifornien, 26.-27. Juli 2003), Symposium on Computer Animation, Eurographics Association, Aire-la-Ville, Schweiz, S. 154-159, 2003
- [Naef02] M. Naef, O. Staadt und M. Gross, "Spatialized audio rendering for immersive virtual environments", in Proceedings of the ACM Symposium on Virtual Reality Software and Technology (Hong Kong, China, 11.-13. November 2002), VRST '02, S. 65-72, ACM Press, New York, NY, 2002
- [Naka06] T. Nakamoto, K. Yoshikawa, "Movie with Scents Generated by Olfactory Display Using Solenoid Valves", S. 291-292, IEEE Virtual Reality Conference (VR 2006), 2006
- [Neyret98] F. Neyret, "Modeling, Animating, and Rendering Complex Scenes Using Volumetric Textures", in IEEE Transactions on Visualization and Computer Graphics, S. 55-70, Januar 1998

- [Ng96] H. N. Ng, R. L. Grimsdale, „Computer Graphics Techniques for Modeling Cloth“, in: Computer Graphics and Applications, IEEE, Vol. 16, Ausg. 5, S. 28-41, IEEE Computer Society, Sept. 1996
- [Nish85] T. Nishita und E. Nakamae, “Continuous tone representation of three-dimensional objects taking account of shadows and interreflection”. SIGGRAPH Comput. Graph. 19, 3, S. 23-30, Jul. 1985
- [Mysz02] K. Myszkowski, “Perception-based global illumination, rendering, and animation techniques”, in Proceedings of the 18th Spring Conference on Computer Graphics (Budmerice, Slovakia, 24.-27. April 2002), SCCG '02, S. 13-24, ACM Press, New York, NY, 2002
- [OBrien95] J. F. O'Brien und J. K. Hodgins, “Dynamic simulation of splashing fluids”, in Computer Animation 1995, S. 198-205, April 1995
- [OBrien02] J. F. O'Brien, C. Shen und C. M. Gatchalian, “Synthesizing sounds from rigid-body simulations”, in Proceedings of the 2002 ACM Siggraph/Eurographics Symposium on Computer Animation (San Antonio, Texas, 21.-22. Juli, 2002), SCA '02, S. 175-181, ACM Press, New York, NY, 2002
- [Pai01] D. K. Pai, K. v. Doel, D. L. James, J. Lang, J. E. Lloyd, J. L. Richmond und S. H. Yau, “Scanning physical interaction behavior of 3D objects“, in Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '01, S. 87-96, ACM Press, New York, NY, 2001
- [Parish01] Y. I. Parish und P. Müller, “Procedural modeling of cities”, in Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques, SIGGRAPH '01, S. 301-308, ACM Press, New York, NY, 2001
- [Peach86] D. R. Peachey, “Modeling waves and surf”, SIGGRAPH Comput. Graph. 20, 4 (Aug. 1986), S. 65-74, 1986
- [Perlin85] K. Perlin, “An image synthesizer”, SIGGRAPH Comput. Graph. 19, 3 (Jul. 1985), S. 287-296, 1985
- [Petrov00] L. Petrović, B. Fujito, L. Williams und A. Finkelstein, ”Shadows for cel animation”, in Proceedings of the 27th Annual Conference on Computer Graphics and interactive Techniques, International Conference on Computer Graphics and Interactive Techniques, S. 511-516, ACM Press/Addison-Wesley Publishing Co., New York, NY, 2000
- [Piegl91] L. Piegl, “On NURBS: a Survey“, IEEE Computer Graphics and Applications, Januar 1991
- [Pigh04] F. Pighin, J. M. Cohen und M. Shah, “Modeling and editing flows using advected radial basis functions”, in Proceedings of the 2004 ACM Siggraph/Eurographics Symposium on Computer Animation (Grenoble, Frankreich, 27.-29. August 2004), SCA '04, S. 223-232, ACM Press, New York, NY, 2004

- [Plenge88] A. Plenge, "AMIGA-3-D-Grafik und Animation", Markt-u.-Technik-Verlag, Haar bei München, 1988
- [Popes00] V. Popescu, J. Eyles, A. Lastra, J. Steinhurst, N. England und L. Nyland, "The WarpEngine: an architecture for the post-polygonal age", in Proceedings of the 27th Annual Conference on Computer Graphics and interactive Techniques International Conference on Computer Graphics and Interactive Techniques, S. 433-442, ACM Press/Addison-Wesley Publishing Co., New York, NY, 2000
- [Por84] T. Porter und T. Duff, "Compositing digital images", SIGGRAPH Comput. Graph. 18, 3, S. 253-259, Jul. 1984
- [Praun01] E. Praun, H. Hoppe, M. Webb, A. Finkelstein, "Real-time hatching", in Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques, SIGGRAPH '01, S. 581-586, ACM Press, New York, NY, 2001
- [Prem03] S. Premoze, T. Tasdizen, J. Bigler, A. Lefohn, R. Whitaker, "Particle-based simulation of fluids", Computer Graphics Forum 22, 3, 2003
- [Qin96] H. Qin und D. Terzopoulos, "D-NURBS: A Physics-Based Framework for Geometric Design", IEEE Transactions on Visualization and Computer Graphics, März 1996
- [Rah03] N. A. Rahrmann und T. Tetiker, "Herr der Ringe: Die zwei Türme", Digital Production, 1/03, S. 46-50, 2003
- [Ras03] N. Rasmussen, D. Q. Nguyen, W. Geiger und R. Fedkiw, "Smoke simulation for large scale phenomena", ACM Trans. Graph. 22, 3 (Jul. 2003), S. 703-707, 2003
- [Reeves83] W. T. Reeves, "Particle systems - a technique for modeling a class of fuzzy objects", SIGGRAPH Comput. Graph. 17, 3, S. 359-375, Jul. 1983
- [Reeves85] W. T. Reeves und R. Blau, "Approximate and probabilistic algorithms for shading and rendering structured particle systems", SIGGRAPH Comput. Graph. 19, 3, S. 313-322, Jul. 1985
- [Reg04] Stephen Regelous, "Part 4", 2004, in [Thal04]
- [Reitsma03] P. S. Reitsma und N. S. Pollard, "Perceptual metrics for character animation: sensitivity to errors in ballistic motion", ACM Trans. Graph. 22, 3, S. 537-542, Jul. 2003
- [Reyn82] C. W. Reynolds, "Computer animation with scripts and actors", SIGGRAPH Comput. Graph. 16, 3 (Jul. 1982), S. 289-296, 1982
- [Reyn87] C. W. Reynolds, "Flocks, herds, and schools: A distributed behavioral model", in Computer Graphics, Vol. 21(4), S. 25-34, USA, 1987
- [Rob06] B. Robertson, "Fluch der Karibik 2: Mocap-Piraten und Tentakel", Digital Production, Ausg. 05:06, S. 28-33, 2006

- [Rosenb03] V. Rosenberger, "Herr der Ringe: Massenszenen in der Praxis", Digital Production, 4/03, S. 40-46, 2003
- [Rudo05] I. Rudomín und E. Millán, "Probabilistic, layered and hierarchical animated agents using XML", in Proceedings of the 3rd international Conference on Computer Graphics and interactive Techniques in Australasia and South East Asia (Dunedin, Neuseeland, 29. November - 2. Dezember 2005), GRAPHITE '05, S. 113-116, ACM Press, New York, NY, 2005
- [Sakas93] G. Sakas, "Modeling and animating turbulent gaseous phenomena using spectral synthesis", The Visual Computer, Vol. 9, Nr. 4, April 1993, S. 200-212, Springer Berlin/Heidelberg, 1993
- [Samoil98] T. Samoilov und G. Elber, "Self-intersection elimination in metamorphosis of two dimensional curves", Vis. Comp. 14, S. 415-428, 1998
- [Schödl02] A. Schödl und I. A. Essa, "Controlled animation of video sprites", in Proceedings of the 2002 ACM Siggraph/Eurographics Symposium on Computer Animation (San Antonio, Texas, 21.-22. Juli 2002), SCA '02, S. 121-127, ACM Press, New York, NY, 2002
- [Schpok03] J. Schpok, J. Simons, D. S. Ebert und C. Hansen, "A real-time cloud modeling, rendering, and animation system", in Proceedings of the 2003 ACM Siggraph/Eurographics Symposium on Computer Animation (San Diego, Kalifornien, 26.-27. Juli 2003), Symposium on Computer Animation, S. 160-166, Eurographics Association, Aire-la-Ville, Switzerland, 2003
- [Schpok05] J. Schpok, W. Dwyer und D. S. Ebert "Modeling and animating gases with simulation features", in Proceedings of the 2005 ACM Siggraph/Eurographics Symposium on Computer Animation (Los Angeles, Kalifornien, 29.-31. Juli 2005), SCA '05, S. 97-105, ACM Press, New York, NY, 2005
- [Seder93] T. W. Sederberg, P. Gao, G. Wang und H. Mu, "2-D shape blending: an intrinsic solution to the vertex path problem", in Proceedings of the 20th Annual Conference on Computer Graphics and interactive Techniques, SIGGRAPH '93, S. 15-18, ACM Press, New York, NY, 1993
- [Sell05] A. Selle, N. Rasmussen und R. Fedkiw, "A vortex particle method for smoke, water and explosions", ACM Trans. Graph. 24, 3 (Jul. 2005), S. 910-914, 2005
- [Shin01] H. J. Shin, J. Lee, S. Y. Shin und M. Gleicher, "Computer puppetry: An importance-based approach", ACM Trans. Graph. 20, 2 (Apr. 2001), S. 67-94, 2001
- [Silver97] D. Silver und X. Wang, "Tracking and Visualizing Turbulent 3D Features", in IEEE Transactions on Visualization and Computer Graphics, S. 129-141, April 1997
- [Sims94] K. Sims, "Evolving virtual creatures", in Proceedings of the 21st Annual Conference on Computer Graphics and interactive Techniques, SIGGRAPH '94, S. 15-22, ACM Press, New York, NY, 1994

- [Smith02] J. Smith, J. Hodgins, I. Oppenheim und A. Witkin, "Creating models of truss structures with optimization", in Proceedings of the 29th Annual Conference on Computer Graphics and interactive Techniques (San Antonio, Texas, 23.-26. Juli 2002), SIGGRAPH '02, S. 295-301, ACM Press, New York, NY, 2002
- [Stam93] J. Stam und E. Fiume, "Turbulent wind fields for gaseous phenomena", in Proceedings of the 20th Annual Conference on Computer Graphics and interactive Techniques, SIGGRAPH '93, S. 369-376, ACM Press, New York, NY, 1993
- [Stam95] J. Stam und E. Fiume, "Depicting fire and other gaseous phenomena using diffusion processes", in Proceedings of the 22nd Annual Conference on Computer Graphics and interactive Techniques, Hrsg. S. G. Mair und R. Cook, SIGGRAPH '95, S. 129-136, ACM Press, New York, NY, 1995
- [Stam97] J. Stam, "A General Animation Framework for Gaseous Phenomena", Forschungsbericht R047, ERCIM (European Research Consortium for Informatics and Mathematics), 1997
- [Steph03] I. Stephenson, „Rendering and Shading“, in „Handbook of computer animation“, Springer, London, 2003
- [Stoer73] J. Stoer und R. Bulirsch, "Einführung in die Numerische Mathematik II“, Springer-Verlag, Berlin, Heidelberg, New York, 1973
- [Stroth02] T. Strothotte und S. Schlechtweg, "Non-Photorealistic Computer Graphics: Modeling, Rendering and Animation", Morgan Kaufmann, 2002
- [Stur94] D. J. Sturman, "A Brief History of Motion Capture for Computer Character Animation", in: Course 9, Siggraph '94, 1994
- [Sullivan99] C. O'Sullivan, "Perceptually-Adaptive Collision Detection for Real-time Computer Animation", Thesis submitted for the degree of Doctor of Philosophy in Computer Science, University of Dublin, Trinity College, Department of Computer Science, June 14th, 1999
- [Sullivan01] C. O'Sullivan und J. Dingliana, "Collisions and perception", ACM Trans. Graph. 20, 3 (Jul. 2001), S. 151-168, 2001
- [Sullivan03] C. O'Sullivan, J. Dingliana, T. Giang und M. K. Kaiser, "Evaluating the visual fidelity of physically based animations", ACM Trans. Graph. 22, 3, S. 527-536, Jul. 2003
- [Sun02] J. Sun, X. Yu, G. Baciú und M. Green, "Template-based generation of road networks for virtual city modelling", in Proceedings of the ACM Symposium on Virtual Reality Software and Technology (Hong Kong, China, 11.-13. November 2002), VRST '02, S. 33-40, ACM Press, New York, NY, 2002
- [Sura01] V. Surazhsky und C. Gotsman, "Controllable morphing of compatible planar triangulations", ACM Trans. Graph. 20, 4 (Okt. 2001), S. 203-231, 2001

- [Terz88] D. Terzopoulos und K. Fleischer, "Modeling inelastic deformation: viscoelasticity, plasticity, fracture", SIGGRAPH Comput. Graph. 22, 4 (Aug. 1988), S. 269-278, 1988
- [Terz94] D. Terzopoulos, H. Qin, "Dynamic NURBS with geometric constraints for interactive sculpting", ACM Transactions on Graphics (TOG) archive, Volume 13, Issue 2 (April 1994), Special issue on interactive sculpting, S.103-136, ACM Press, New York, NY, USA, 1994
- [Tess02] J. Tessendorf, "Simulating Ocean Water", in: Course 09 (SIGGRAPH '02), S. 45-66, 2002
- [Thal04] D. Thalmann, C. Hery, S. Lippman, H. Ono, S. Regelous und D. Sutton, "Crowd and group animation", in ACM SIGGRAPH 2004 Course Notes (Los Angeles, CA, 8.-12. August 2004), SIGGRAPH '04, ACM Press, New York, NY, 2004
- [Tole02] P. Tole, F. Pellacini, B. Walter und D. P. Greenberg, "Interactive global illumination in dynamic scenes", ACM Trans. Graph. 21, 3 (Jul. 2002), S. 537-546, 2002
- [Tsing01] N. Tsingos, T. Funkhouser, A. Ngan und I. Carlbom, "Modeling acoustics in virtual environments using the uniform theory of diffraction", in Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '01, S. 545-552, ACM Press, New York, NY, 2001
- [Tu94] X. Tu und D. Terzopoulos, "Perceptual modeling for the behavioral animation of fishes", in Proceeding of the Second Pacific Conference on Fundamentals of Computer Graphics (Beijing, China), J. Chen, N. Magnenat Thalmann, Z. Tang, and D. Thalmann, Hrsg., World Scientific Publishing Co., River Edge, NJ, S. 185-200, 1994
- [Tu99] Xiaoyuan Tu, "Artificial animals for computer animation", Lecture notes in computer science, 1635, Springer, Berlin, 1999 (Zugl.: Toronto, Univ., Diss., 1996)
- [Umble06] E. A. Umble, "Making it real: the future of stereoscopic 3D film technology", SIGGRAPH Comput. Graph. 40, 1 (Mai 2006), 2006
- [Vill05] J. Villard und H. Borouchaki, "Adaptive meshing for cloth animation", Engineering with Computers, 20, 4 (Aug. 2005), S. 333-341, 2005
- [Vol04] P. Volino und N. Magnenat-Thalmann, "Animating complex hairstyles in real-time", in Proceedings of the ACM Symposium on Virtual Reality Software and Technology (Hong Kong, 10.-12. November 2004), VRST '04, S. 41-48, ACM Press, New York, NY, 2004
- [Vol06] P. Volino und N. Magnenat-Thalmann, "Real-Time Animation of Complex Hairstyles", in IEEE Transactions on Visualization and Computer Graphics, März 2006
- [Wang02] R. Wang, "Adaptive Cloth Simulation", Diplomarbeit, Carnegie Mellon University, 2002

- [Wang05] H. Wang, P. J. Mucha und G. Turk, "Water drops on surfaces", *ACM Trans. Graph.* 24, 3 (Jul. 2005), S. 921-929, 2005
- [Ward99] G. Ward und M. Simmons, "The holodeck ray cache: an interactive rendering system for global illumination in nondiffuse environments", *ACM Trans. Graph.* 18, 4 (Okt. 1999), S. 361-368, 1999
- [Watt01] A. H. Watt und F. Policarpo, "3D games - Real-time rendering and software technology", Addison-Wesley, Harlow, 2001
- [Wei03] X. Wei, Y. Zhao, Z. Fan, W. Li, S. Yoakum-Stover und A. Kaufman, "Blowing in the wind", in *Proceedings of the 2003 ACM Siggraph/Eurographics Symposium on Computer Animation (San Diego, Kalifornien, 26.-27. Juli 2003)*, Symposium on Computer Animation, S. 75-85, Eurographics Association, Aire-la-Ville, Switzerland, 2003
- [Wejch91] J. Wejchert und D. Haumann, "Animation aerodynamics", in *Proceedings of the 18th Annual Conference on Computer Graphics and interactive Techniques, SIGGRAPH '91*, S. 19-22, ACM Press, New York, NY, 1991
- [Whit80] T. Whitted, "An improved illumination model for shaded display", *Commun. ACM* 23, 6, S. 343-349, Jun. 1980
- [Witkin88] A. Witkin und M. Kass, "Spacetime constraints", *SIGGRAPH Comput. Graph.* 22, 4 (Aug. 1988), S. 159-168, 1988
- [Witt99] P. Witting, "Computational fluid dynamics in a traditional animation environment", in *Proceedings of the 26th Annual Conference on Computer Graphics and interactive Techniques, International Conference on Computer Graphics and Interactive Techniques*, S. 129-136, ACM Press/Addison-Wesley Publishing Co., New York, NY, 1999
- [Wonk03] P. Wonka, M. Wimmer, F. Sillion und W. Ribarsky, "Instant architecture", in *ACM SIGGRAPH 2003 Papers (San Diego, Kalifornien, 27.-31. Juli 2003)*, SIGGRAPH '03, S. 669-677, ACM Press, New York, NY, 2003
- [Xu01] Y. Xu, Y. Chen, S. Lin, H. Zhong, E. Wu, B. Guo, und H. Shum, "Photorealistic rendering of knitwear using the lumislice", in *Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '01*, S. 391-398, ACM Press, New York, NY, 2001
- [Yaeg86] L. Yaeger, C. Upson und R. Myers, "Combining physical and visual simulation - creation of the planet Jupiter for the film '2010'", *SIGGRAPH Comput. Graph.* 20, 4 (Aug. 1986), S. 85-93, 1986
- [Yee01] H. Yee, S. Pattanaik und D. P. Greenberg, "Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments", *ACM Trans. Graph.* 20, 1 (Jan. 2001), S. 39-65, 2001

- [Yngve00] G. D. Yngve, J. F. O'Brien und J. K. Hodgins, "Animating explosions", in Proceedings of the 27th Annual Conference on Computer Graphics and interactive Techniques, International Conference on Computer Graphics and Interactive Techniques, S. 29-36, ACM Press/Addison-Wesley Publishing Co., New York, NY, 2000
- [Zelev91] R. C. Zelevnik, D. B. Conner, M. M. Wloka, D. G. Aliaga, N. T. Huang, P. M. Hubbard, B. Knep, H. Kaufman, J. F. Hughes und A. van Dam, "An object-oriented framework for the integration of interactive animation techniques", in Proceedings of the 18th Annual Conference on Computer Graphics and interactive Techniques, SIGGRAPH '91, S. 105-112, ACM Press, New York, NY, 1991
- [Zhang02] Z. Zhang, L. Wang, B. Guo und H. Shum, "Feature-based light field morphing", ACM Trans. Graph. 21, 3 (Jul. 2002), S. 457-464, 2002
- [Zord03] V. B. Zordan und N. C. Van Der Horst "Mapping optical motion capture data to skeletal motion using a physical model", in Proceedings of the 2003 ACM Siggraph/Eurographics Symposium on Computer Animation (San Diego, Kalifornien, 26.-27. Juli, 2003), S. 245-250, Symposium on Computer Animation. Eurographics Association, Aire-la-Ville, Schweiz, 2003
- [Zord05] V. B. Zordan, A. Majkowska, B. Chiu und M. Fast, "Dynamic response for motion capture animation", ACM Trans. Graph. 24, 3 (Jul. 2005), S. 697-701, 2005

15 Auszug aus der Klassenstruktur von MaxControl

Dieser Auszug aus der Klassenstruktur von MaxControl wurde mit Javadoc⁶⁹ generiert. Dabei entspricht die Einrückung der Klassenstruktur.

- **java.lang.Object**
 - **javax.vecmath.AxisAngle4d** (implements **java.lang.Cloneable**, **java.io.Serializable**)
 - **MCH_AxisAngleDegrees**
 - **MC_C**
 - **MC_Entity**
 - **MC_MaterialAttribute**
 - **MCMA_Conducting**
 - **MCMA_Superconducting**
 - **MCMA_Deformable**
 - **MCMA_Hard**
 - **MCMA_Hollow**
 - **MCMA_Magnetic**
 - **MCMA_Rough**
 - **MCMA_Slimily**
 - **MCMA_Slippery**
 - **MCMA_Smooth**
 - **MCMA_Soft**
 - **MCMA_Sticky**
 - **MCMA_Wet**
 - **MC_MaterialDescriptor**
 - **MCMD_Asphalt**
 - **MCMD_Concrete**
 - **MCMD_Dirt**
 - **MCMD_Grass**
 - **MCMD_Metal**
 - **MCMD_Rubber**
 - **MCMD_Wood**
 - **MC_MaterialEvaluator**
 - **MCME_WoodTester**
 - **MC_Owner**
 - **MC_Behaviour**
 - **MCB_2DNavigation**
 - **MCB_2DNavigation_with_Avoider2D_uAB2DCLB**⁷⁰
 - **MCB_6LegRobotMainBody**
 - **MCB_AutoPSimSubStepChanger**
 - **MCB_AvoidBorder**
 - **MCB_AvoidBorder2D**
 - **MCB_AvoidBorder2DCylindrical**
 - **MCB_AvoidBorder2DCylindricalLegBased**
 - **MCB_Avoider2D**
 - **MCB_Avoider2D_uAB2DCLB**⁷¹

⁶⁹ <http://java.sun.com/j2se/javadoc/>

⁷⁰ Die Bezeichnung dieser Klasse wurde in Text als „MCB_2DNavigation_with_Avoider2D“ abgekürzt, „_uAB2DCLB“ steht für „using MCB_AvoidBorder2DCylindricalLegBased“.

- **MCB_Camera**
- **MCB_CameraBasicControl**
- **MCB_CameraJumpSoundSilencer**
- **MCB_CameraPointSelector**
- **MCB_CarAutoGear**
- **MCB_CarControlAI**
- **MCB_CarControlBasic**
- **MCB_CarLightController**
- **MCB_CarTyreControl**
- **MCB_CarTyreLightController**
- **MCB_ContactSFXController**
 - **MCB_ImpactSFXController**
 - **MCB_RollingSFXController**
 - **MCB_SkiddingSFXController**
- **MCB_Debug**
- **MCB_DTsoundPitchController**
- **MCB_EngineSFXController**
- **MCB_FloatIncrementer**
- **MCB_FoleySound**
 - **MCB_ContactSFX**
 - **MCB_ImpactSFX**
 - **MCB_RollingSFX**
 - **MCB_SkiddingSFX**
 - **MCB_EngineSFX**
- **MCB_FootSimple**
- **MCB_FootSimpleMover**
- **MCB_ImplicitPhysicsObjectsCreator**
 - **MCB_ImplicitPhysicsObjectsCreatorCarTyre**
 - **MCB_ImplicitPhysicsObjectsCreatorCarTyreDamper**
 - **MCB_ImplicitPhysicsObjectsCreatorCarTyreJoint**
- **MCB_IntervalSwitcher**
- **MCB_LegController**
 - **MCB_6LegController**
- **MCB_Light**
 - **MCB_LightBlinking**
 - **MCB_LightBlinkingDelayed**
 - **MCB_LightDelayed**
- **MCB_LightSwitch**
 - **MCB_LightSwitchDelayed**
- **MCB_MassGravity**
- **MCB_PhysicalBody**
 - **MCB_Car**
 - **MCB_CarTyre**
 - **MCB_CarTyreDamper**
 - **MCB_CarTyreJoint**
 - **MCB_DebugTorque**
- **MCB_Physics**
- **MCB_PowerReactor**
- **MCB_PowerReactorFX**

⁷¹ Die Bezeichnung dieser Klasse wurde in Text als „MCB_Avoider2D“ abgekürzt.

- **MCB_PSimObjectDeactivator**
- **MCB_PSimRollingDragForce**
- **MCB_RobotHead**
- **MCB_RobotHeadMover**
- **MCB_RobotMainBodyMainBehaviour**
- **MCB_Rotor**
- **MCB_RotorMover**
- **MCB_Runway**
 - **MCB_KITTScanner**
- **MCB_RunwayLights**
 - **MCB_KITTScannerLights**
- **MCB_SimpleFollower**
- **MCB_SimpleFollowerMover**
- **MCB_SimpleMotion**
- **MCB_SimpleMotionMover**
- **MCB_SoundAverager**
- **MCB_Vector**
- **MCB_VectorDataReader**
- **MCB_VectorSetter**
- **MCB_Walk**
 - **MCB_6LegWalk**
- **MCB_WalkBodyMovement**
- **MCBU_Physics**
- **MCBU_TICS_SceneController**
- **MC_Object**
 - **MCO_Rendering**
 - **MCO_Universe**
 - **MCU_Physics**
 - **MCU_TICS_SceneController**
 - **MCU_Time**
 - **MCO_WithTransform**
 - **MCO_Dynamic**
 - **MCD_6LegRobotMainBody**
 - **MCD_CameraBasic**
 - **MCD_Camera**
 - **MCD_CameraPoint**
 - **MCD_CarBorder2D**
 - **MCD_DarkAreaMarker**
 - **MCD_Debug**
 - **MCD_FloatIncrementer**
 - **MCD_FoleySound**
 - **MCD_ContactSFX**
 - **MCD_ImpactSFX**
 - **MCD_RollingSFX**
 - **MCD_SkiddingSFX**
 - **MCD_EngineSFX**
 - **MCD_Foot**
 - **MCD_FootSimple**
 - **MCD_LaneMarker**
 - **MCD_Light**
 - **MCD_LightBlinking**


- **MCD_LightBlinkingDelayed**
 - **MCD_LightDelayed**
 - **MCD_PhysicalBody**
 - **MCD_Car**
 - **MCD_CarTyre**
 - **MCD_CarTyreDamper**
 - **MCD_CarTyreJoint**
 - **MCD_DebugTorque**
 - **MCD_PowerReactor**
 - **MCD_RobotHead**
 - **MCD_Rotor**
 - **MCD_Runway**
 - **MCD_KITTSScanner**
 - **MCD_SimpleFollower**
 - **MCD_SimpleMotion**
 - **MCD_StereoCameraMaster**
 - **MCD_TICS_Block**
 - **MCD_TICS_Block_Battery**
 - **MCD_TICS_Block_Capacitor**
 - **MCD_TICS_Block_Coil**
 - **MCD_TICS_Block_Connection**
 - **MCD_TICS_Block_CornerConnection**
 - **MCD_TICS_Block_CrossingConnection**
 - **MCD_TICS_Block_TConnection**
 - **MCD_TICS_Block_Light**
 - **MCD_TICS_Block_Resistor**
 - **MCD_TICS_Block_Switch**
 - **MCD_TICS_Block_Transistor**
 - **MCD_Vector**
 - **MCD_VectorHead**
 - **MCO_Generating**
- **MC_Property**
 - **MC_PropertyCommOptimized**
 - **MCP_Boolean**
 - **MCP_BooleanStandardProperty**
 - **MCP_Double**
 - **MCP_DeltaT**
 - **MCP_DoubleStandardProperty**
 - **MCP_PersistentDouble**
 - **MCP_StereoCameraWidth**
 - **MCP_Long**
 - **MCP_LongStandardProperty**
 - **MCP_PersistentLong**
 - **MCP_GeometryMessenger**
 - **MCP_SceneObjectNA**
 - **MCP_StringNA**
 - **MCP_MaterialDescriptorContainer**
 - **MCP_XYZ**
 - **MCP_Position**
 - **MCP_Rotation**

- MCP_Scale
- MC_Help
 - MCH_AnimationKey (implements java.lang.Cloneable)
 - MCH_BooleanOnOffAnimationKey
 - MCH_BooleanFloatAnimationKey
 - MCH_DoubleAnimationKey
 - MCH_LongAnimationKey
 - MCH_CollisionInfo
 - MCH_CollisionPair
 - MCH_EscapeVectorPair
 - MCH_Spring
 - MCH_ValueHistoryElement
 - MCH_BooleanHistoryElement
 - MCH_DoubleHistoryElement
 - MCH_TorqueHistoryElement
- java.lang.Throwable (implements java.io.Serializable)
 - java.lang.Exception
 - java.lang.RuntimeException
 - MC_Exception
 - MCE_BehaviourAndPropertyContainerFieldsMustBeDeclaredWithTheExactTypeOfTheObjectTheyAreInitializedWith
 - MCE_BehaviourAndPropertyContainerFieldsMustBeFinal
 - MCE_BehaviourAndPropertyContainerFieldsMustNotBeInitializedWithNull
 - MCE_BehaviourOwningCycle
 - MCE_ContainerFieldOfMainBehaviourMustBeNamed_mainBehaviour
 - MCE_InheritedBehavioursAndPropertiesMayOnlyBeOverwrittenByObjectsOfTheSameTypeOrByObjectsOfASubtype
 - MCE_IsNotParentException
 - MCE_MC_BehaviourNotFoundException
 - MCE_MC_PropertyNotFoundException
 - MCE_NamesOfStandardSPsMustBeUnique
 - MCE_NamesOfStandardSPsMustNotContainIllegalCharacters
 - MCE_NamesOfStandardSPsMustNotEqualMaxScriptKeyword
 - MCE_NoAppropriateParentFoundException
 - MCE_NonAnimatablePropertyValuesMustNotBeChangedOutsideOfInitMethodsOfRootOwner
 - MCE_ObjectTypesMustNotDirectlyOwnMoreThanOneBehaviour
 - MCE_StandardPropertiesMustBeTheFirstPropertiesDefinedForAnObject
 - MCE_StandardPropertyMustDirectlyBeOwnedByObject
- javax.media.j3d.Transform3D
 - MCH_Transform3D
- javax.vecmath.Tuple2d (implements java.lang.Cloneable, java.io.Serializable)
 - javax.vecmath.Vector2d (implements java.io.Serializable)
 - MCH_Vector2D
- javax.vecmath.Tuple3d (implements java.lang.Cloneable, java.io.Serializable)
 - javax.vecmath.Vector3d (implements java.io.Serializable)
 - MCH_Vector3D

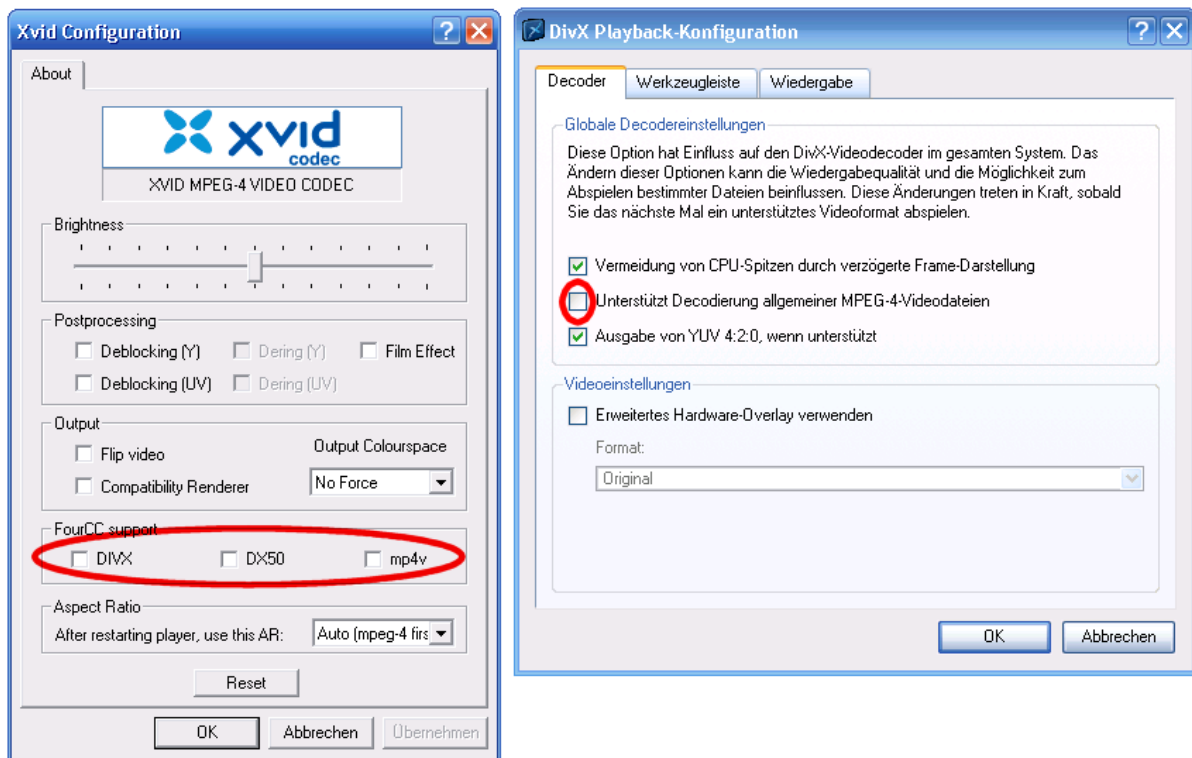
16 Anlagen

Diesem Dokument liegt eine DVD mit folgendem Inhalt bei:

Im Hauptverzeichnis befindet sich die Datei „Doktorarbeit_Jan_Paul.pdf“, welche diesen Text im PDF-Format enthält.

Das Unterverzeichnis „Movies“ enthält die Filme, welche in diesem Text über das Symbol  referenziert werden. Dabei befinden sich in „Movies\CarRace“ die in Kapitel 10.4.1.5 referenzierten Filme.

Die Filme verwenden die Codecs „DivX⁷²“ und „Xvid⁷³“. Diese Codecs sind unter den als Fußnoten angegebenen Internet-Adressen zu finden. Unter Windows-Betriebssystemen sollten die Decoder beider Codecs wie nachfolgend angegeben so konfiguriert werden, dass sie nur Filme im eigenen und nicht im jeweils anderen Codec wiedergeben. Dabei sind die rot elliptisch umrahmten Einstellungen relevant.



Abbildungen 103: Codec-Einstellungen

⁷² <http://www.divx.com/>, 18.07.2007

⁷³ <http://www.xvid.org/>, 18.07.2007