

Kollisionserkennung und -reaktion physikalischer Objekte in virtuellen Umgebungen

Diplomarbeit, vorgelegt von Bernd Kurtenbach

Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik III

3. Februar 1998

Zusammenfassung

Objekten virtueller Welten, die in ihrer geometrischen Beschreibung gegeben sind und die deshalb mit den Mitteln der Computergraphik dargestellt werden können, werden physikalische Eigenschaften im Sinne der newtonschen Mechanik hinzugefügt.

Betrachtet werden nur Aspekte der Kinematik und Dynamik, die zunächst in anschaulicher Form zusammengestellt werden. Statische Aspekte werden nicht betrachtet.

Als besonders aufwendig erweist sich die Realisierung von Kollisionen. Es wird sowohl auf Kollisionserkennung (collision detection) als auch auf Reaktionen auf Kollisionen (collision response) eingegangen.

Im Rahmen des MRT (Minimal Rendering Tool, entwickelt von der Computergraphikgruppe der Abteilung III des Instituts für Informatik der Universität Bonn) werden physikalische Objekte mit plausiblen Kollisionseigenschaften realisiert und es wird ein Kollisionserkennungsalgorithmus implementiert, der durch den Einsatz von Raumunterteilungsverfahren beschleunigt wird.

Da das MRT in der objektorientierten Sprache C++ implementiert ist, wird ein Multiple-Dispatching-Verfahren zur Auswahl der Kollisionsroutinen zweier Objekte eingesetzt. Außerdem sind Erweiterungsmöglichkeiten, wie sie für das MRT typisch sind, auch für die physikalisch basierten Konzepte vorgesehen.

Das Programm *mrtphys* wird vorgestellt, das zum einen als Beispielimplementation im Rahmen des MRT, zum anderen aber auch zur Erprobung der physikalisch basierten Konzepte dient.

Inhaltsverzeichnis

1	Einleitung	4
2	Physikalische Objekte in der Computergraphik	5
2.1	Beweglichkeit	5
2.2	Undurchdringlichkeit	5
2.3	Effizienz der Berechnung	5
3	Physikalische Grundlagen	6
3.1	Ruhendes Objekt	6
3.2	Gleichförmige Bewegung	6
3.3	Gleichförmig beschleunigte Bewegung	6
3.4	Krafteinwirkung	7
3.5	Arbeit und Energie	7
3.6	Impuls	7
3.6.1	Lineare kinetische Energie	8
3.7	Drehimpuls	8
3.7.1	Winkelgeschwindigkeit	8
3.7.2	Trägheitsmoment	9
3.7.3	Rotationsenergie	10
3.7.4	Drehmoment	10
3.8	Stoß	11
3.8.1	Impulsübertrag	11
3.8.2	Reibungsfreier, elastischer Stoß	12
3.8.3	Zentraler, elastischer Stoß	14
3.8.4	Reibung	15
3.8.5	Stoßobjekte mit unendlich großer Masse	15
4	Partikelsystem	17
4.1	Strömungsfelder	17
4.2	Kraftfelder	17
5	Kollisionserkennung	18
5.1	Problematik	18
5.1.1	Berechnungsaufwand	18
5.1.2	Programmieraufwand	18
5.1.3	Diskretisierung	18
5.1.4	Kollisionsreihenfolge	19
5.2	Kollisionserkennungsalgorithmen	19
5.2.1	Algorithmus I	19
5.2.2	Algorithmus II	20
5.2.3	Algorithmus III	21
5.3	Beschleunigung	22
5.3.1	Hüllkörper	22
5.3.2	Reguläres Gitter	22
5.3.3	Adaptive Raumunterteilung	23
5.3.4	Quantitativer Vergleich	24

6	Implementierung physikalischer Objekte im MRT	28
6.1	t_VelocityField	28
6.2	t_ForceField	28
6.3	t_PhysicalObject	28
6.3.1	physicalFunction	29
6.3.2	physicalFrame	30
6.3.3	Kollisionsrelevante Daten	30
6.3.4	boundingSphere	30
6.3.5	translate und transform	30
6.3.6	genericCollisionDetect	30
6.3.7	genericCollisionResponse	31
6.4	t_CollisionDetection	31
6.4.1	t_CDRegularGrid	31
6.4.2	t_CDAdaptiveGrid	31
6.5	Multiple Dispatching	33
6.5.1	t_CollisionDispatching	34
6.6	Erzeugung physikalischer Objekte	36
6.6.1	t_RigidSphere	36
6.6.2	t_RigidPolyhedron	37
6.6.3	t_RigidQuadrangle	38
6.7	Begrenzungsobjekte	38
6.7.1	t_LimitPlane	38
6.7.2	t_LimitHalfSpace	39
6.7.3	t_LimitBox/t_LimitQuadrangle	39
7	mrtphys	40
7.1	Interaktive Steuerung der Animation	41
7.2	Aufbau des Programms	41
7.2.1	penguin-basierte Dialogführung	42
7.2.2	t_SceneWindow	42
7.2.3	t_PhysicalAnimation	42
8	Ausblick	43
A	Klassenreferenz	44
A.1	t_BSphere	44
A.2	t_CDAdaptiveGrid	44
A.3	t_CDRegularGrid	46
A.4	t_CollisionDetection	47
A.5	t_CollisionDispatch	48
A.6	t_EarthGravity	49
A.7	t_ForceField	50
A.8	t_Invisible	50
A.9	t_LimitBox	50
A.10	t_LimitHalfSpace	51
A.11	t_LimitPlane	51
A.12	t_LimitQuadrangle	53
A.13	t_PhysicalObject	53
A.14	t_RigidPolyhedron	57
A.15	t_RPRep	58

A.16	t_RigidQuadrangle	59
A.17	t_RigidSphere	60
A.18	t_RigidTetrahedron	61
A.19	t_Velocityfield	62
B	<i>mrtphys</i>-Klassenreferenz	63
B.1	t_AnimationDialog	63
B.2	t_ForcesDialog	63
B.3	t_PhysicalAnimation	64
B.4	t_RenderDialog	65
B.5	t_SceneWindow	65

1 Einleitung

Wenn in der Computergraphik physikalische Zusammenhänge berücksichtigt werden sollen, dann hat es Tradition (etwa bei den Beleuchtungsmodellen), daß man zwischen realistischer Modellierung und effizienter Darstellung abwägt.

In dieser Arbeit wird Wert darauf gelegt, daß im 3. Kapitel (Physikalische Grundlagen) nicht nur die Basis für die Erzeugung realistisch wirkender Animationen gelegt wird, sondern auch, daß sauber abgegrenzt wird, welche natürlichen Beobachtungen in der Modellierung keine Berücksichtigung finden. Diese Grenze soll nicht willkürlich erscheinen, sondern es soll ein abgerundetes Konzept entstehen, dessen Ziele in Kapitel 2 abgesteckt sind.

Auch die im 4. Kapitel eingeführten Konzepte eines Partikelsystems sollen die resultierenden Animationen plausibler erscheinen lassen. Es sei aber noch einmal ausdrücklich darauf hingewiesen, daß hier nicht der Schwerpunkt dieser Arbeit zu finden ist.

Ab dem 5. Kapitel geht es um die für den naiven Betrachter selbstverständlich erscheinende Tatsache, daß sich Dinge nicht durcheinander hindurch bewegen, sondern, daß sie voneinander abprallen. Dies zu realisieren, erweist sich für die Computergraphik als aufwendiger, als man es zunächst annehmen könnte. Aufgrund der Komplexität des Problems ($O(n^2)$) werden deshalb neben einer intuitiven Lösung auch Beschleunigungsverfahren betrachtet.

Diese Diplomarbeit hat neben dem theoretischen auch einen praktischen Teil, mit dem sich die Kapitel 6 und 7 befassen. Auf ihn entfällt der weitaus größte Teil des Zeitaufwandes für die Erstellung dieser Arbeit, was jedoch absolut notwendig ist, da nur die praktische Anwendung den angestellten Überlegungen Sinn gibt.

In Kapitel 6 wird beschrieben, wie die gewünschten Konzepte in das MRT integriert wurden. Unter anderem wird ein Multiple-Dispatching-Verfahren zur Auswahl einer geeigneten Kollisionsroutine für zwei Objekte vorgestellt.

Kapitel 7 stellt mit *mrtphys* ein Programm vor, daß sowohl als Beispielimplementation, als auch zur Erprobung der erarbeiteten Objekte dient.

2 Physikalische Objekte in der Computergraphik

In ihrem Bestreben, möglichst realistisch wirkende Bilder zu erzeugen, hat die Computergraphik eine Vielzahl von Techniken entwickelt. Betrachtet man die existierenden Verfahren zur Beleuchtung, so kann man eine Entwicklung beobachten, die vom Flat-Shading mit einfachem Lambert-Reflektierer über Phong-Shading und -Beleuchtung bis hin zu globalen Beleuchtungsmodellen (Raytracing, Radiosity) immer mehr physikalische Gesetzmäßigkeiten berücksichtigt [Fel92] [Gla95]. Dabei mußten immer Zugeständnisse an die verfügbare Anzeigetechnik gemacht werden. Insbesondere mußte die Berechnungszeit der Bilder zumutbar bleiben.

Bei der Realisierung physikalischer Objekte wird analog vorgegangen. Eine Reihe physikalischer Eigenschaften, die die Objekte haben sollen, werden definiert, ohne jedoch zu behaupten, daß die resultierenden Animationen der Realität in jeder Hinsicht entsprechen.

2.1 Beweglichkeit

Physikalische Objekte sollen sich frei im Raum bewegen können. Sie sollen dabei auf einwirkende Kräfte im Sinne der newtonschen Mechanik reagieren, d.h. entsprechend ihrer Massenträgheit beschleunigt werden.

Ziel 1 *Abgesehen von Diskretisierungsfehlern sollen sich die Objekte im Sinne newtonscher Kinematik bewegen und sie sollen im Sinne newtonscher Dynamik auf Kräfte reagieren.*

2.2 Undurchdringlichkeit

Wie in der Realität sollen sich Objekte, die aufeinander treffen, nicht gegenseitig durchdringen, sondern sie sollen voneinander abprallen.

Ziel 2 *Unter der Einschränkung, daß alle Bewegungen während eines Zeitintervalls als linear angenommen werden, sollen alle Kollisionen in der Szene erkannt werden.*

Kollisionen, die aufgrund der Einschränkung nicht erkannt werden, verlaufen annähernd parallel. Sie haben deshalb wenig Einfluß auf die Szene. Auf dieses Problem wird in Kapitel 5 näher eingegangen.

Ziel 3 *Die Objekte sollen auf Kollisionen im Sinne des vollkommen elastischen Stoßes reagieren.*

Insbesondere heißt das, daß sich die Objekte nicht durch den Stoß verformen. Auf den vollkommen elastischen Stoß wird in 3.8 eingegangen.

2.3 Effizienz der Berechnung

Ziel 4 *Mit heute verfügbarer 3D-Graphik-Hardware, die die Darstellung von Polygon-Szenen signifikant beschleunigt, sollen Echtzeitanimationen auch komplexerer Szenen möglich sein.*

Es soll also nicht der Berechnungsaufwand von Einzelbildern in Sekunden oder gar Minuten betrachtet werden müssen, sondern Einzelbildraten (frames per second) sollen erreicht werden, die beim Betrachter den Eindruck einer physikalisch plausiblen Animation erwecken.

Selbstverständlich stößt man hier auf Grenzen. Quantitative Betrachtungen werden in 5.3.4 angestellt. Sie zeigen, wie „komplex“ Szenen vorläufig sein dürfen.

3 Physikalische Grundlagen

Im folgenden werden die Grundlagen newtonscher Mechanik in dem Umfang aufbereitet, der für die Realisierung der in Kapitel 2 formulierten Ziele notwendig ist. Dabei werden einige vereinfachende Annahmen gemacht.

Die verwendete Notation richtet sich nach [GV93]. Sie ist in der Physik üblich. Fettgedruckte Symbole bezeichnen Vektoren, Punkte über den Symbolen Ableitungen nach der Zeit t . Der Betrag des Vektors a wird mit $a = |a|$ bezeichnet.

3.1 Ruhendes Objekt

Das ruhende Objekt ist durch seine Ausdehnung und seine Masse beschrieben. Die Masse m eines Objekts wird in Kilogramm kg angegeben. Es existiert ein Schwerpunkt S . Der Massenerhaltungssatz lautet:

In einem geschlossenen System bleibt die Masse erhalten.

Ein geschlossenes System ist hierbei, wie bei den noch folgenden Erhaltungssätzen, eine Menge von Massepunkten, auf die keine äußeren Kräfte einwirken.

Annahme 1 Mit Objekt werden im folgenden unverformbare (starre) Körper bezeichnet. Jedes für sich bildet ein geschlossenes System. Kommt es zu einer Kollision, so bilden die beiden betroffenen Objekte ein geschlossenes System.

Annahme 2 Die Dichte (Masse pro Volumen $\frac{\text{kg}}{\text{m}^3}$) innerhalb der Objekte ist homogen, d.h. in jedem Volumenelement des Objekts gleich. Der Schwerpunkt fällt deshalb mit dem geometrischen Mittelpunkt zusammen.

3.2 Gleichförmige Bewegung

Ein Objekt bewegt sich mit Geschwindigkeit v . Sie wird in Metern pro Sekunde $\frac{\text{m}}{\text{s}}$ gemessen.

Für den zurückgelegten Weg r (gemessen in Metern m) nach einem Zeitintervall der Länge t (gemessen in Sekunden s) gilt:

$$r = v \cdot t \quad (1)$$

$$\Rightarrow v = \dot{r} \quad (2)$$

3.3 Gleichförmig beschleunigte Bewegung

Die Beschleunigung a eines Objekts wird in $\frac{\text{m}}{\text{s}^2}$ gemessen. Es gelten

$$v = a \cdot t \quad (3)$$

$$r = \frac{1}{2} a \cdot t^2 \quad (4)$$

$$\Rightarrow a = \dot{v} = \ddot{r} \quad (5)$$

Bei Anfangsgeschwindigkeit v_0 gilt wegen (1)

$$r = v_0 \cdot t + \frac{1}{2} a \cdot t^2 \quad (6)$$

3.4 Krafteinwirkung

Die Kraft F , die auf ein Objekt einwirkt, wird in Newton $N = \frac{\text{kg} \cdot \text{m}}{\text{s}^2}$ gemessen. Das Objekt beschleunigt gemäß dem „Aktionsprinzip“:

$$F = m \cdot a \quad (7)$$

3.5 Arbeit und Energie

Wirkt eine beschleunigende Kraft längs eines Weges auf ein Objekt, so spricht man von Arbeit W , gemessen in Joule $J = \text{Nm}$. Arbeit ist eine ungerichtete Größe. Für Kraft F und Weg r gilt die „goldene Regel der Mechanik“:¹

$$W = F \cdot r \quad (8)$$

Arbeit kann auch in anderen Formen verrichtet werden, z.B. durch Hochheben oder Erhitzen. Die an einem Objekt verrichtete Arbeit wird in Form von Energie gespeichert. Diese entspricht der verrichteten Arbeit und wird somit ebenfalls in Joule gemessen.

Es gilt der Energieerhaltungssatz:

$$\textit{In einem geschlossenen System bleibt die Energie erhalten.} \quad (9)$$

Da sich Energie in verschiedene Formen umwandeln kann, gilt der Energieerhaltungssatz im Allgemeinen nicht, wenn man ihn auf spezielle Energieformen einschränkt.

Die Energie bewegter Massepunkte heißt kinetische Energie. Sie ist die einzige in dieser Arbeit betrachtete Energieform. Sie kann sowohl in linearer Form W_{lin} , als auch als Rotationsenergie W_{rot} vorkommen. Für die Gesamtenergie W eines Objektes gilt somit im Rahmen dieser Arbeit:

$$W = W_{lin} + W_{rot} \quad (10)$$

3.6 Impuls

Der Impuls p ist das Produkt aus Masse und Geschwindigkeit eines Objekts:

$$p = m \cdot v \quad (11)$$

Wegen (7) und (3) gilt als Zusammenhang zwischen Impuls und Kraft

$$F = m \cdot \frac{v}{t}$$

$$\Leftrightarrow p = F \cdot t \quad (12)$$

$$\Rightarrow F = \dot{p} \quad (13)$$

Als Einheit für den Impuls ergibt sich $\frac{\text{kg} \cdot \text{m}}{\text{s}}$ das entspricht Newtonsekunden Ns . Es gilt der Impulserhaltungssatz:

$$\textit{In einem geschlossenen System bleibt der Impuls erhalten.} \quad (14)$$

Der Gesamtimpuls ist hierbei als Summe der Einzelimpulse im System definiert. Da der Impuls eine gerichtete Größe ist, können sich einzelne Impulse also gegenseitig aufheben.

Bei geschlossenen Systemen mehrerer Objekte bewegt sich der gemeinsame Masseschwerpunkt gleichförmig geradlinig mit Geschwindigkeit v_S , egal welche Impulse zwischen den einzelnen Objekten ausgetauscht werden. Für ein System mit zwei Objekten gilt:

$$v_S = \frac{p_1 + p_2}{m_1 + m_2} \quad (15)$$

¹ $a \cdot b$ bezeichnet das Skalarprodukt. Es gelten: $a \cdot b = \cos(a, b) \cdot |a| \cdot |b|$ und $a^2 = |a|^2$

3.6.1 Lineare kinetische Energie

Ein Objekt mit Impuls $p = m \cdot v$ hat die lineare kinetische Energie W_{lin} , die der Arbeit entspricht, die zur Beschleunigung des Objektes nötig war. Es gilt wegen (3), (4), (7), (8) und (11):

$$\begin{aligned} W_{lin} &= \mathbf{F} \cdot \mathbf{r} \\ &= m \cdot a \cdot r \\ &= m \cdot a \cdot \frac{1}{2} a \cdot t^2 \\ &= m \cdot \frac{v}{t} \cdot \frac{1}{2} \frac{v}{t} \cdot t^2 \\ &= \frac{1}{2} m \cdot v^2 \end{aligned} \tag{16}$$

$$= \frac{p^2}{2m} \tag{17}$$

3.7 Drehimpuls

Im Gegensatz zum linearen Impuls eines Objekts liegt ein Drehimpuls $\mathbf{L} \neq 0$ vor, wenn sich das Objekt um eine Rotationsachse dreht. Da wegen Annahme 1 keine äußeren Kräfte auf die Objekte einwirken, verläuft diese immer durch den Schwerpunkt S . Es gilt:

$$\mathbf{L} = J \cdot \boldsymbol{\omega} \tag{18}$$

Hierbei ist das Trägheitsmoment J (Einheit $\text{kg} \cdot \text{m}^2$) das Analogon zur Masse bei der Berechnung des linearen Impulses, und die Winkelgeschwindigkeit $\boldsymbol{\omega}$, gemessen in Radiant pro Sekunde $\frac{1}{\text{s}}$, ist analog zur linearen Geschwindigkeit zu sehen. Für den Drehimpuls ergibt sich somit die Einheit $\frac{\text{kg} \cdot \text{m}^2}{\text{s}}$. Das entspricht Joulesekunden Js.

Analog zum Impulserhaltungssatz gilt der Drehimpulserhaltungssatz:

$$\text{In einem geschlossenen System bleibt der Drehimpuls erhalten.} \tag{19}$$

Der Gesamtdrehimpuls eines Systems, in dem sich Objekte sowohl linear als auch drehend bewegen, besteht jedoch nicht nur aus der Summe der rotationsbedingten Drehimpulse. Auch mit linearer Bewegung ist ein Drehimpuls verbunden.

Sei P ein fester Bezugspunkt im Raum und S der Schwerpunkt eines Objektes mit Masse m , das sich mit Geschwindigkeit \mathbf{v} bewegt. Dann gilt mit dem Vektor $\mathbf{r} = \overrightarrow{S \leftrightarrow P}$, daß das Objekt den Gesamtdrehimpuls \mathbf{L}_P bezüglich P wie folgt hat:

$$\mathbf{L}_P = m \cdot (\mathbf{r} \times \mathbf{v}) \tag{20}$$

Es gilt, daß sich dieser Drehimpuls nicht ändert, wenn das Objekt sich gleichförmig geradlinig an P vorbeibewegt, da die Änderungen von Distanz und Winkel sich gegenseitig aufheben.

\mathbf{L}_P ist jedoch von der Wahl von P abhängig. Ein Objekt, das sich auf P zubewegt hat keinen Drehimpuls bezüglich P . Ein Objekt, das sich senkrecht zu \mathbf{r} bewegt, kann auch bei kleiner Geschwindigkeit einen sehr großen Drehimpuls bezüglich P haben, wenn es weit von P entfernt ist.

3.7.1 Winkelgeschwindigkeit

Die Winkelgeschwindigkeit $\boldsymbol{\omega}$ zeigt in die Richtung der Rotationsachse. Der Winkelgeschwindigkeitsbetrag wird mit $\omega = |\boldsymbol{\omega}|$ bezeichnet. Für den Rotationsvektor \mathbf{R} gilt:

$$\mathbf{R} = \frac{\boldsymbol{\omega}}{\omega} \tag{21}$$

Seine positive Drehrichtung ist gegen den Uhrzeigersinn, wenn er zum Betrachter hin zeigt.

Ein Punkt P eines Objektes mit Schwerpunkt S , das sich mit Geschwindigkeit v und Winkelgeschwindigkeit w bewegt, hat mit $r = P \leftrightarrow S$ die lineare Geschwindigkeit:

$$v_P = v + (w \times r) \quad (22)$$

3.7.2 Trägheitsmoment

Die Bestimmung des Trägheitsmoments starrer, rotierender Körper J ist nicht trivial, da die Entfernung r jedes Massepunktes zur Drehachse berücksichtigt werden muß:

$$J = \int_m r^2 dm \quad (23)$$

Das zu lösende Integral ist also von der Objektform und der Drehachse abhängig. Das Problem läßt sich etwas vereinfachen, indem die Rotation um eine beliebige Achse in Rotationen um drei Achsen aufgeteilt wird.

Körper neigen dazu, ihre Rotation auf bestimmte Hauptachsen zu verlagern. Symmetrieachsen sind immer solche Hauptachsen. In der Literatur [GV93, Seite 68] sind Trägheitsmomente für einige Objekte und ihre Hauptträgheitsachsen zu finden. Hierbei wird, wie in Annahme 2, von homogener Dichteverteilung innerhalb der Objekte ausgegangen.

Besonders einfach verhält es sich für die Kugel mit Radius r . Hier ist jede Achse durch den Mittelpunkt eine Hauptträgheitsachse und es gilt:

$$J = m \cdot \frac{2}{5} r^2 \quad (24)$$

Beim Würfel mit Kantenlänge l sind die drei Achsen durch den Mittelpunkt, die die Außenflächen mittig durchstoßen Hauptträgheitsachsen. Für jede gilt:

$$J = m \cdot \frac{1}{6} l^2 \quad (25)$$

Bei einem Zylinder mit Radius r und Höhe h gilt für die Symmetrieachse, die die Mittelpunkte der beiden Kreisflächen schneidet:

$$J = m \cdot \frac{1}{2} r^2 \quad (26)$$

Die beiden anderen Hauptträgheitsachsen können beliebig gewählt werden, so daß ein Orthogonalsystem mit Ursprung im Mittelpunkt entsteht. Für diese gilt dann:

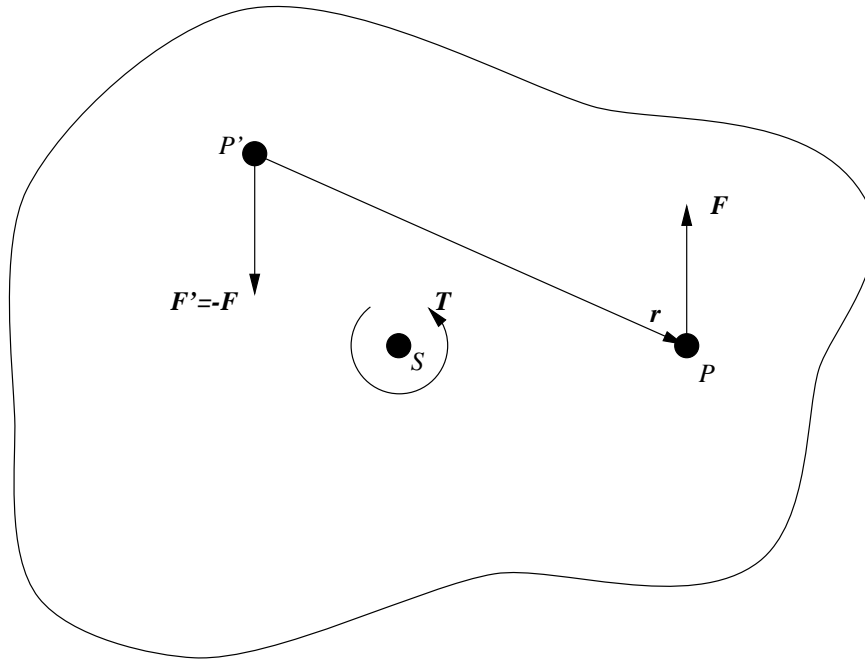
$$J = m \cdot \left(\frac{1}{4} r^2 + \frac{1}{12} l^2 \right) \quad (27)$$

Unabhängig von der Rotationsachse läßt sich das Trägheitsmoment als 3×3 Tensor $\mathbf{J} = J_{i,k}$ darstellen. Dabei entspricht jede Zeile einer Achse des lokalen Bezugssystems. Wählt man als Bezugssystem drei Hauptträgheitsachsen, so hat der Tensor Diagonalfom. Die Trägheitsmomente J , die sich aus den obigen Beispielen ergeben, stehen dann in der Zeile auf der Diagonalen, die ihrer Rotationsachse entspricht.

Annahme 3 Im Rahmen dieser Arbeit wird davon ausgegangen, daß alle Objekte das Trägheitsmomentverhalten einer Kugel haben. Es gilt also gemäß (24):

$$J_{\mathbf{R}} = m \cdot \frac{2}{5} r^2 \text{ für alle Rotationsvektoren } \mathbf{R} \quad (28)$$

Der „Radius“ r ist sinnvoll zu wählen.

Abbildung 1: Drehmoment T aufgefaßt als Kräftepaar (F, F')

3.7.3 Rotationsenergie

Die Rotationsenergie W_{rot} , die mit einem rotationsbedingten Drehimpuls verbunden ist, berechnet sich analog zur linearen kinetischen Energie:

$$W_{rot} = \frac{1}{2} J \cdot \omega^2 = \frac{L^2}{2J} \quad (29)$$

3.7.4 Drehmoment

Auch die „Kraft“, die auf ein Objekt einwirken muß, um es in Rotation zu versetzen, ist ein Analogon zum linearen Fall. Sie heißt Drehmoment T und es gilt analog zum Aktionsprinzip:

$$T = J \cdot \dot{\omega} \quad (30)$$

Für die Winkelbeschleunigung $\dot{\omega}$ gilt ebenfalls analog:

$$\dot{\omega} = \frac{w}{t} \quad (31)$$

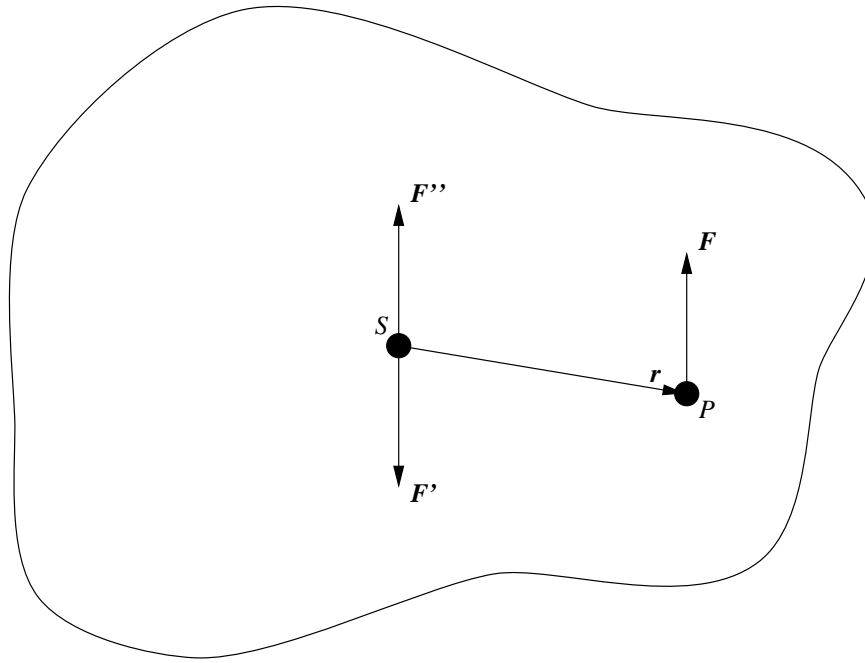
Somit folgt erwartungsgemäß:

$$L = T \cdot t \quad (32)$$

$$\Rightarrow T = \dot{L} \quad (33)$$

Als Drehmoment kann auch das lineare Kräftepaar $F = \Leftrightarrow F'$ angesehen werden (Abbildung 1). F wirkt hierbei auf einen Punkt P des Objekts, während F' genau entgegengesetzt auf Punkt P' wirkt. Mit $r = P \Leftrightarrow P'$ gilt:

$$T = r \times F \quad (34)$$

Abbildung 2: Nichtzentral einwirkende Kraft F

3.8 Stoß

Bei der Kollision zweier Objekte mit Index i und j ($i \neq j \in \{1, 2\}$) handelt es sich um ein geschlossenes System im Sinne des Energieerhaltungssatzes (9) und des Impulserhaltungssatzes (14).

Der Energieerhaltungssatz gilt jedoch im Allgemeinen nicht, wenn er auf die lineare kinetische Energie oder die Rotationsenergie der Objekte eingeschränkt wird. Es kommt nicht nur zu Übergängen zwischen beiden Energieformen. Es kann auch, durch Verrichtung von Verformungsarbeit, Verformungsenergie entstehen, die wiederum in andere Energieformen übergehen kann.

Aufgrund von Annahme 1 erfolgt der Stoß zweier Objekte mit Index jedoch völlig hart. Das heißt, er hat die Dauer $t = 0$ und der Energieerhaltungssatz gilt eingeschränkt auf kinetische Energie.

3.8.1 Impulsübertrag

In [Ham78] wird der Begriff „Stoßkraft“ S_v definiert. Gemeint ist, daß während eines harten Stoßes eine sehr große Kraft (genannt $K_v \rightarrow \infty$) für die Dauer eines sehr kleinen Zeitintervalls ($\tau \rightarrow 0$) auf ein Objekt wirkt:

$$\int_t^{t+\tau} K_v dt \rightarrow S_v$$

Da Kraft und Impulsänderung jedoch Synonyme sind (13), kann auf den Begriff „Stoßkraft“ verzichtet werden. Wirkt S_v auf ein sich frei bewegendes Objekt ein, so kommt es zu einem Impulsübertrag Δp , für den gilt:

$$S_v = \Delta p = p' \Leftrightarrow p \quad (35)$$

Hierbei ist p der Impuls vor und p' der Impuls nach der „Stoßkrafteinwirkung“.

Wie Abbildung 2 zeigt, wirkt eine Kraft F , die auf Punkt P eines Objektes wirkt sowohl als Kräftepaar (F, F') , wobei F' auf den Schwerpunkt S wirkt, als auch als linear beschleunigende Kraft $F'' = F$, die Gegenkraft von F' ist.

Betrachtet man statt der Kraft \mathbf{F} eine „Stoßkraft“ und somit den Impulsübertrag $\Delta\mathbf{p}$, so gilt für die ebenfalls mit dem Stoß verbundene Drehimpulsdifferenz $\Delta\mathbf{L}$:

$$\Delta\mathbf{L} = \mathbf{r} \times \Delta\mathbf{p} \quad (36)$$

Stößt ein rotierendes Objekt auf ein festes Hindernis an Punkt P , so entsteht ein Kräftepaar analog Abbildung 2. An P wirkt also eine Kraft \mathbf{F} und an S eine Kraft $\mathbf{F}' = \Leftrightarrow\mathbf{F}$ für die gilt:

$$\mathbf{T} = \mathbf{r} \times \mathbf{F} \quad (37)$$

$$\Rightarrow \mathbf{L} = \mathbf{r} \times \mathbf{p}_{P,rot}$$

$$\Leftrightarrow \mathbf{p}_{P,rot} = \frac{\mathbf{L} \times \mathbf{D}}{r^2} \quad (38)$$

Der rotationsbedingte Impuls $\mathbf{p}_{P,rot}$ des Punktes P , der mit \mathbf{L} um S rotiert, ist also umgekehrt proportional zum Abstand r .

Wegen (22) ergibt sich für seine Masse $m_{P,rot}$:

$$\begin{aligned} m_{P,rot} &= \frac{|\mathbf{p}_P|}{|v_P|} \\ &= \frac{|\mathbf{L} \times \mathbf{D}|}{r^2 \cdot |\mathbf{w} \times \mathbf{D}|} \\ &= \frac{|\mathbf{L}| \cdot r}{r^2 \cdot |\mathbf{w}| \cdot r} \\ &= \frac{J}{r^2} \end{aligned} \quad (39)$$

Während die Geschwindigkeit proportional zum Abstand wächst, nimmt die Masse quadratisch ab.

Zum rotationsbedingten Impuls kommt noch der lineare Impuls des Objektes, so daß für den Impuls des Punktes P gilt:

$$\mathbf{p}_P = \mathbf{p} + \mathbf{p}_{P,rot} \quad (40)$$

3.8.2 Reibungsfreier, elastischer Stoß

Abbildung 3 soll zunächst einige Bezeichnungen klären. Die Objekte 1 und 2 mit den Schwerpunkten S_1 und S_2 stoßen am Kollisionspunkt P_{col} zusammen. Die $r_{col,i} = P_{col} \Leftrightarrow S_i$ heißen Kollisionsradien.

Im abgebildeten Beispiel ist Objekt 2 an P_{col} differenzierbar. Die Oberflächennormale \mathbf{N}_2 definiert deshalb die Kollisionsnormalen $\mathbf{N}_{col,1} = \Leftrightarrow\mathbf{N}_2$ und $\mathbf{N}_{col,2} = \mathbf{N}_2$. Diese stehen senkrecht auf der Kollisionsfläche A_{col} .²

Sind beide Objekte an P_{col} differenzierbar, so gilt $\mathbf{N}_1 = \Leftrightarrow\mathbf{N}_2$. Es kann auch der Fall auftreten, daß keines der Objekte an P_{col} differenzierbar ist. Dann sind die $\mathbf{N}_{col,i}$ so zu wählen, daß A_{col} den Winkel der Kollisionsradien halbiert.

Beim Stoß der Objekte kommt es zu den Impulsüberträgen $\Delta\mathbf{p}_i$, wobei wegen des Impulserhaltungssatzes gilt:

$$\Delta\mathbf{p}_1 = \Leftrightarrow\Delta\mathbf{p}_2 \quad (41)$$

Für die Drehimpulsdifferenzen $\Delta\mathbf{L}_i$ gilt gemäß (36) und wegen Annahme 3:

$$\Delta\mathbf{L}_i = \mathbf{r}_{col,i} \times \Delta\mathbf{p}_i \quad (42)$$

$$\Leftrightarrow \Delta\mathbf{w}_i = \frac{\mathbf{r}_{col,i} \times \Delta\mathbf{p}_i}{J_i} \quad (43)$$

²Normalenvektoren \mathbf{N} sind immer normiert, d.h. es gilt $|\mathbf{N}| = 1$

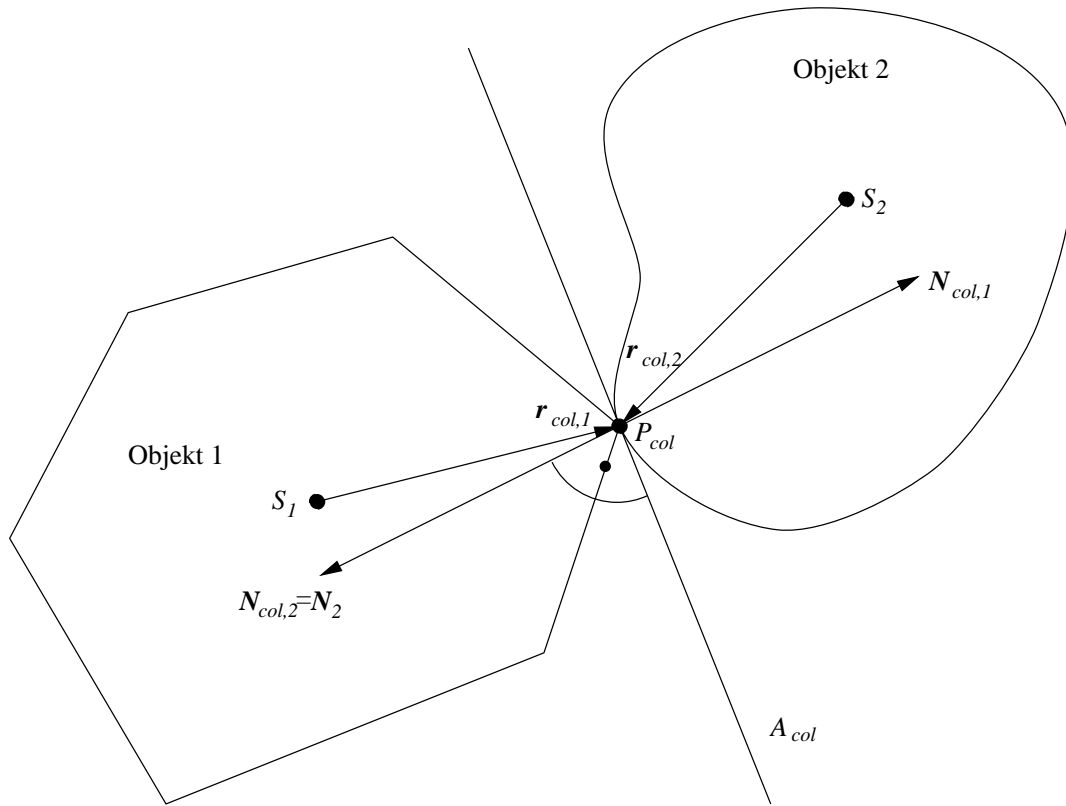


Abbildung 3: Einpunktiger Stoß zweier starrer Körper gegeneinander

Beim reibungsfreien Stoß erfolgt der Impulsübertrag längs der Kollisionsnormalen. Da somit mit den $N_{col,i}$ die Richtungen der Δp_i bereits bekannt sind, sind noch die Δp_i gesucht, so daß gilt:

$$\Delta p_i = \Delta p \cdot N_{col,i} \quad (44)$$

$$\Delta v_i = \frac{\Delta p \cdot N_{col,i}}{m_i} \quad (45)$$

$$\Delta w_i = \frac{\Delta p \cdot (r_{col,i} \times N_{col,i})}{J_i} \quad (46)$$

Für die Differenzen der linearen kinetischen Energie $\Delta W_{lin,i}$ gilt:

$$\begin{aligned} \Delta W_{lin,i} &= W'_{lin,i} \Leftrightarrow W_{lin,i} \\ &= \frac{(p_i + \Delta p_i)^2}{2m_i} \Leftrightarrow \frac{p_i^2}{2m_i} \\ &= \frac{p_i \Delta p_i}{m_i} + \frac{\Delta p_i^2}{2m_i} \\ &= m_i v_i \Delta v_i + \frac{1}{2} m_i \Delta v_i^2 \end{aligned} \quad (47)$$

Für die Rotationsenergieunterschiede $\Delta W_{rot,i}$ gilt analog:

$$\Delta W_{rot,i} = J_i w_i \Delta w_i + \frac{1}{2} J_i \Delta \omega_i^2 \quad (48)$$

Da der Stoß elastisch ist, bleibt die Gesamtenergie erhalten und es gilt:

$$\begin{aligned}
0 &= \Delta W \\
&= \Delta W_{lin,1} + \Delta W_{rot,1} + \Delta W_{lin,2} + \Delta W_{rot,2} \\
&= m_1 v_1 \Delta v_1 + m_2 v_2 \Delta v_2 + \frac{1}{2} m_1 \Delta v_1^2 + \frac{1}{2} m_2 \Delta v_2^2 + \\
&\quad J_1 w_1 \Delta w_1 + J_2 w_2 \Delta w_2 + \frac{1}{2} J_1 \Delta \omega_1^2 + \frac{1}{2} J_2 \Delta \omega_2^2
\end{aligned} \tag{49}$$

Durch Einsetzen von (45) und (46) ergibt sich:

$$\begin{aligned}
0 &= \Delta p v_1 N_{col,1} + \Delta p v_2 N_{col,2} + \\
&\quad \Delta p w_1 (r_{col,1} \times N_{col,1}) + \Delta p w_2 (r_{col,2} \times N_{col,2}) + \\
&\quad \frac{\Delta p^2}{2m_1} + \frac{\Delta p^2}{2m_2} + \frac{\Delta p^2 (r_{col,1} \times N_{col,1})^2}{2J_1} + \frac{\Delta p^2 (r_{col,2} \times N_{col,2})^2}{2J_2}
\end{aligned}$$

Wegen $\Delta p > 0$ kann einmal durch Δp dividiert werden. Außerdem läßt sich $N_{col,1}$ durch $N_{col} = \Leftrightarrow N_{col,2}$ ersetzen, so daß gilt:

$$\Delta p = 2 \frac{A}{B} \tag{50}$$

$$\begin{aligned}
A &= v_1 N_{col} \Leftrightarrow v_2 N_{col} + \\
&\quad w_1 (r_{col,1} \times N_{col}) \Leftrightarrow w_2 (r_{col,2} \times N_{col}) \\
&= (v_1 + (w_1 \times r_{col,1}) \Leftrightarrow (v_2 + (w_2 \times r_{col,2}))) \cdot N_{col} \\
&= (v_{P_{col,1}} \Leftrightarrow v_{P_{col,2}}) \cdot N_{col}
\end{aligned} \tag{51}$$

$$= v_{P_{col,1,2}} \cdot N_{col} \tag{52}$$

$$B = \frac{1}{m_1} + \frac{1}{m_2} + \frac{(r_{col,1} \times N_{col})^2}{J_1} + \frac{(r_{col,2} \times N_{col})^2}{J_2} \tag{53}$$

Die $v_{P_{col,i}}$ sind hierbei die Kollisionspunktgeschwindigkeiten gemäß (22). $v_{P_{col,1,2}}$ bezeichnet deren Differenz.

3.8.3 Zentraler, elastischer Stoß

Beim zentralen Stoß liegen die Massepunkte und der Kollisionspunkt auf einer Linie, die durch $r_{col,i}$ und N_{col} gegeben ist. Für diese gilt $r_i \times N_{col} = 0$, so daß es nicht zu Drehimpulsänderungen kommt.

Die Geschwindigkeiten v_i werden auf N_{col} projiziert. Die Bewegungen senkrecht zu N_{col} werden also vernachlässigt. Es gilt:

$$\Delta p = 2\mu(v_1 \Leftrightarrow v_2) N_{col} \tag{54}$$

Wobei μ auch „reduzierte Masse“ genannt wird. Es gilt:

$$\mu = \frac{m_1 \cdot m_2}{m_1 + m_2} \tag{55}$$

Stößt ein Objekt auf ein ruhendes Objekt gleicher Masse, so überträgt sich der gesamte Impuls.

3.8.4 Reibung

Durch Oberflächenreibung kann es auch zu Überträgen parallel zur Kollisionsfläche A_{col} kommen. Im reibungsfreien Fall wird der Anteil der Kollisionspunktgeschwindigkeitsdifferenz $v_{P_{col}}$ vernachlässigt, der senkrecht zu $N_{col,2}$ und somit parallel zu A_{col} ist. Es gilt:

$$v_{P_{col},1,2\perp} = (v_{P_{col},1,2} \cdot N_{col,2}) \cdot N_{col,2} \quad (56)$$

$$v_{P_{col},1,2\parallel} = v_{P_{col},1,2} \Leftrightarrow v_{P_{col},1,2\perp} \quad (57)$$

Annahme 4 Jede Oberfläche hat einen Oberflächenreibungskoeffizienten ϕ_i . Liegt senkrecht zur Kollisionsnormalen eine Kollisionspunktgeschwindigkeitsdifferenz $v_{P_{col},1,2\parallel} \neq 0$ vor, so wirkt diese gemäß

$$\phi v_{P_{col},1,2\parallel} = \phi_1 \cdot \phi_2 \cdot v_{P_{col},1,2\parallel} \quad (58)$$

Diese Annahme ist eine grobe Vereinfachung, denn Oberflächen haben im allgemeinen weder einen einheitlichen Koeffizienten für Reibung in beliebiger Richtung, noch für Reibung mit beliebigen anderen Oberflächen.

Durch dieses Modell ist es jedoch möglich, sehr rutschige (z.B. geölte) Oberflächen durch einen Wert nahe 0 und stark haftende Oberflächen (z.B. Sandpapier) durch einen Wert bei 1 zu betrachten.

Zur Realisierung dieser Heuristik muß die Kollisionsnormale N_{col} in (52) und (53) durch $N_{col\phi}$ ersetzt werden. Die Richtung des senkrechten Stoßanteils bleibt gleich. Die des parallelen Stoßanteils ergibt sich aus der Kollisionspunktimpulsdifferenz $p_{P_{col},1,2}$. Beide Anteile werden gemäß des Kollisionspunktimpulses und ϕ gewichtet:³

$$p_{P_{col},1,2} = p_{P_{col},1} \Leftrightarrow p_{P_{col},2} \quad (59)$$

$$p_{P_{col},1,2\perp} = |p_{P_{col},1,2} \cdot N_{col,2}| \cdot N_{col,2} \quad (60)$$

$$p_{P_{col},1,2\parallel} = p_{P_{col},1,2} \Leftrightarrow p_{P_{col},1,2\perp} \quad (61)$$

$$N_{col\phi} = \parallel p_{P_{col},1,2\perp} + \phi \cdot p_{P_{col},1,2\parallel} \parallel \quad (62)$$

3.8.5 Stoßobjekte mit unendlich großer Masse

Aufgrund der enorm großen Masse der Erde, kommt es bei Kollisionen mit Mauern oder anderen Objekten, die fest mit ihr verbunden sind, nicht zu einer meßbaren Impuls- oder Drehimpulsübertragung.

Es ist sinnvoll, für solche Objekte eine unendlich große Masse anzunehmen.

Da es hingegen keinen Sinn hat, Kollisionen zweier Objekte unendlich großer Masse zu betrachten, ist an einem solchen Stoß immer ein bewegliches und ein unbewegliches Objekt beteiligt. Im folgenden sei Objekt 1 beweglich und Objekt 2 habe unendliche Masse.

Vernachlässigt man Rotation und Oberflächenreibung, so wird der lineare Impuls des beweglichen Objektes längs der Kollisionsnormalen reflektiert. Man sagt: „Einfallswinkel gleich Ausfallswinkel“

Für Objekte unendlich großer Masse kann ein unendlich großer Impuls angenommen werden. Der Gesamtimpuls bleibt also erhalten, obwohl sich nur der Impuls des beweglichen Objektes ändert. Dasselbe gilt für den Drehimpuls.

³ $\|a\|$ bezeichnet den normierten Vektor von a . Es gilt: $\|a\| = \frac{a}{|a|}$

Unter Berücksichtigung der Oberflächenreibung gilt für den Impulsübertrag Δp_∞ in diesem Fall:

$$v_{P_{col}\perp\infty} = (v_{P_{col,1}} \cdot N_{col,2}) \cdot N_{col,2} \quad (63)$$

$$v_{P_{col}\parallel\infty} = v_{P_{col}\infty} \Leftrightarrow v_{P_{col}\perp\infty} \quad (64)$$

$$v_{P_{col}\varphi\infty} = v_{P_{col}\perp\infty} + \Phi v_{P_{col}\parallel\infty} \quad (65)$$

$$N_{col\infty} = \|v_{P_{col}\varphi\infty}\| \quad (66)$$

$$A_\infty = |v_{P_{col}\varphi\infty}| \quad (67)$$

$$B_\infty = \frac{1}{m_1} + \frac{(r_{col,1} \times N_{col\infty})^2}{J_1} \quad (68)$$

$$\Delta p_\infty = 2 \frac{A_\infty}{B_\infty} \quad (69)$$

$$\Delta p_\infty = \Delta p_\infty \cdot N_{col\infty} \quad (70)$$

4 Partikelsystem

Objekte können in der Szenenbeschreibung bereits eine Geschwindigkeit zugeordnet haben. Beschleunigungen können auf diese Weise nicht gegeben werden, weil sie nur in einem Zeitpunkt existieren und nicht durch die Trägheit der Objekte erhalten bleiben.

Sollen Objekte also nicht nur durch Kollisionen in Bewegung geraten können, so sind globale Kräfte nötig, die im Sinne eines Partikelsystems auf die Objekte einwirken.

4.1 Strömungsfelder

Durch Strömungsfelder kann das Medium modelliert werden, das die Objekte umgibt. Die verschiedene Zähflüssigkeit von beispielsweise Luft und Wasser wird Viskosität genannt. Jedem Punkt im Raum ist ein Strömungsvektor zugeordnet. Je größer die Viskosität des Mediums, desto stärkere Kräfte wirken auf die Objekte ein, um deren Bewegung dem Umgebungsfluß anzugleichen.

Durch Addition einfacher Flußprimitiven, z.B. gleichförmiger Wind, Wirbel oder punktförmiger Sog, können komplexe Strömungsfelder modelliert werden.

Ordnet man dem Medium zudem eine Dichte zu, so kann für Objekte deren Dichte bekannt ist auch Auftrieb modelliert werden. Die Dichte eines Objektes ist seine Masse pro Volumen.

4.2 Kraftfelder

Während bei Strömungsfeldern die Krafteinwirkung auf die Objekte von deren Bewegung abhängig ist, modellieren Kraftfelder Kräfte, die unabhängig davon wirken.

Erdanziehung ist beispielsweise eine konstante Kraft zum Erdmittelpunkt hin.

Auch hier bietet es sich an, einfache Primitiven zu komplexen Kraftfeldern zu kombinieren. Durch punktförmige Anziehung bzw. Abstoßung können Gravitation oder Magnetismus modelliert werden. Denkbar sind auch Kraftfelder, die an bewegliche Objekte gekoppelt sind.

5 Kollisionserkennung

5.1 Problematik

Die für den naiven Betrachter völlig selbstverständlich erscheinende Tatsache, daß sich Objekte, die aufeinander stoßen, nicht gegenseitig durchqueren, sondern, daß sie voneinander abprallen, birgt bei der Realisierung in einem 3D-Rendering Tool wie dem MRT⁴ einige erstaunlich große Schwierigkeiten.

Bevor, durch die Umsetzung der in Kapitel 3 betrachteten Grundlagen, realistisch erscheinende Reaktionen der Objekte implementiert werden können, muß zunächst die Frage geklärt sein, ob, wo und wann es überhaupt zu Objektkollisionen kommt.

5.1.1 Berechnungsaufwand

Will man zu einem bestimmten Zeitpunkt in einer Szene, die aus n Objekten besteht, alle Berührungen und Durchdringungen bestimmen, so ist ein Test für jedes Objekt mit jedem nötig. Für die Komplexität gilt somit:

$$\sum_{i=1}^{n-1} i = \frac{(n \Leftrightarrow 1) \cdot n}{2} = O(n^2) \quad (71)$$

In Kapitel 5.3 werden Techniken betrachtet, mit Hilfe derer trotz dieses Aufwandes gute Resultate erreicht werden. Aber auch die Ausführung eines einzelnen Tests ist noch eine aufwendige Angelegenheit, je nachdem welche Objekttypen beteiligt sind.

5.1.2 Programmieraufwand

Will man für zwei beliebig geformte Objekte bestimmen, ob sie sich berühren oder durchdringen, so müssen die geometrischen Eigenschaften der konkreten Objekttypen berücksichtigt werden. Eine Kollisionserkennungsroutine wird in der Regel also nur einen Kollisionstest für zwei bestimmte Objekttypen durchführen können. Möchte man für k implementierte Objekttypen paarweise alle Kollisionstests ausführen können, so bedeutet das einen Programmieraufwand von quadratischer Komplexität $O(k^2)$. Möchte man einem bestehenden System einen $k + 1$ ten Objekttyp hinzufügen, so sind k Kollisionstestroutinen für alle bereits bestehenden Objekttypen zu implementieren. Hinzu kommt eine Routine für Tests mit Objekten des eigenen Typs.

Mit Hilfe der objektorientierten Programmierung kann dieser Aufwand verringert werden, indem Lösungen für ganze Objektklassen gesucht werden. Da die im Rahmen dieser Arbeit verwendete Programmiersprache C++ jedoch keine Möglichkeit bietet, einer Funktion zwei Objekttypen zuzuordnen (sie ist immer *member-function* einer *class*), muß explizit ein *multiple dispatching* [SEYA96] realisiert werden, das dieses leistet.

5.1.3 Diskretisierung

Testet man in einer bewegten Szene nur zu einzelnen Zeitpunkten auf Kollisionen, so können Durchdringungen, die nur zu einem Teil des Zwischenintervalls bestanden, übersehen werden.

⁴Das MRT wurde von der Computergrafikgruppe der Abteilung III des Instituts für Informatik der Universität Bonn entwickelt.

Es vereint approximative Darstellungstechniken, durch geeignete Schattierung von Polygonen, mit fotorealistischem Raytracing und Radiosity.

Seine Stärken sind Plattformunabhängigkeit und Erweiterbarkeit. Dies wird durch die objektorientierte Struktur der C++-Implementation und durch die Anzeige mittels CGI-3D, das an verschiedenste Grafikumgebungen angepaßt werden kann, erreicht.

Kollisionen dieser Art können aus der Sicht des Betrachters zweierlei Natur sein. Erstens (Typ 1) kann es dazu kommen, daß zwei sich nahezu parallel bewegende Objekte eine kurze Berührung haben, die sich sofort wieder auflöst. Ursache können gekrümmte Flugbahnen der Objekte oder hervorstehende Teile, wie z.B. die Ecke eines Würfels sein.

Es ist jedoch auch denkbar, daß ein schnelles Objekt ein anderes völlig durchquert (Typ 2). Ein Ball könnte beispielsweise durch eine Tischplatte fallen.

Werden Kollisionen vom Typ 1 übersehen, so fällt dies dem Betrachter in der Regel nicht auf, da Überschneidungen zu keinem Zeitpunkt sichtbar werden und da keine Richtungsänderungen oder andere Reaktionen erwartet werden. Kollisionen vom Typ 2 sollten jedoch unbedingt erkannt werden, damit Animationen physikalisch plausibel sind.

Je nach Objekttyp kann der Übergang zwischen beiden Kollisionstypen fließend sein. Es werden später Objekttypen vorgestellt, für die Typ 2-Kollisionen sicher erkannt werden. Dies ist besonders für die Objekttypen wichtig, die zur Begrenzung der Szene verwendet werden.

5.1.4 Kollisionsreihenfolge

Hat man eine Reihe von Kollisionen in einem Zeitintervall erkannt, so kommt es mit Sicherheit nur zur ersten. Spätere Kollisionen können durch vorherige verhindert und zusätzliche Kollisionen können ausgelöst werden.

5.2 Kollisionserkennungsalgorithmen

Es gibt verschiedene Herangehensweisen an die Kollisionserkennung. Sie unterscheiden sich nicht nur hinsichtlich ihrer Komplexität, sondern auch in der Genauigkeit der Kollisionspunkt- und Kollisionszeitpunktbestimmung.

Der vorgestellte Algorithmus I ist im Rahmen dieser Arbeit nicht implementiert worden. Algorithmus II hingegen wurde längere Zeit favorisiert, bevor aus den unten genannten Gründen Algorithmus III der Vorzug gegeben wurde.

5.2.1 Algorithmus I

Aufgrund der hohen Zeitkomplexität bei der Bestimmung aller Durchdringungen in einer Szene von n Objekten ($O(n^2)$) sollte ständiges Neutesten der Szene vermieden werden.

Folgender Algorithmus hat deshalb nach einem Initialisierungsschritt mit Aufwand $O(n^2)$ nur noch linearen Aufwand pro Kollision.

Er funktioniert nur, wenn Position und Lage der Objekte analytisch für beliebige Zeitpunkte, unter Vernachlässigung anderer Kollisionen, bestimmt werden können. Diese Einschränkung macht ihn leider praktisch unbrauchbar.

Besteht eine Szene nur aus Kugeln, die sich gleichmäßig geradlinig bewegen, so ist die Bestimmung der Kollisionszeitpunkte durch die Lösung eines Polynoms zweiten Grades möglich. Auch wenn die Kugeln gleichmäßig beschleunigt werden (z.B. durch Erdanziehung) ist eine Lösung noch möglich. Es muß jedoch bereits ein Polynom vierten Grades gelöst werden.

Außerdem ist darauf zu achten, daß es bei der Darstellung der Flugbahnen nicht zu Diskretisierungsfehlern kommt, wie es bei Partikelsystemen häufig der Fall ist.

Algorithmus

1. Kollisionszeitpunktbestimmung für alle Objektpaare
2. Jedes Objekt speichert nur Objekt und Zeitpunkt der frühesten Kollision

3. Animation bis zur ersten Kollision
4. Kollisionsreaktion
5. Kollisionszeitpunktbestimmung für die kollidierten Objekte mit allen anderen.
6. Speicherung der frühesten Kollision. Korrektur auch beim jeweiligen getroffenen Objekt, falls Zeitpunkt früher.
7. zurück zu 3

Korrektheit

- Von den gespeicherten Kollisionen muß nur die früheste mit Sicherheit eintreffen. Zu Beginn ist dies sicher.
- Dann wird diese Eigenschaft durch Schritt 6 weiterhin gewährleistet.

Aufwandsabschätzung

- Schritt 1, mit Komplexität $O(n^2)$, muß nur einmal ausgeführt werden.
- Schritt 5 hat lineare Komplexität $2 \cdot (n \Leftrightarrow 1) = O(n)$.
- Sei ein *Frame* das Zeitintervall, das einem Einzelbild der Animation entspricht und sei \bar{m} die durchschnittliche Zahl der Kollisionen pro Frame, so ist $O(\bar{m} \cdot n)$ die Komplexität pro Frame.
- Für eine Animation mit f Frames ergibt sich ein Gesamtaufwand von $O(n^2 + f \cdot \bar{m} \cdot n)$.

$f \cdot \bar{m}$ ist die Gesamtzahl der Kollisionen. Diese kann, je nach Szene, sehr groß sein. Bewegt sich ein Ball zwischen zwei parallelen Wänden hin und her, so kann durch Engerstellen der Wände eine beliebige Zahl von Kollisionen in einem beliebig kleinen Zeitintervall erzielt werden. Bewegen sich die Wände dabei noch so auf den Ball zu, daß die nächste Kollision innerhalb des verbleibenden Teilintervalls auftritt, kommt es sogar zu unendlich vielen Kollisionen.⁵

Im Rahmen dieser Arbeit sind die erreichten Einzelbildraten das entscheidende Effizienzkriterium. Der Initialisierungsaufwand kann also, solange er die Realisierbarkeit des Algorithmus nicht gefährdet, vernachlässigt werden. Algorithmus I hat somit linearen Aufwand mit den genannten Einschränkungen.

5.2.2 Algorithmus II

Der zweite vorgestellte Algorithmus umgeht die einschränkenden Annahmen, die den vorhergehenden unpraktikabel machen.

Kollisionen werden erst erkannt, wenn sie bereits geschehen sind.

Die Animation wird mit Hilfe der zuvor gesicherten Informationen auf den Kollisionszeitpunkt zurückgesetzt und von dort nach der Kollisionsreaktion fortgeführt.

Der Algorithmus erlaubt es, während kleiner Teilintervalle lineare Flugbahnen, wie es bei Partikelsystemen üblich ist, anzunehmen.

Dies vereinfacht auch die Berechnung des Kollisionszeitpunktes im verstrichenen Teilintervall. Für beliebige Objekte bleibt sie dennoch schwierig, besonders wenn sich diese zusätzlich zur linearen Bewegung noch drehen.

⁵Dieses Verhalten ist nur in virtuellen Umgebungen denkbar. In der Realität können Objekte nicht mit beliebiger Frequenz schwingen. Erreicht der Ball seine Eigenfrequenz, so wird aus der äußeren Schwingung eine innere und er bewegt sich nicht mehr.

Algorithmus pro frame

1. Sichern der Zustände.
2. Probeweise Ausführung des Zeitintervalls.
3. Bestimmung der ersten Kollision im Intervall.
4. Rücksetzung der Zustände.
5. Ausführung des Intervalls bis zur ersten Kollision (ohne Kollisionserkennung).
6. Kollisionsreaktion
7. Rekursive Ausführung des restlichen Intervalls.

Aufwandsabschätzung

- Schritt 3 hat Zeitkomplexität $O(n^2)$.
- Schritt 7 muß für jede Kollision im Intervall ausgeführt werden ($O(\bar{m})$).
- Pro frame ergibt sich $O(\bar{m} \cdot n^2)$.
- Der Gesamtaufwand beträgt $O(f \cdot \bar{m} \cdot n^2)$.

Dieser Algorithmus ist nicht so eingeschränkt brauchbar wie der erste. Den Lösungen einiger Probleme steht jedoch ein erheblicher Mehraufwand gegenüber. Die unverändert mögliche große Zahl von Kollisionen vervielfacht den quadratischen Zeitaufwand sogar noch zusätzlich.

5.2.3 Algorithmus III

Der im Rahmen dieser Arbeit realisierte Algorithmus diskretisiert auch die Kollisionszeitpunkte. Kollisionen, die in einem kleinen Zeitintervall (*physical step*) auftreten, werden als gleichzeitig aufgefaßt.

Die Frames der Animation sind zur Zusammenfassung aller Kollisionen ein zu grobes Zeitraster, weil es zu deutlich sichtbaren Durchdringungen kommen kann. Sie werden deshalb in p *physical steps* zerlegt. Dennoch sind Durchdringungen und auch Durchquerungen von Objekten nicht allein durch die Wahl von p auszuschließen. Sollen Durchquerungen bestimmter Objekte unmöglich sein, so ist dies unter Umständen mit einem Mehraufwand an Implementation und Laufzeit verbunden.

Die Zahl der Kollisionen in einem step ist jetzt auf $O(n^2)$ beschränkt, da jedes Objekt mit jedem höchstens einmal pro step kollidieren kann. Wegen der räumlichen Ausdehnung der Objekte ist es sogar als unwahrscheinlich anzusehen, daß $O(n^2)$ Kollisionen in einem step auftreten.

Eine Szene von n Kegeln, die alle zum selben Zeitpunkt mit den Spitzen zusammentreffen, wäre ein entsprechend konstruiertes Beispiel. In der Praxis wird dies selten vorkommen.

Algorithmus pro step

1. Ausführung eines *physical steps*.
2. Test auf Berührungen und Durchdringungen.
3. Kollisionsreaktion

Aufwandsabschätzung

- Schritt 2 hat Aufwand $O(n^2)$.
- Bei p physical steps pro frame ist der Gesamtaufwand $O(f \cdot p \cdot (\bar{m} + n^2))$.

5.3 Beschleunigung

Bei den betrachteten Beschleunigungstechniken zur Kollisionserkennung standen entsprechende Techniken aus strahlbasierten Rendering-Algorithmen (z.B. Raytracing) Pate [Mül97].

Diese Techniken haben jedoch durchweg einen Nachteil: Sie argumentieren damit, daß sich ein gewisser Vorverarbeitungsaufwand zu Beginn lohnt, um die Darstellung selbst dann erheblich zu beschleunigen. Bei Animationen bedeutet *Beginn* jedoch einmal pro Frame, so daß der Vorverarbeitungsaufwand von erheblicher Bedeutung für die erreichbaren Frame-Raten ist.

Man kann vorweg sagen, daß sich der ständige Neuaufbau einer ausgefeilten Datenstruktur (Hüllkörperhierarchie oder Raumunterteilung) nicht lohnt. Es werden deshalb ein besonders einfacher Ansatz mit ständiger Neuinitialisierung eines regulären Gitters und ein adaptiver Raumunterteilungsansatz, bei dem jedoch nicht immer die vollständige Struktur neu berechnet werden muß, vorgestellt.

5.3.1 Hüllkörper

Als einfachste Beschleunigungstechnik aus dem Renderingbereich, läßt sich der Ansatz übernehmen, vor einem „teuren“ Kollisionstest zweier spezieller Objekte, einen Test auf einfache Hüllkörper anzuwenden. Dieser macht in vielen Fällen den speziellen Test unnötig.

Besonders schnell ist ein Kollisionstest zweier Kugeln: Hier muß nur der Abstand der Mittelpunkte mit der Summe der Radien verglichen werden.

Da es sich im Rahmen der objektorientierten Realisierung ferner anbietet, physikalischen Objekten - implementiert in der Basisklasse - das Verhalten einer Kugel zu geben, stehen kugelförmige Hüllkörper bereits zur Verfügung. Ihnen ist gegenüber den gängigen Quadern in Hauptachsenlage der Vorzug zu geben, weil sie die rotierenden Objekte meist besser annähern.

5.3.2 Reguläres Gitter

Besonders einfach zu implementieren und für den Vergleich mit ausgefeilteren Beschleunigungsverfahren deshalb besonders interessant sind reguläre Gitter.

Die gesamte Szene wird in gleich große Raumelemente *Zellen* eingeteilt. Für jede Zelle wird eine Liste der Objekte verwaltet, deren Hüllkörper (Quader in Hauptachsenlage) einen nichtleeren Schnitt mit ihr haben.

Bei Kollisionstests müssen nur solche Objektpaare berücksichtigt werden, die zu einer gemeinsamen Raumzelle gehören.

Da Objekte häufig mehrere Raumzellen einnehmen, ist noch zu verhindern, daß Objektpaare für verschiedene Zellen mehrfach getestet werden.

Um dies zu erreichen, hinterläßt ein Objekt eine Kennung bei jedem anderen, mit dem es getestet wird. Mit ihrer Hilfe wird erkannt, wenn ein Objektpaar erneut getestet werden soll.

Nachdem ein Objekt mit allen anderen Objekten seiner Raumzelle verglichen wurde, wird es auch mit den Objekten aller anderen Raumzellen verglichen, in denen es sich befindet. Um welche Raumzellen es sich hierbei handelt, kann leicht mit Hilfe des Hüllkörpers bestimmt werden. Bei diesen Tests müssen Objekte mit der Kennung des ersten Objektes nicht erneut berücksichtigt werden.

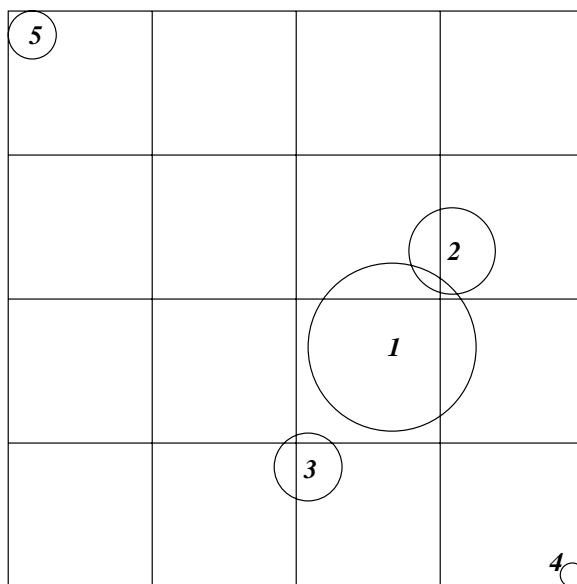


Abbildung 4: Reguläres Gitter

Nachdem ein Objekt mit allen anderen Objekten einer Raumzelle verglichen wurde, kann es aus dieser Raumzelle entfernt werden. Dies bedeutet keinen Mehraufwand, wenn die Listen der Raumzellen doppelt verkettet sind. Da die Listen aller Raumzellen eines Objektes bereits für die Kollisionserkennung durchlaufen werden müssen, kann das Objekt sofort entfernt werden, ohne daß es neu gesucht werden muß.

Eine zusätzliche Beschleunigung kann erreicht werden, wenn die Gesamtszene ihre Ausdehnung nicht ändert. Dann ändern sich die Positionen der Raumzellen nicht und unbewegliche Objekte müssen nicht neu einsortiert werden.

In diesem Falle dürfen jedoch keine unbeweglichen Objekte aus den Raumzellen gelöscht werden. Sie müssen stattdessen markiert werden, damit sie nicht mehrfach getestet werden.

5.3.3 Adaptive Raumunterteilung

Das zweite Beschleunigungsverfahren ist, wie das reguläre Gitter, zu den Raumunterteilungsverfahren zu zählen. Die Szene wird jedoch in unregelmäßige Raumelemente aufgeteilt.

Die Aufteilung erfolgt rekursiv. Ein Raumelement wird immer quer zu seiner maximalen Ausdehnung bzgl. einer der drei Hauptachsen halbiert. Eine Halbierung kann ein leeres und ein unverändert gefülltes Raumelement erzeugen. Unsymmetrische Aufteilungen würden diesen Nachteil verhindern und für eine ausbalancierte Baumstruktur sorgen. Sie können sich jedoch nachteilig auswirken, wenn die Objekte sich bewegt haben. Denn dann wären die Kriterien für die ursprüngliche Aufteilung nicht mehr gegeben und Korrekturen, die den Baum erneut ausbalancierten, kämen einer unerwünschten Neukonstruktion gleich.

Objekte, die mehrere Raumzellen einnehmen, werden auf höheren Hierarchieebenen gespeichert. Dies macht auch das Hinterlassen einer ID, wie beim regulären Gitter, unnötig.

Die so entstehende Binärstruktur hat den Vorteil, daß Regionen mit höherer Objektdichte feiner unterteilt werden als solche, in denen sich nur vereinzelt Objekte befinden.

Bei regulären Gittern zieht eine entsprechend feine Aufteilung die Erzeugung unzähliger leerer Zellen nach sich. Der entstehende Aufwand übersteigt dann leicht den der verhinderten Kollisionstests.

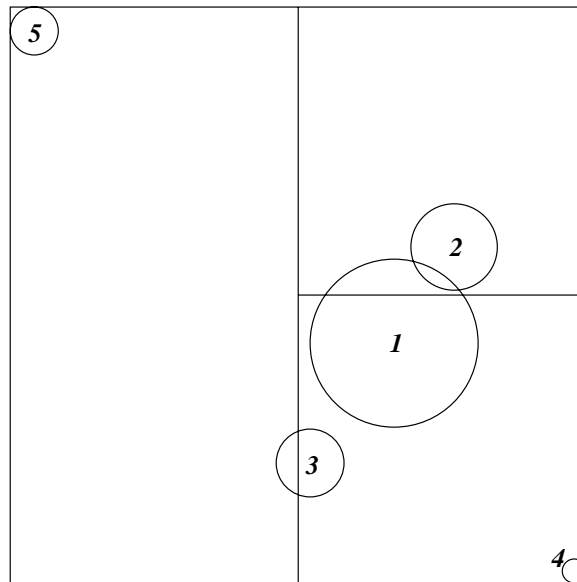


Abbildung 5: Adaptive Raumunterteilung

Da sich die Objekte bewegen und der Binärbaum nicht für jedes Einzelbild neukonstruiert werden soll, müssen bestimmte Funktionen zur Modifizierung des Baumes zur Verfügung stehen:

- Ein Objekt, das den Rand eines Raumelements erreicht, muß auch in benachbarte Raumzellen eingefügt werden, also in eine höhere Hierarchieebene aufsteigen.
- Ein Objekt, das sich völlig in eine Raumzelle hineinbewegt hat, kann weiter unten in die Hierarchie verschoben werden.
- Raumzellen, aus denen sich zu viele Objekte entfernt haben, können zusammengefaßt werden.
- Raumzellen, in denen sich zu viele Objekte befinden, müssen aufgeteilt werden.
- Für Objekte die sich aus der Szene hinaus bewegen, müssen zusätzliche Raumzellen erzeugt werden.
- Die Ausdehnung der Szene kann auch abnehmen.

5.3.4 Quantitativer Vergleich

Kollisions- und Hüllkugeltests Um die Qualität der Beschleunigungsverfahren beurteilen zu können, wurde zunächst die Zahl der Kollisionstests pro *Step* betrachtet. Diese wird von *mrtphys* (s. Kapitel 7), das auch als Testumgebung gedacht ist, nach jedem *Frame* für den letzten *Step* ausgegeben.

Da diese Zahl jedoch von *Step* zu *Step* verschieden ist, wurde der Mittelwert über 640 *Steps* ermittelt. Dies entspricht zehn Sekunden Simulationszeit bei acht *Frames per Second* und acht *Steps per Frame*.

Es wurden sieben Beispielszenen getestet, die alle von sechs Begrenzungsflächen auf den Einheitswürfel ($x, y, z \in [0, 1]$) begrenzt sind. Zu diesen sechs Objekten kommen noch n^3 bewegliche Objekte (Kugeln und Tetraeder) hinzu, wobei n die Werte 2 bis 8 hat. Ferner wurden drei Beschleunigungsverfahren getestet:

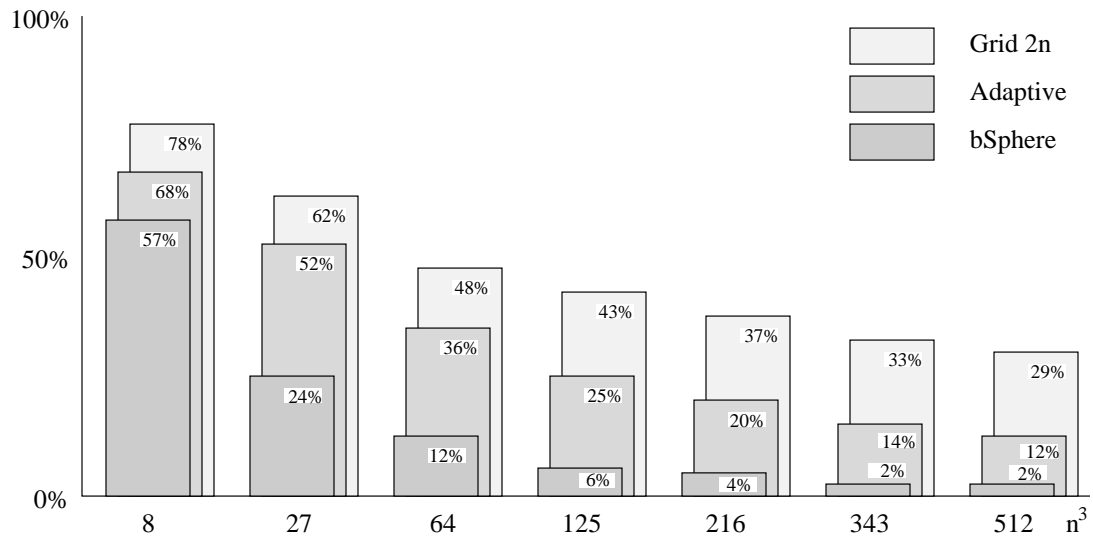


Abbildung 6: Kollisionstestvermeidung durch Hüllkugeln

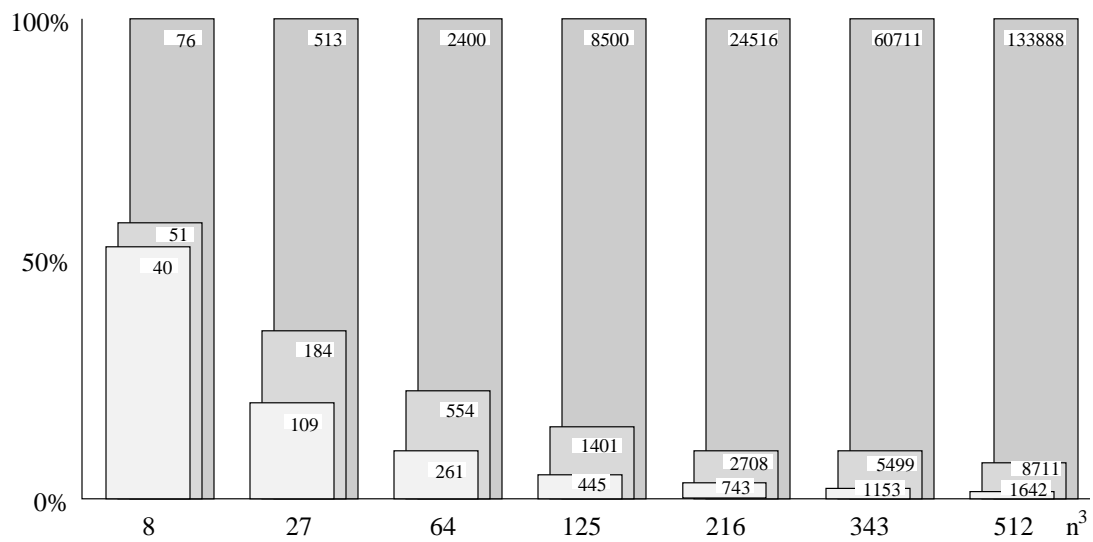


Abbildung 7: Hüllkugeltests pro Step

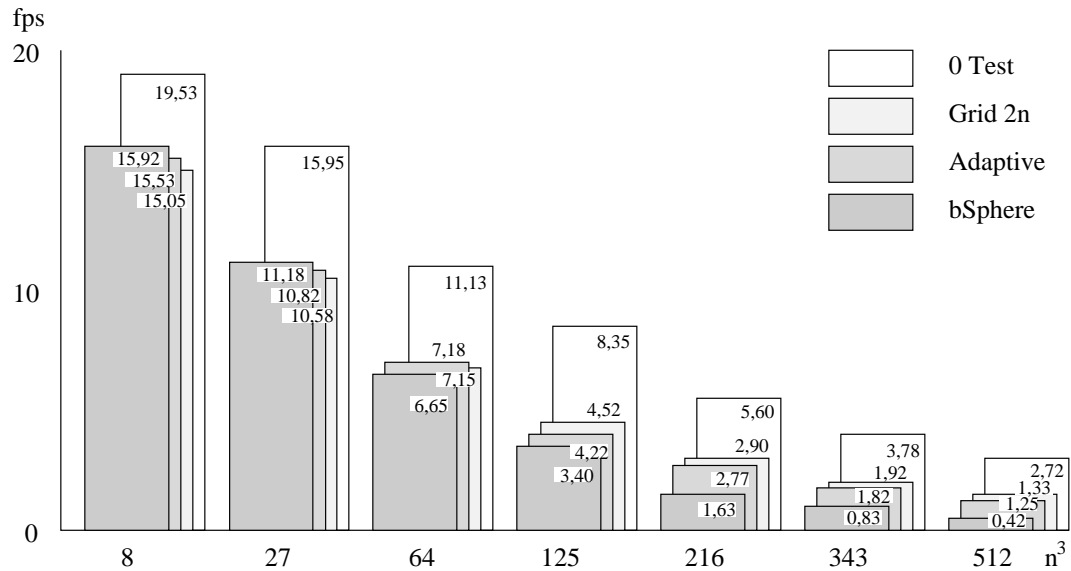


Abbildung 8: Frames per Second

bSphere wendet nur Hüllkugeln zur Beschleunigung an und testet jedes Objekt mit jedem. Auch die beiden anderen Verfahren verwenden Hüllkugeln. *Adaptive* verwendet zusätzlich ein adaptives Raumunterteilungsverfahren (s. 5.3.3 u. 6.4.2), und *Grid 2n* verwendet ein reguläres Gitter der Auflösung $2n$ (s. 5.3.2 u. 6.4.1).

Abbildung 6 zeigt den Anteil der Hüllkugeltests, die einen speziellen Kollisionstest nötig machen. Es wird deutlich, daß die meisten Kollisionstests bereits durch den Einsatz eines Hüllkugeltests vermieden werden.

Je schlechter ein Beschleunigungsverfahren die Objekte strukturiert, desto mehr überflüssige Hüllkugeltests werden gemacht. Bei 512 Objekten und *bSphere*-Beschleunigung machen nur 2% der Hüllkugeltests einen speziellen Kollisionstest nötig.

Es wird deutlich, daß auf Hüllkugeltests nicht verzichtet werden kann. Abbildung 7 zeigt jedoch, wie die Zahl der Hüllkugeltests bei steigender Objektzahl zunimmt (Absolutwerte bei 100%). Statt der speziellen Kollisionstests müssen diese deshalb zum Vergleich der Beschleunigungsverfahren herangezogen werden.

Die Leistung der Verfahren *Adaptive* und *Grid 2n* ist enorm. Bei 512 Objekten macht *Grid 16* nur 1642 der 133888 möglichen Hüllkugeltests. ⁶ 29% davon entspricht 472 Kollisionstests, was wesentlich weniger als 2% von 133888 ist, was 2014 Kollisionstests entspricht. ⁷

Adaptiv macht deutlich mehr Hüllkugeltests und somit auch mehr Kollisionstests nötig als *Grid 2n*.

Einzelbildraten Wichtigstes Effizienzkriterium sind die erreichten Einzelbildraten (*Frames per Second*). Diese wurden mit einem PC mit Pentium Pro 200 CPU, 64MB Arbeitsspeicher und einer 3D-Graphikkarte ermittelt.

Die Einzelbildrate läßt sich, wie die Kollisionstestzahl, von *mrtphys* ablesen. Die in Abbildung 8 dargestellten Resultate sind jedoch wieder ein Mittelwert. Zugrunde liegt die Zahl der Einzel-

⁶Da die sechs Begrenzungsobjekte untereinander nicht getestet werden, läßt sich die von *bSphere* benötigte Zahl von Hüllkugeltests bei $m = n^3$ Objekten auch durch $\frac{1}{2}(11m + m^2)$ berechnen.

⁷Bei der Berechnung der %-Werte wurden diese auf ganze % gerundet.

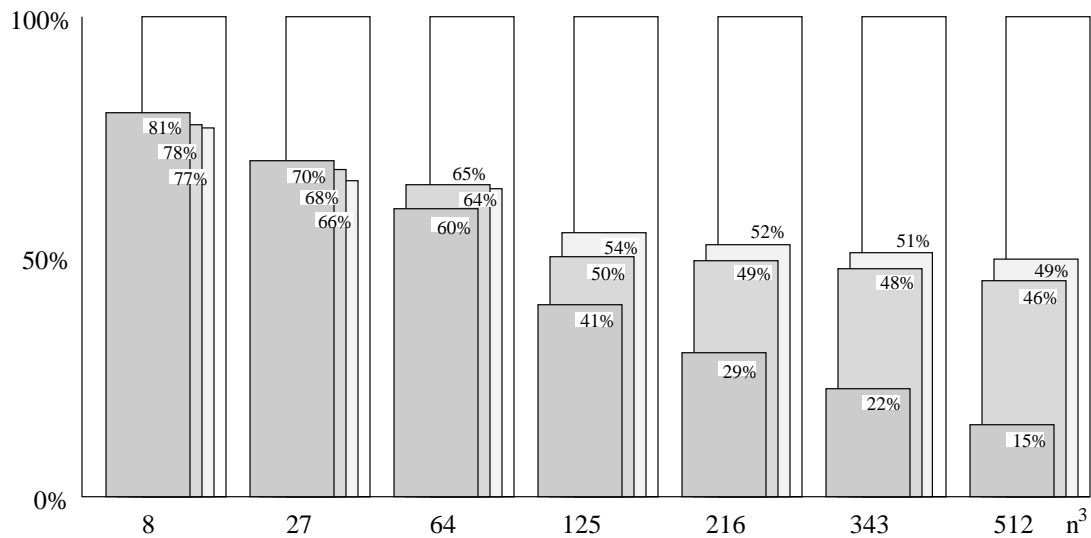


Abbildung 9: Performance-Verlust durch physikalische Funktionalität

bilder die innerhalb einer Minute dargestellt werden konnten. Es wurden ebenfalls acht *Steps per Frame* berechnet.

Abbildung 8 zeigt, wie dramatisch die Einzelbildraten bei zunehmender Objektzahl abnehmen. Die *0 Test*-Werte sind die erreichten Einzelbildraten der selben Szenen mit einem modifizierten *mrtphys*. Bei diesem wurde die physikalische Funktionalität deaktiviert. Sie machen deutlich, daß allein durch das Rendern der vielen Einzelflächen bei hoher Objektzahl unzumutbare Einzelbildraten entstehen.

Abbildung 8 stellt die Einzelbildraten der physikalischen Animationen in Relation zu den *0 Test*-Werten dar. Es wird deutlich, daß Animationen bei simpler Hüllkugel-Beschleunigung auch dann unmöglich bleiben, wenn die Renderingtechnik ausreichend schnelle Animationen vieler Objekte zuläßt.

Der Performance-Verlust durch physikalische Funktionalität steigt jedoch bei zunehmender Objektzahl deutlich langsamer, wenn bessere Verfahren gewählt werden.

Beim Vergleich der beiden Verfahren *Adaptive* und *Grid 2n* fällt auf, daß die bessere Hüllkugeltestvermeidung von *Grid 2n* offenbar teuer erkaufte ist. *Adaptive* erreicht mit seiner hierarchischen Datenstruktur ähnlich gute Einzelbildraten, obwohl es mehr Hüllkugeltests zu bewältigen hat.

Hinzu kommt, daß sich die Objekte in den Beispielszenen sehr gleichmäßig im Einheitswürfel verteilen. Bei großen Szenen, in denen sich die Objekte an verschiedenen Stellen lokal häufen, wird *Grid 2n* sicherlich deutlich schlechtere Resultate liefern, während sich *Adaptive* darauf einstellen kann. Es hängt somit vom Aufbau der Szene ab, welchem Beschleunigungsverfahren der Vorzug zu geben ist.

6 Implementierung physikalischer Objekte im MRT

An dieser Stelle werden die Klassen und ihre Funktionen beschrieben, die zusammen die konkrete Realisierung der vorgestellten Konzepte im MRT bilden. Die Header dieser Klassen sind in Anhang A noch einmal in alphabetischer Reihenfolge aufgeführt.

Die physikalischen Berechnungen wurden bereits in den vorangegangenen Kapiteln beschrieben. Für ihre Programmierung sei auf die C++-Quelltexte verwiesen, die Bestandteil dieser Arbeit sind. Dies gilt generell für Details, die über die hier erwähnten hinaus gehen.

6.1 `t_VelocityField`

Im Rahmen dieser Arbeit wurde nur der Ansatz eines Partikelsystems implementiert.

`t_VelocityField` dient als Basisklasse für Flußprimitiven. Eine abgeleitete Klasse muß die Funktion

```
virtual t_3DVector velocity (const t_3DVector& position) const = 0
```

so überladen, daß sie zu jeder Position einen Strömungsvektor liefert. Die statische Funktion

```
static t_3DVector allVel (const t_3DVector& position) const
```

liefert die Summe aller Vektoren der instantiierten Strömungsfelder. Da selten auf Strömungsvektoren einzelner Flußprimitiven zugegriffen wird, nimmt diese Funktion der Anwendung die Arbeit ab, eine Liste der Primitiven zu führen und über diese zu addieren.

Da dies nicht Gegenstand dieser Arbeit ist, sind noch keine Flußprimitiven implementiert. Die Viskosität wirkt sich jedoch bereits bremsend auf die Animationen aus.

6.2 `t_ForceField`

`t_ForceField` dient als Basisklasse für Kraftfelder.

Als einziges Beispiel ist `t_EarthGravity` implementiert. Es überlädt die Funktion

```
virtual t_3DVector acceleration (const t_3DVector& position,
                                t_Real mass) const = 0
```

so, daß immer die Erdbeschleunigungskonstante g zurückgegeben wird.

Im allgemeinen hängt die Beschleunigung durch eine Kraft jedoch von der Masse des Objektes ab ($\mathbf{F} = m \cdot \mathbf{a}$), so daß diese in der Regel nicht ignoriert werden wird.

Die statische Funktion

```
static t_3DVector allAcc (const t_3DVector& position,
                          t_Real mass) const
```

liefert wiederum die Summe aller Beschleunigungsvektoren der instantiierten Felder.

6.3 `t_PhysicalObject`

Die Implementierung der Basisklasse `t_PhysicalObject` bildet das Herzstück des praktischen Teils dieser Arbeit.

`t_PhysicalObject` ist von `t_RefObject` und somit auch von `t_SurfaceObject` abgeleitet, welches wiederum von `t_Object` abgeleitet ist. Dadurch stehen die Fähigkeiten der geometrischen Objekte des MRT den physikalischen Objekten zur Verfügung und es müssen nur noch die physikalischen Eigenschaften hinzugefügt werden.

Ein erster Schritt in diese Richtung ist schon mit der Ableitung von `t_RefObject` getan. Diese Klasse fügt den Objekten eine Transformationsmatrix hinzu, die unter anderem Translation und Rotation ermöglicht. Spätere physikalische Erweiterungen können noch von der Möglichkeit Gebrauch machen, Objekte mit Hilfe der Transformationsmatrix zu deformieren.

Keine der implementierten Funktionen ist *pure virtual*. Ein abgeleitetes Objekt hat zunächst das Verhalten einer Kugel. Kollisionen werden bereits erkannt, wenn eine Durchdringung der Hüllkugel (`boundingSphere`) vorliegt. Auch das Rotationsverhalten (s.a. Annahme 3) und die Kollisionsreaktionen entsprechen einer Kugel.

6.3.1 `physicalFunction`

Die Funktion

```
virtual void physicalFunction(t_Real timeStep)
```

berechnet den physikalischen Zustand des Objektes nach Verstreichen der Zeit `timeStep`. Wird diese Funktion nicht überladen, so wird das Objekt gemäß seiner Geschwindigkeit und seiner Winkelgeschwindigkeit bewegt. Ferner werden die instantiierten Strömungs- und Kraftfelder berücksichtigt, die sich beschleunigend bzw. bremsend auswirken können.

Die Bewegung beschleunigter Objekte wird für die Dauer des Intervalls als gleichförmig und geradlinig angenommen. Diese Form der Interpolation wird *Euler-steps* genannt. Bei der Implementierung eines besseren Partikelsystems kann an dieser Stelle auch eine bessere Annäherung der Flugbahn durch andere Interpolationsverfahren (z.B. *Runge-Kutta*) realisiert werden.

Um gleichzeitiges Auftreten verschiedener Interpolationsverfahren zu vermeiden, empfiehlt es sich jedoch, ein solches nicht durch Überladen der `physicalFunction` zu realisieren, sondern - ähnlich wie beim Kollisionserkennungsverfahren - eine Möglichkeit vorzusehen, die entsprechende Routine komplett auszutauschen.

Verwendet man einen Kollisionserkennungsalgorithmus, der zur Kollisionszeitbestimmung die Positionen von Objekten während des Intervalls rekonstruiert, so ergibt sich durch verbesserte Interpolationsverfahren zusätzlicher Aufwand.

Zur Realisierung der Rotation wird das Objekt mittels einer Rotationsmatrix R transformiert. Für diese gilt nach [Fel92, S. 246-250]:

$$R = T \cdot R_x \cdot R_y \cdot R_z \cdot R_y^{-1} \cdot R_x^{-1} \cdot T^{-1} \quad (72)$$

T verschiebt das Objekt zunächst so, daß die Rotationsachse durch den Ursprung verläuft. Durch die Rotationsmatrizen R_x und R_y wird das Objekt so um x- und y-Achse gedreht, daß die Rotationsachse mit der z-Achse zusammenfällt. R_z dreht das Objekt dann um den gewünschten Winkel um die z-Achse. Anschließend werden die zuvor getätigten Transformationen rückgängig gemacht, so daß nur die Rotation um die Rotationsachse bleibt.

Die Translation T muß bei jedem Schritt abhängig von der Position des Objektes ausgeführt werden. Die Rotationsmatrizen ändern sich hingegen nur, wenn sich die Winkelgeschwindigkeit des Objektes ändert - dies ist in der Regel bei einer Kollision der Fall - oder wenn sich die Länge der Zeitintervalle (*physical steps*) ändert. Dann werden sie durch `calcRotMatrix` neu berechnet.

Geschwindigkeit und Rotation sind nur zwei mögliche physikalische Eigenschaften von Objekten. Abgeleitete Objekte könnten auch Eigenschaften wie veränderliche Farbe, Masse oder begrenzte Lebensdauer in der `physicalFunction` realisieren.

6.3.2 physicalFrame

An dieser Stelle ist Algorithmus III implementiert:

`physicalFunction` ist ein *protected-member* der Klasse. Durch das *public-member*

```
static void physicalFrame (t_Real timeStep,
                          int    physicalSteps)
```

wird es für alle instantiierten physikalischen Objekte `physicalSteps` mal aufgerufen. Außerdem wird der Kollisionserkennungsalgorithmus aufgerufen, der mit

```
static void collisionDetectionMode (t_CollisionDetection* mode)
```

gesetzt wurde. Dieser Algorithmus ruft neben den Kollisionserkennungsroutinen auch die Kollisionsreaktionsroutinen auf. Deren Aufruf kann also auch - abweichend von Algorithmus III - unmittelbar nach der Erkennung erfolgen, was keinen Laufzeitunterschied ausmacht.

Auch beim Aufruf von `checkIntersections`, bei dem die Szene auf die Existenz von Durchdringungen geprüft wird, wird der so gesetzte Modus verwendet.

6.3.3 Kollisionsrelevante Daten

Kollisionspunkt P_{col} , Kollisionsnormale N_{col} und Kollisionsradius r_{col} werden während der Kollisionserkennung berechnet und durch `colPoint`, `colNormal` und `colRadius` gesetzt sowie für die Kollisionsreaktion bereitgestellt. Für diese stehen außerdem Kollisionspunktgeschwindigkeit $v_{P_{col}}$ und Kollisionspunktimpuls $p_{P_{col}}$ zur Verfügung.

Zur Vermeidung mehrfacher Kollisionstests kann mittels `colMailboxId` eine Kennung hinterlassen werden.

6.3.4 boundingSphere

Der kugelförmige Hüllkörper ist sowohl für das Ausgangsverhalten der abgeleiteten Objekte, als auch für die Beschleunigung der Kollisionserkennung von Bedeutung. Die Berechnung der Hüllkugel erfolgt nicht bei jedem Aufruf von `boundingSphere`, sondern sie wird explizit durch `calcBSphere` aufgerufen. Die Basisimplementation berechnet eine Hüllkugel, die den gesamten Hüllkörper `boundingVolume` einschließt. Es lohnt sich daher fast immer, `calcBSphere` zu überladen.

6.3.5 translate and transform

Die entsprechenden Funktionen der Basisklasse `t_RefObject` modifizieren nur die darstellungsrelevanten Daten, die dort abgelegt sind. Die Daten, die lokal bei den Objekten gespeichert sind, müssen jedoch ebenfalls transformiert werden. In `t_PhysicalObject` ist beispielsweise der Schwerpunkt `v_massCenter` betroffen. Abgeleitete Klassen sollten also, nach der Modifikation der lokalen Daten, noch die entsprechende Funktion ihrer Basisklasse aufrufen.

6.3.6 genericCollisionDetect

Als Ausgangsverhalten der Basisklasse werden Kollisionen bereits erkannt, wenn eine Durchdringung der Hüllkugel vorliegt.

Da vor dem Aufruf einer Kollisionserkennungsfunktion immer auf einen nichtleeren Schnitt der Hüllkugeln getestet wird (s. 5.3.1), wird von `genericCollisionDetect` immer eine Kollision erkannt. Es sind somit nur noch die kollisionsrelevanten Daten zu ermitteln.

Der Kollisionspunkt liegt bei Kugeln immer auf der Geraden, die durch beide Mittelpunkte verläuft. Da in der Regel eine echte Durchdringung vorliegt und nicht nur eine punktförmige Berührung, werden für beide Objekte verschiedene Kollisionspunkte ermittelt. Diese befinden sich dort, wo die erwähnte Gerade, die Hüllkugeloberflächen schneidet.

Die Kollisionsradien sind kollinear zu dieser Geraden. Die Kollisionsnormalen entsprechen den normierten Kollisionsradien.

6.3.7 genericCollisionResponse

Die Kollisionsreaktions-Routine realisiert bereits alle in Kapitel 3 beschriebenen Konzepte, einschließlich der Oberflächenreibung und Stößen mit Objekten unendlicher Masse. Sie ist daher nicht nur für den Einsatz mit Kugeln, sondern für beliebige starre Objekte geeignet.

6.4 t_CollisionDetection

Um verschiedene Beschleunigungstechniken bei der Kollisionserkennung erproben zu können, wurde der Kollisionserkennungsalgorithmus aus der Klasse `t_PhysicalObject` ausgelagert. Dort könnte er fest in der Funktion `physicalFrame` implementiert werden. Stattdessen wird dort die Funktion

```
void detectAndResponse()
```

einer Instanz des Typs `t_CollisionDetection` aufgerufen. Ist diese Instanz vom Typ der Basisklasse selbst, so wird ein einfacher Algorithmus, der jedes physikalische Objekt mit jedem vergleicht, ausgeführt. Verbesserte Verfahren können als abgeleitete Klassen implementiert werden.

6.4.1 t_CDRegularGrid

Dieses von `t_CollisionDetection` abgeleitete Verfahren realisiert die Kollisionsbeschleunigung wie in 5.3.2 beschrieben.

Zunächst wird das Gitter initialisiert. Dann werden mit Hilfe der Funktion

```
void setToGrid (const t_PhysicalObjectPtr& phyObject)
```

die Objekte einsortiert.

Dieses Verfahren muß nach jedem physikalischen Schritt neu gestartet werden. Es wird jedoch automatisch erkannt, ob sich die Ausdehnung der Gesamtszene geändert hat. Dann kann die Neuberechnung des Gitters und ein erneutes Einfügen der unbeweglichen Objekte vermieden werden.

Zur Hinterlassung einer ID wird die Funktion `colMailboxID` von `t_PhysicalObject` genutzt.

6.4.2 t_CDAdaptiveGrid

Diese ebenfalls von `t_CollisionDetection` abgeleitete Klasse realisiert das in 5.3.3 beschriebene Beschleunigungsverfahren.

Der Konstruktor erzeugt einen Binärbaum durch sukzessive Halbierung der Szene entsprechend ihrer größten Ausdehnung. Dies erledigt rekursiv die Funktion

```
void splitCell(TreeItem* item)
```

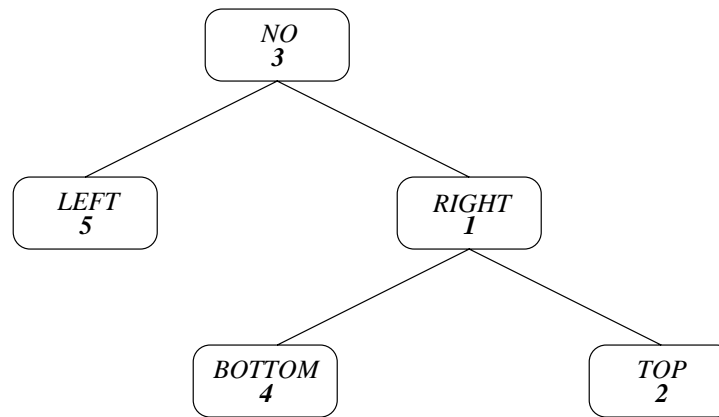


Abbildung 10: Szenenhierarchie zu Abb. 5

Während der Animation werden die Objekte lokal in dieser Struktur bewegt, wobei diese nach Bedarf der veränderten Objektdichte in den Teilszenen angepaßt wird.

Wird `detectAndResponse()` aufgerufen, so wird zunächst der Binärbaum den Bewegungen der Objekte seit dem letzten Aufruf angepaßt. Danach werden mit Hilfe der Funktionen

```

void detectAndResponse (TreeItem* item)
void detectAndResponse (PhyObjItem* objItem, TreeItem* item)
void detectAndResponse (PhyObjItem* objItem1,
                        PhyObjItem* objItem2)
  
```

die Kollisionen rekursiv erkannt, wobei die Vorteile der Hierarchie genutzt werden, indem nur solche Zweige berücksichtigt werden, deren Raumzellen einen nichtleeren Schnitt mit dem Hüllkörper des betreffenden Objektes haben.

Die Anpassung des Binärbaums geschieht in zwei Schritten:

- Zuerst werden durch

```
void moveUpObjects(TreeItem*)
```

alle Objekte in der Hierarchie nach oben bewegt, die sich aus ihrer Raumzelle hinaus bewegt haben. Abbildung 10 zeigt, daß an jedem Knoten die Richtung gespeichert ist, in der er neben seinem Nachbarknoten liegt. Zu diesem Zweck wird der Aufzählungstyp

```
enum direction {NO, LEFT, RIGHT, BOTTOM, TOP, BACK, FRONT}
```

verwendet.

Da bekannt ist, in welche Richtung sich ein Objekt aus seiner Zelle bewegt hat, können auf dem Weg nach oben alle Knoten überschlagen werden, die nicht mit der entgegengesetzten Richtung markiert sind. Die Funktion

```
direction moved(const t_BVol& bVol1, const t_BVol& bVol2)
```

liefert zu diesem Zweck gleich die entgegengesetzte Richtung, in der sich `bVol1` aus `bVol2` bewegt hat, zurück.

`moveUpObjects` überprüft auch ob Blattknoten des Binärbaums noch genügend Objekte enthalten, um diese gegebenenfalls mittels

```
void joinCell (TreeItem* item)
```

zusammenzufassen.

Hat ein Objekt die bisherige Szene verlassen, so wird sie von

```
void enlargeTree (PhyObjItem* objItem, direction oppDir)
```

solange rekursiv entgegengesetzt der Richtung `oppDir` verdoppelt, bis sie das Objekt wieder umfaßt. Dazu müssen neue Wurzelknoten vor der ursprünglichen Wurzel, mitsamt leerer Nachbarknoten für diese, eingefügt werden. Das Objekt wird schließlich bei der neuen Wurzel eingetragen.

- Im zweiten Schritt werden durch

```
void moveUpObjects(TreeItem*)
```

alle Objekte in der Hierarchie nach unten bewegt, die sich in eine Raumzelle hineinbewegt haben.

Auch hierbei hilft die Richtungsmarkierung: Wurde ein Knoten beispielsweise in `LEFT` und `RIGHT` aufgeteilt, so muß für ein Objekt, das sich innerhalb der Raumzelle dieses Knotens befindet nur noch überprüft werden, ob es völlig links oder völlig rechts der Trennungsgrenze liegt, um es dann zu einem Kind-Knoten zu verschieben.

Bei Blattknoten wird noch überprüft, ob diese genügend Objekte beinhalten, um durch `splitCell` weiter aufgeteilt zu werden.

Schließlich überprüft die Funktion

```
void shrinkTree()
```

noch rekursiv, ob die Szene sich völlig in einem Teilbaum befindet. Die Wurzel dieses Teilbaums wird dann zur neuen Wurzel und die leeren Knoten darüber werden mitsamt der leeren Nachbarknoten entfernt.

6.5 Multiple Dispatching

In der objektorientierten Programmierung werden die Datentypen Klassen (*class*) genannt. Diese enthalten neben den Daten selbst (*data-members*), auch die Methoden *member-functions*, die zu diesem Datentyp gehören. Der Aufruf einer solchen Methode erfolgt dann in Verbindung mit einer Instanz dieser Klasse oder - bei *static member-functions* - zusammen mit einer expliziten Angabe der Klasse. So werden gleichnamige Operationen für verschiedene Datentypen und somit auch Infixnotation mit einfachen Operatoren ermöglicht.

Ein weiterer Vorteil ergibt sich durch die Möglichkeit Klassen abzuleiten. Dabei können abgeleitete Datentypen Eigenschaften (*member-functions*) von ihrer Basisklasse erben.

Die im Rahmen dieser Arbeit benötigten Kollisionserkennungs und -reaktionsroutinen lassen sich jedoch nicht einem einzelnen Objekttyp zuordnen, sondern sie betreffen immer gleich zwei Objekttypen.

Multiple-Dispatching-Verfahren erlauben den Aufruf einer Funktion, abhängig vom Datentyp mehrerer ihrer Argumente. Zusätzlich ermöglichen sie abgeleiteten Klassen, die Eigenschaften ihrer Basisklasse zu erben.

Zur Realisierung ist zunächst die Erkennung der Datentypen der Funktionsargumente zu ermöglichen. Diese sind alle von einer Basisklasse abgeleitet, mit deren Hilfe es möglich ist, Multiple-Dispatching-Funktionen zu definieren. Beispiel:

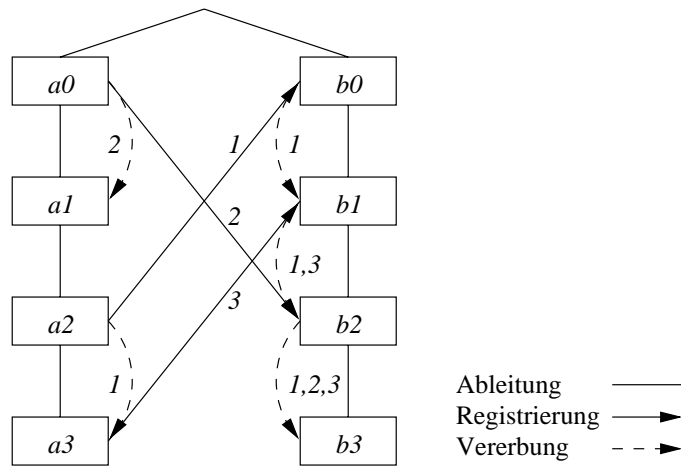


Abbildung 11: Multiple Dispatching

```
typedef void (*multipleDispatchingFunction) (t_BaseClass1* arg1,
                                             t_BaseClass2* arg2)
```

Da C++ keine Typerkennung zur Laufzeit unterstützt, muß diese zusätzlich implementiert werden. Zu diesem Zweck wird jedes Objekt mit einer ID versehen. Zusätzlich muss die Ableitungshierarchie protokolliert werden, um Vererbung zu ermöglichen.

Mit der Klasse `t_Rti` [Fis95] steht im Rahmen des MRT bereits eine solche Funktionalität zur Verfügung. Diese kann jedoch in diesem Fall nicht genutzt werden, da für eine schnelle Funktionsauswahl kleine Integer-IDs benötigt werden, die nicht größer als die Zahl der implementierten physikalischen Objekte sind.

Die Auswahl ererbter Funktionen ist nicht immer eindeutig. Es ist zum Beispiel denkbar, daß für zwei Argumente keine Funktion bereitsteht, wohl aber eine für jedes Argument in Kombination mit der unmittelbaren Basisklasse des anderen.

Existiert eine Funktion für den Fall, daß alle Argumente vom Typ der Basisklasse selbst sind, so kann durch Vererbung immer eine Funktion für beliebige Kombinationen abgeleiteter Klassen ausgewählt werden.

Beim Multiple-Dispatching sind neben der Auswahl einer geeigneten Funktion für mehrere Datentypen noch einige Verwaltungsaufgaben für die Klassen und die Funktionen zu erledigen. Die Funktionsauswahl ist sehr zeitkritisch, da sie ständig während der Laufzeit erfolgt. Es empfiehlt sich deshalb bei der Verwaltung von Klassen und Funktionen so vorzugehen, daß die Auswahl nur noch aus der Bestimmung der Datentypen (ID's) und dem Aufruf der entsprechenden Funktion aus einem Funktionenarray besteht.

6.5.1 `t_CollisionDispatching`

Zur Realisierung der Kollisionserkennung und -reaktionsroutinen wurde ein Multiple Dispatching mit zwei vertauschbaren Argumenten, die von `t_PhysicalObject` abgeleitet sind, realisiert.

Die Multiple-Dispatching-Funktionen haben den Typ

```
typedef void (*func) (const t_PhysicalObjectPtr& phyObject1,
                     const t_PhysicalObjectPtr& phyObject2);
```

Ein Objekttyp besorgt beim ersten Aufruf seines Konstruktors mittels

```
static t_Id registerClass (t_Id parentMDId)
```

eine ID, wobei er die ID der Klasse angibt, von der er abgeleitet ist (parent). Im Array `s_class` wird für jeden Objekttyp die ID parent, die ID einer abgeleiteten Klasse (child) und die ID einer weiteren von parent abgeleiteten Klasse gespeichert. Letztere (nextSibling) realisiert eine Liste aller von parent abgeleiteten Klassen.

Die Funktionen

```
virtual t_Id MDId()
static t_Id sMDId()
```

eines physikalischen Objektes liefern die ID's für die Funktionsauswahl.

Diese müssen jedoch zunächst mittels

```
void registerFunction (t_Id MDId1, t_Id MDId2, func function)
```

registriert werden.

Die Auswahl einer geeigneten Funktion besorgt

```
void execFunction (const t_PhysicalObjectPtr& phyObject1,
                  const t_PhysicalObjectPtr& phyObject2)
{
    t_Id id1 = phyObject1->MDId();
    t_Id id2 = phyObject2->MDId();
    if (v_dispatchArray[id1][id2] != NULL) {
        v_dispatchArray[id1][id2](phyObject1, phyObject2);
    }
    else {
        v_dispatchArray[id2][id1](phyObject2, phyObject1);
    }
}
```

Diese Funktion ist deshalb sehr schnell, weil sie davon ausgeht, daß die auszuwählende Funktion bereits im array `v_dispatchArray` indizierbar durch die ID's abgelegt ist. Ist ein Eintrag NULL, so wird die Existenz des entsprechenden Eintrags mit vertauschten Argumenten vorausgesetzt.

`registerClass` sorgt zu diesem Zweck dafür, daß alle Einträge von parent auch in die `v_dispatchArray`-Zellen der neuen Klasse übertragen werden.

`registerFunction` trägt die neue Funktion nicht nur für die beiden Argumenttypen, sondern auch rekursiv für deren abgeleitete Klassen ein, sofern dort nicht schon speziellere Funktionen registriert sind. Dies wird erkannt, wenn die dort eingetragene Funktion von der der Basis-klasse abweicht.

Dieses Verfahren hat den Nachteil, daß die Zahl der registrierten Klassen durch die Größe des Arrays begrenzt ist. In diesem Fall ist dies jedoch gerechtfertigt, da die Zahl der implementierten physikalischen Objekte überschaubar ist und bleibt.

Bei uneindeutigen Fällen wird die Funktion ausgeführt, die für die Basisklasse registriert wurde, die in der Hierarchie am nächsten steht. Eine einmal registrierte Funktion kann nur durch eine solche überschrieben werden, deren Argumenttypen beide näher oder gleichweit in der Hierarchie sind wie die ursprünglichen.

Die Abbildungen 11 und 12 zeigen ein Beispiel: Nachdem Funktion 1 für die Klassen a_2, b_0 registriert wurde, wird diese an die Kombinationen a_2, b_1 bis a_2, b_3 und a_3, b_0 bis a_3, b_3 vererbt.

Die anschließende Registrierung der Funktion 2 für a_0, b_2 wird nur bis a_1 vererbt. Weiter nicht, da die Einträge für a_0, b_2 und a_2, b_2 verschieden sind.

Funktion 3 hingegen kann Funktion 1 überschreiben.

	<i>a0</i>	<i>a1</i>	<i>a2</i>	<i>a3</i>
<i>b0</i>	0	0	1	1
<i>b1</i>	0	0	1	3
<i>b2</i>	2	2	1	3
<i>b3</i>	2	2	1	3

Abbildung 12: Uneindeutige Fälle

6.6 Erzeugung physikalischer Objekte

Die Hauptgründe für die Wahl von C++ für die Entwicklung des MRT sind Plattformunabhängigkeit und Erweiterbarkeit. Letztere wird durch die objektorientierte Programmierung erreicht: Dem MRT werden neue Objekte hinzugefügt, indem diese von entsprechenden Basisklassen abgeleitet werden. Anschließend ist noch die Syntax des Parsers zu erweitern, mit dessen Hilfe die Szenenbeschreibungsdateien interpretiert werden.

Genauso verhält es sich bei den physikalischen Objekten. Diese werden von der Basisklasse `t_PhysicalObject` oder einem anderen bereits abgeleiteten physikalischen Objekt abgeleitet.

Handelt es sich um ein sichtbares Objekt, so muß dieses zunächst ohne physikalische Eigenschaften als `t_SurfaceObject` zur Verfügung stehen. Das physikalische Objekt erzeugt dann in seinem Konstruktor eine Instanz dieses Objektes und übergibt diese an die Basisklasse `t_RefObject`

Das Objekt funktioniert dann bereits. Es hat jedoch - von seinem Aussehen abgesehen - die Eigenschaften einer starren Kugel.

Sollen Kollisionen extakt an der Objektoberfläche und nicht an der Oberfläche der Hüllkugel stattfinden, so sind entsprechende Kollisionsroutinen für die gewünschten anderen physikalischen Objekttypen zu implementieren. Diese werden dann von `t_CollisionDispatching` registriert. Da `t_CollisionDispatching` Vererbung unterstützt, können unter Umständen mehrere voneinander abgeleitete Objekttypen durch eine Multiple-Dispatching-Funktion abgedeckt werden.

Die generische Kollisionsreaktionsroutine, die bereits durch `t_PhysicalObject` registriert wird, behandelt bereits beliebige starre Körper. Sie muß also nur überladen werden, wenn andere Kollisionsreaktionen erwünscht sind.

Durch Überladung der `physicalFunction` können die physikalischen Eigenschaften geändert werden. Der Phantasie sind hierbei keine Grenzen gesetzt.

6.6.1 `t_RigidSphere`

Da das Ausgangsverhalten der von `t_PhysicalObject` abgeleiteten Objekte bereits das einer Kugel ist, muß zur Realisierung der starren Kugel fast nichts mehr implementiert werden. Es muß lediglich die Funktion `calcBSphere` überladen werden, da diese sonst eine Hüllkugel berechnet, die den die Kugel umgebenden hauptachsenparallelen Quader einschließt. Stattdessen werden einfach Mittelpunkt und Radius der Kugel selbst verwendet.

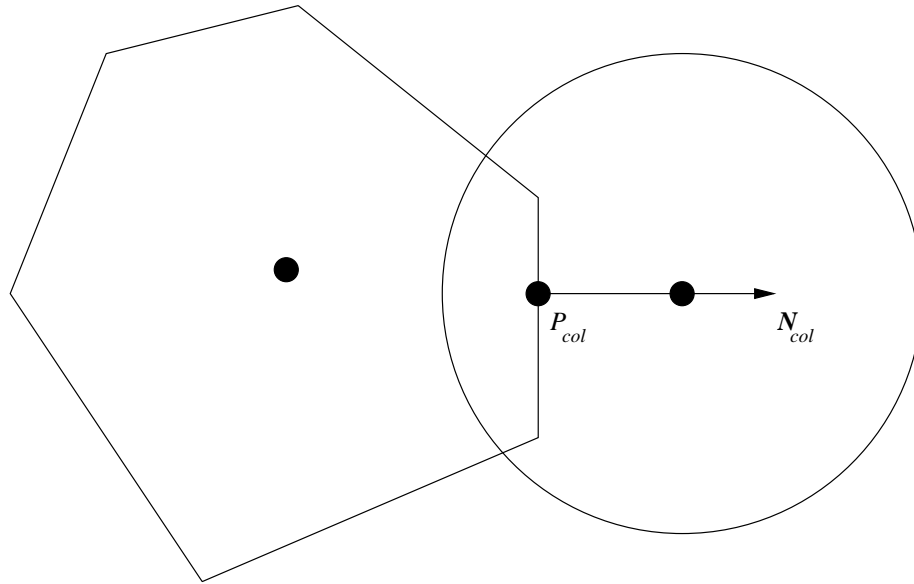


Abbildung 13: Kollision von Polyeder und Kugel

6.6.2 `t_RigidPolyhedron`

Diese Klasse kann als Basisklasse für Polyeder verwendet werden. Dazu ist eine Repräsentation des Objektes durch eine `t_RPRep`-Struktur zu erzeugen. Diese ist der bereits vorhandenen `t_BRep`-Struktur sehr ähnlich. Sie ist jedoch etwas einfacher aufgebaut.

Zur Verfügung stehen je ein Zeiger auf eine Liste der Eckpunkte, der Kanten und der Flächen. Bei der Transformation des Objektes wird diese Struktur mittransformiert, indem alle Eckpunkte bei einem Aufruf von `transform` oder `translate` transformiert werden. Die `t_BRep`-Struktur kann schon aus diesem Grund nicht verwendet werden, da sie für die approximative Anzeige benötigt wird und deshalb nicht verändert werden darf.

Die Erzeugung der `t_RPRep`-Struktur erfolgt noch manuell für jede abgeleitete Klasse. Versuchen mit einer automatischen Konvertierung von `t_BRep` nach `t_RPRep` könnte diese Klasse die Basisfunktionalität für beliebige `t_SurfaceObjects` stellen. Es sind jedoch der erhebliche Performanceunterschied gegenüber den Hüllkugeln und die ebenfalls nur angenäherte Genauigkeit der Objektform zu beachten.

Dieser Ansatz der Kollisionserkennung ist durchaus verbesserungswürdig. Die vorgestellten Verfahren sind nicht nur sehr rechenintensiv, sie werden im Laufe eines Schrittes auch häufig ausgeführt.

Eine Beschleunigungsmöglichkeit ergibt sich, indem auf der Datenstruktur der Polyeder selbst eine Hierarchie eingeführt wird, mit Hilfe derer das Testen einiger unnötiger Ecken, Kanten und Flächen vermieden werden kann.

Benachbarte Flächen könnten zu Blattknoten des Hierarchiebaumes zusammengefaßt werden. Benachbarte Knoten würden wiederum unter einem gemeinsamen Vorgängerknoten zusammengefaßt, wodurch die Hierarchie entstünde.

Innerhalb dieser Hierarchie würden unnötige Tests vermieden, weil ganze Teilbäume aufgrund von Hüllkörpertests als Kollisionsflächen ausgeschlossen werden könnten.

`polyhedronCollisionDetect`

Zur Überprüfung ob ein Schnitt mit der Hüllkugel eines anderen Objektes vorliegt, reicht es

nicht zu testen, ob einer der Eckpunkte innerhalb dieser Hüllkugel ist. Es muß auch der Fall berücksichtigt werden, daß eine Kante oder eine Fläche durch die Kugel verläuft, obwohl die Eckpunkte außerhalb der Kugel sind.

Als Kollisionspunkt ist der Punkt des Polyeders zu ermitteln, der dem Mittelpunkt der Kugel am nächsten ist. Zuerst wird geprüft, welcher Punkt der Kanten dem Mittelpunkt am nächsten ist.

Anschließend wird auf der Ebene jeder Fläche der Punkt P bestimmt, von dem aus die Normale den Mittelpunkt schneidet. Liegt P auf der Fläche selbst, so befindet er sich näher am Mittelpunkt als die Kanten (Abbildung 13).

Auf diese Weise werden außerdem Schnitte mit der Kugel erkannt, bei denen keine Kante beteiligt ist, die Kugel also in eine Fläche des Polyeders eindringt.

Die Kollisionsnormale ist durch die Kugel gegeben. Der Kollisionspunkt der Kugel wird entlang dieser auf die Oberfläche projiziert.

polyhedronPolyhedronCollisionDetect

Bei einer Kollision zweier Polyeder sind immer Kanten beteiligt. Dringt eine Spitze des einen Polyeders jedoch in eine Fläche des anderen ein, so sind nur die Kanten eines Objekts betroffen.

Der vorliegende Algorithmus bestimmt alle Schnittpunkte von Kanten mit Flächen. Als Kollisionspunkt wird dann der Mittelpunkt des kleinsten hauptachsenparallelen Quaders angenommen, der alle Schnittpunkte enthält.

Die Kollisionsnormale steht senkrecht auf der Kollisionsfläche, die den Winkel der Kollisionsvektoren halbiert.

6.6.3 t_RigidQuadrangle

Dieses Objekt hätte abgeleitet von `t_RigidPolyhedron` implementiert werden können. Es ist jedoch älter und enthält einige Vorarbeiten dazu.

Im Grunde ist es besser ein Objekt für sich zu implementieren, anstatt es von einer allgemeineren Klasse abzuleiten, da so seine Besonderheiten berücksichtigt werden können, was sich häufig beschleunigend auswirkt.

Man sollte jedoch abwägen, ob sich der zusätzliche Programmieraufwand lohnt, der ja bei später hinzukommenden Objekten immer wieder entsteht. Erkennt ein solches Kollisionen mit `t_RigidPolyhedron`, so erkennt es noch keine mit `t_RigidQuadrangle`.

6.7 Begrenzungsobjekte

Besonderes Augenmerk verdienen die Begrenzungsobjekte. Diese sind unbeweglich, unsichtbar und sie sollen verhindern, daß Objekte die Szene verlassen. Insbesondere müssen Typ 2-Durchdringungen erkannt werden, bei denen ein Objekt das andere völlig durchquert, was einem Verlassen der Szene gleich käme.

6.7.1 t_LimitPlane

Der erste Ansatz zur Lösung dieser Problemstellung hat sich nicht bewährt. `t_LimitPlane` ist eine unendliche Ebene, gegeben durch einen Punkt P und einen Normalenvektor N . Sie hat unendlich große Masse und reflektiert entsprechend alle Objekte, die einen Schnitt mit ihr haben. Objekte, die die Ebene durchquert haben, werden erkannt und zurückreflektiert.

limitPlaneCollisionDetect

Um zu erkennen, in welchem der beiden Halbräume sich ein Objekt vor dem letzten Zeitintervall befunden hat, muß die vorherige Position rekonstruiert werden. Alle Objekte müssen untersucht

werden, auch die, die keine Durchdringung mit der Ebene selbst haben. Objekte, die die Ebene durchquert haben müssen in einer Liste gespeichert werden, damit sie später nicht erneut reflektiert sondern, zurückgelassen werden.

Es ist zwar selten, daß Objekte die Ebene völlig durchqueren, dennoch handelt es sich um einen Komplexitätsanstieg durch die Liste. Außerdem ist es nicht möglich, die Zahl der Kollisionen durch Hüllkörper oder andere Beschleunigungsverfahren zu reduzieren.

Der Vorteil dieses Objektes ist, daß es geeignet ist, aneinanderstoßende Szenen voneinander zu trennen.

6.7.2 `t_LimitHalfSpace`

In der Regel befinden sich alle Objekte einer Szene auf der selben Seite der Begrenzungsebene. Folglich hat jedes Objekt, das sich auf der anderen Seite befindet, die Ebene durchquert und muß zurückgelassen werden. Der gesamte unerwünschte Aufwand der `t_LimitPlane` entfällt wegen dieser einen Annahme. Der Begrenzungshalbraum ist ebenfalls durch P und N gegeben.

`halfSpaceCollisionDetect`

Wird diese Funktion nicht überladen, so werden, wie in `t_PhysicalObjekt` üblich, alle anderen Objekte als Kugeln mit Mittelpunkt C und Radius r betrachtet. Um eine Kollision zu erkennen ist lediglich der Vektor $D = C \Leftrightarrow P$ zu bestimmen. Zeigt dieser in den selben Halbraum wie N und ist er projiziert auf N länger als r , so liegt keine Kollision vor.

`polyhedronHalfSpaceCollisionDetect` / `quadrangleHalfSpaceCollisionDetect`

Beim Polyeder liegt keine Durchdringung vor, wenn alle Punkte im richtigen Halbraum liegen. Kollisionspunkt ist immer ein Eckpunkt und die Kollisionsnormale ist durch den Halbraum gegeben.

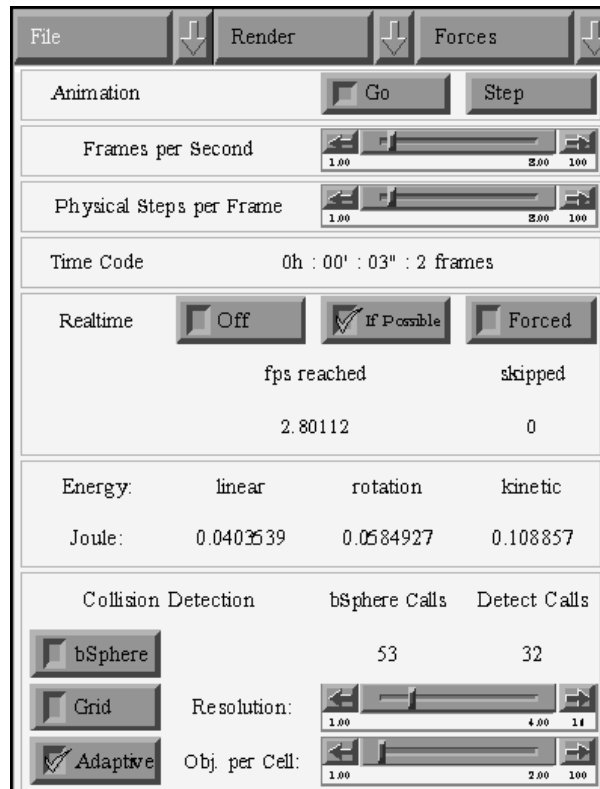
6.7.3 `t_LimitBox` / `t_LimitQuadrangle`

`t_LimitHalfSpace` hat den Nachteil, daß wegen der unendlich großen Ausdehnung keine Beschleunigungstechniken greifen. `t_LimitQuadrangle` ist von `t_LimitHalfSpace` abgeleitet und fügt als einzige Eigenschaft einen hauptachsenparallelen Hüllkörper hinzu. Kollisionen außerhalb dieses Hüllkörpers werden also unter Umständen nicht erkannt.

`t_LimitQuadrangle` ist für den Einsatz durch `t_LimitBox` gedacht. Diese setzt sechs Begrenzungsquadrate zu einem Begrenzungsquader zusammen. Kollisionen außerhalb der Hüllkörper oder mit den Kanten der Begrenzungsquadrate sind somit ausgeschlossen.

Da die Kollisionseigenschaften von `t_LimitHalfSpace` geerbt werden, müssen keine zusätzlichen Kollisionserkennungsfunktionen implementiert werden.

Die durch `t_LimitBox` erzeugten Begrenzungsquadrate haben Hauptachsenlage. Sie sind deshalb optimal durch ihre Hüllkörper umgeben und somit gut für den Einsatz von Beschleunigungstechniken geeignet.

Abbildung 14: *mrtphys*

7 *mrtphys*

In der Computergraphikgruppe der Abteilung III des Instituts für Informatik der Universität Bonn wird neben dem MRT auch die Graphische Benutzerschnittstelle *penguin* entwickelt. Um Plattformunabhängigkeit und Erweiterbarkeit zu erreichen wurde auch diese objektorientiert unter Verwendung von C++ realisiert.

mrtphys, das Programm, das als Bestandteil dieser Diplomarbeit zur Erprobung der physikalischen Objekte dient, nutzt *penguin*, da auch das MRT, einschließlich der hier vorgestellten physikalischen Erweiterung, einen besonderen Schwerpunkt auf Plattformunabhängigkeit und Erweiterbarkeit legt.

penguin bietet die Möglichkeit, ein Fenster als CGI-Ausgabegerät ⁸ zu öffnen. Da mittels *cgi-3D* (ebenfalls in dieser Abteilung entwickelt) Ausgaben des MRT möglich sind, wird die Animation in einem solchen Fenster angezeigt.

In diesem Fenster kann außerdem mit der Maus die Position des Beobachters der Szene manipuliert werden:

Bewegt man die Maus mit gedrückter linker Maustaste, so bewegt sich der Betrachter auf einer Kugel, die den *lookPoint* umgibt. Seine Entfernung zu diesem Punkt ändert sich nicht.

Bei gedrückter mittlerer Maustaste wird der *lookPoint* verschoben.

Bewegt man die Maus mit gedrückter rechter Maustaste vertikal, so nähert man sich dem *lookPoint*, bzw. man entfernt sich. Bewegt man sie horizontal so manipuliert man den *upVector*, der immer nach oben zeigt. Das Bild dreht sich.

⁸CGI = Computer Graphics Interface

7.1 Interaktive Steuerung der Animation

Ein weiteres Fenster ermöglicht die Steuerung der Animation (Abbildung 14). Alle Eingaben werden sofort berücksichtigt, was Eingriffe in die laufende Animation ermöglicht.

Im Untermenü *Render* werden Einstellungen vorgenommen, die die Visualisierung der Szene betreffen: *Shading* betrifft die Interpolation der einzelnen Dreiecke, *Illumination* die verwendete Beleuchtungsgleichung und *Rendering* das HLHSR-Verfahren (Entfernung nicht sichtbarer Kanten und Flächen). Ferner kann die Qualität der Triangulierung eingestellt werden.

Das Untermenü *Forces* dient zur Aktivierung des Partikelsystems. Vorerst sind nur Erdanziehungskraft und Viskosität einstellbar.

Einstellungen, die die Animation betreffen, werden im Hauptmenü selbst vorgenommen:

Animation dient zum Starten und Stoppen der Animation (*Go*) oder zur Erzeugung von Einzelschritten (*Step*).

Frames per Second legt die Zahl der Einzelbilder pro Sekunde fest. (Die Sekunde wird in allen physikalischen Funktionen als Zeiteinheit zugrunde gelegt.)

Zur Reduktion der physikalischen Diskretisierungsfehler werden die Einzelbilder noch in *Physical Steps per Frame* Schritte aufgeteilt.

Time Code zeigt die Animationszeit im Format:

Stundenh : Minuten' : Sekunden'' : Einzelbilder frames.

Es sind drei *Realtime*-Modi wählbar: *Off* bedeutet, daß die Animation so schnell wie möglich läuft. *fps reached* zeigt, wieviele Einzelbilder pro (echter) Sekunde berechnet werden.

If Possible bedeutet, daß die Animation wenn nötig gebremst wird, damit die Animationszeit nicht schneller verstreicht als die echte Zeit. Ob dies nötig ist, kann ebenfalls bei *fps reached* abgelesen werden.

Im *Forced*-Modus wird erzwungen, daß Animationszeit und echte Zeit annähernd synchron sind. Dazu werden, nachdem ein Einzelbild zuviel Zeit benötigt hat, folgende Einzelbilder zusammengefaßt und insgesamt nur in *Physical Steps per Frame* Schritte aufgeteilt. Dies kann zu deutlichen Diskretisierungsfehlern führen. Die Zahl der übersprungenen Einzelbilder wird unter *skipped* angezeigt.

Energy zeigt die kinetische Energie in der Szene an. Dabei sind bei jeder Kollision Überträge zwischen linearer kinetischer Energie (*linear*) und Rotationsenergie (*rotation*) zu beobachten. Wegen des Energieerhaltungssatzes bleibt die gesamte kinetische Energie (*kinetic*) bei starren Körpern erhalten. Da dieser nur im geschlossenen System gilt, ändert sie sich bei Aktivierung des Partikelsystems.

Schließlich läßt sich noch der Beschleunigungsalgorithmus für die Kollisionserkennung wählen. Da aufgrund der Vorgabe durch `t_PhysicalObject` Hüllkugeln immer berücksichtigt werden, wird sowohl die Zahl der Hüllkugeltests (*bSphere Calls*) als auch die Zahl der speziellen Tests (*detect Calls*) angezeigt. Die speziellen Tests werden immer aufgerufen, wenn nach einem Hüllkugeltest eine Kollision nicht ausgeschlossen ist.

Grid aktiviert ein reguläres Gitter, dessen Auflösung einstellbar ist.

Adaptive aktiviert die adaptive Raumunterteilung. Einstellbar ist, wieviele Objekte höchstens in einer Raumzelle sein sollen.

7.2 Aufbau des Programms

mrtphys ist modular aufgebaut. Im Hauptprogramm werden die *penguin*-Dialoge initialisiert, das Ausgabefenster (`t_SceneWindow`) wird geöffnet und die physikalisch basierte Animation wird vorbereitet.

Zum Schluß wird die Kontrolle an *penguin* übergeben. Eingaben des Benutzers (*events*) werden von *penguin* an die entsprechenden Aktionen (*actions*) weitergeleitet.

Mit Hilfe eines *idle-events*, daß immer erzeugt wird, wenn das Programm im Leerlauf ist, werden die Aktionen der Animation aufgerufen.

Die Header der *mrtphys*-spezifischen Klassen sind in Anhang B aufgeführt.

7.2.1 *penguin*-basierte Dialogführung

Die Klassen `t_AnimationDialog`, `t_RenderDialog` und `t_ForcesDialog` stehen für die drei Menüs, wobei `t_AnimationDialog` das Hauptmenü ist. Das *File*-Menü ist Teil des Hauptprogramms. Die Layouts der Menüs sind in *mrtphys.pen* definiert.

7.2.2 `t_SceneWindow`

Die *penguin*-Klasse `t_CgiGlyph` öffnet ein Fenster als CGI-Ausgabegerät. `t_SceneWindow` ist von `t_CgiGlyph` abgeleitet. Es wertet zusätzlich die Parameter des Programmaufrufs im Sinne des MRT aus, liest die Szene aus der Szenenbeschreibungsdatei ein und stellt diese dar.

Die Darstellung hängt von den Parametern aus der Kommandozeile, von der Szenenbeschreibung und von den Einstellungen im Menü *Render* ab. Es werden nur die approximativen Darstellungsmodi des MRT unterstützt. Raytracing und Radiosity sind für Echtzeitanimationen noch uninteressant.

Die Aktionen zur Modifikation der Betrachterperspektive werden bereitgestellt und die Mauseingaben in das Szene-Fenster werden an diese weitergeleitet.

7.2.3 `t_PhysicalAnimation`

Die Klasse `t_PhysicalAnimation` stellt die Aktionen bereit, an die das *idle-event* weitergeleitet wird.

Tritt ein solches auf, so ruft `nextFrame` die Funktion `physicalFrame` auf, falls dies aufgrund des eingestellten *Realtime*-Modus und der verstrichenen Zeit nötig ist.

Ferner werden die im Animationsmenü angezeigten Werte *Time Code*, *fps reached* und *skipped* geliefert.

8 Ausblick

Die vorliegende Arbeit ist als grundlegend für die Erweiterung des MRT in physikalisch basierter Richtung zu betrachten. Es wurde ein Konzept vorgestellt, mit dem sich bewegliche starre Objekte realisieren lassen.

Die vorgestellten Beschleunigungsverfahren der Kollisionserkennung sind vielversprechend - vor allem dann, wenn bessere 3D-Graphik-Hardware den Stellenwert der Kollisionserkennung gegenüber dem Rendering erhöht, weil komplexere Szenen in animationsfähiger Geschwindigkeit dargestellt werden können.

Doch es sind auch viele Punkte aufgefallen, an denen eine Weiterentwicklung lohnend erscheint. Zu nennen wären z.B. die völlig ausgesparten statischen Eigenschaften von Objekten, die zusammen mit dem bisher nur rudimentär ausgeprägten Partikelsystem Erweiterungen ermöglichen könnten, bei denen Objekte über die hier vorgestellten Kollisionen hinaus interagieren.

Auch die Einschränkung auf starre Objekte ist im Rahmen dieses Konzeptes nicht zwingend. Plastische und elastische Objekte sind denkbar, ebenso wie solche, die während der Animation entstehen oder vergehen.

Fortschritte bei der Grafikhardware würden ebenfalls die angedachten Beschleunigungsmöglichkeiten für die Kollisionserkennung mit Polyedern interessanter machen. Würden Objekte durch sehr viele kleine Einzelflächen in zumutbarer Zeit angezeigt, so würde das nicht nur die Genauigkeit der Approximation und damit auch der Kollisionserkennung erhöhen, es würden durch die vorgeschlagene Hierarchie auch die meisten der unzähligen Kollisionstests von Einzelflächen vermieden.

Schließlich bleibt noch zu hoffen, daß eine Vielzahl neu zu implementierender physikalischer Objekte die derzeitigen virtuellen Welten von Kugeln und ähnlich simplen Objekten bereichern wird.

A Klassenreferenz

In der folgenden Klassenübersicht sind alle `public`- und `protected-members` der Klassen aufgeführt. Einige `private-members` sind erwähnt, wenn es dem besseren Verständnis der Klasse dient.

Die Klassen, die ausschließlich *mrtphys* betreffen, sind in Anhang B aufgeführt.

A.1 t_BSphere

```
class t_BSphere
    Hüllkugel

public:
t_BSphere()
    Erzeugt leere Hüllkugel.

t_BSphere (const t_3DVector& center, t_Real radius)
    Erzeugt Hüllkugel. Bei negativem Radius wird eine unendlich große Hüllkugel erzeugt.

t_Boolean checkIntersect (const t_BSphere& bSphere2) const
    Test auf Schnitt von this mit bSphere2.

t_Boolean empty() const
    Test auf  $0 \leq \text{radius} < \text{epsilon}$ .

void        center (const t_3DVector& center)
void        radius (t_Real radius)
t_3DVector center () const
t_Real     radius () const
    Setzen und Abfragen der Hüllkugelspezifikation.
```

A.2 t_CDAdaptiveGrid

```
class t_CDAdaptiveGrid : public t_CollisionDetection
    Kollisionserkennungsbeschleunigung unter Einsatz einer adaptiven Raumunterteilung.

public:
t_CDAdaptiveGrid (int maxObjectsPerCell, int maxDepth = 16)
    Erzeugt Raumunterteilung mit maximal maxObjectsPerCell Objekten pro Element und einer Binärbaumtiefe von höchstens maxDepth.

virtual ~t_CDAdaptiveGrid()
    Löscht Binärbaum.

virtual void detectAndResponse()
    Startet Kollisionserkennung und -reaktion.
```

```
private:
enum direction {NO,LEFT,RIGHT,BOTTOM,TOP,BACK,FRONT}
    Unterteilungs- bzw. Bewegungsrichtungen
```

```
class PhyObjItem
public
    t_PhysicalObjectPtr phyObject
    PhyObjItem*         pre
    PhyObjItem*         next
    Objektlistenelement
```

```
class TreeItem
public:
    int         num
    int         depth
    PhyObjItem* first
    t_BVol      bVol
    direction   dir
    TreeItem*   parent
    TreeItem*   child1
    TreeItem*   child2
```

Binärbaumknoten, der in `dir` speichert, durch welche Unterteilung er entstanden ist.

```
void splitCell (TreeItem* item)
```

Unterteilt die Raumzelle entsprechend ihrer größten Ausdehnung.

```
void joinCell (TreeItem* item)
```

Vereinigt die Raumzelle mit der Nachbarzelle, von der sie abgeteilt wurde.

```
void moveUpObjects (TreeItem* item)
```

Verschiebt rekursiv alle Objekte, die ihre Raumzelle verlassen haben, in der Hierarchie so weit wie nötig nach oben.

```
void moveDownObjects (TreeItem* item)
```

Verschiebt rekursiv alle Objekte, die sich in eine Raumzelle hineinbewegt haben, in der Hierarchie so weit wie möglich nach unten.

```
void enlargeTree (PhyObjItem* objItem, direction oppDir)
```

Erweitert die Hierarchie an der Wurzel entgegengesetzt der Richtung `oppDir` und wiederholt dies rekursiv, bis `objItem` in die entsprechende Raumzelle paßt.

```
void shrinkTree()
```

Macht den obersten benötigten Knoten des Binärbaums zur Wurzel.

```
void deleteTree(TreeItem* item)
```

Löscht Binärbaum rekursiv (wird von `~t_CDAdaptiveGrid` aufgerufen).

```
PhyObjItem* moveItem (PhyObjItem* objItem,
                      TreeItem*   src,
                      TreeItem*   dest)
```

Löscht objItem aus der Raumzelle src und fügt es bei dest ein.
Gibt objItem->next zurück.

```
static
direction moved (const t_BVol& bVol1, const t_BVol& bVol2)
```

Gibt die entgegengesetzte Richtung der Richtung zurück, in die bVol1 bVol2 verläßt. Die Gleichheit zweier Seiten reicht bereits aus. epsilon wird berücksichtigt.

```
static
t_Boolean movedLeft (const t_BVol& bVol1, const t_BVol& bVol2)
```

```
static
t_Boolean movedRight (const t_BVol& bVol1, const t_BVol& bVol2)
```

```
static
t_Boolean movedBottom (const t_BVol& bVol1, const t_BVol& bVol2)
```

```
static
t_Boolean movedTop (const t_BVol& bVol1, const t_BVol& bVol2)
```

```
static
t_Boolean movedBack (const t_BVol& bVol1, const t_BVol& bVol2)
```

```
static
t_Boolean movedFront (const t_BVol& bVol1, const t_BVol& bVol2)
```

Wie moved, es wird jedoch nur eine Richtung überprüft.

```
void detectAndResponse (PhyObjItem* objItem1,
                       PhyObjItem* objItem2)
```

```
void detectAndResponse (PhyObjItem* objItem,
                       TreeItem*   item)
```

```
void detectAndResponse (TreeItem*   item)
```

Rekursives Durchlaufen des Binärbaumes zur Kollisionserkennung und -reaktion.

A.3 t_CDRegularGrid

```
class t_CDRegularGrid : public t_CollisionDetection
```

Kollisionserkennungsbeschleunigung unter Einsatz eines regulären Gitters.

```
public:
```

```
t_CDRegularGrid(int gridResolution)
```

Teilt die gesamte physikalische Szene in ein dreidimensionales Gitter mit Auflösung gridResolution ein.

```
virtual void detectAndResponse()
```

Startet Kollisionserkennung und -reaktion.

```
private:
```

```
void setToGrid (const t_PhysicalObjectPtr& phyObject)
```

Trägt phyObject in alle Gitterzellen ein, die einen nichtleeren Schnitt mit seinem Hüllkörper (boundingVolume) haben.

```

class Item
public:
    t_PhysicalObjectPtr  phyObject
    int                  xCells
    int                  yCells
    int                  zCells
    t_Bool               tag
    Item*                pre
    Item*                next

```

Objektlistenelement mit der Anzahl der Zellen die das Objekt einnimmt.

```

Item* v_grid [CD_MAX_REGULAR_GRID_RES]
           [CD_MAX_REGULAR_GRID_RES]
           [CD_MAX_REGULAR_GRID_RES]

```

Objektlistenarray

A.4 t_CollisionDetection

```

class t_CollisionDetection

```

Basisklasse zur Kollisionserkennung. Beschleunigungsverfahren können als abgeleitete Klasse implementiert werden.

```

public:

```

```

virtual void detectAndResponse()

```

Kollisionserkennung und -reaktion aller physikalischer Objekte.

Wird von `t_PhysicalObject::physicalFrame` aufgerufen.

Wird diese Funktion nicht überladen, so wird zur Kollisionserkennung jedes Objekt mit jedem getestet. Ein Hüllkugelttest verhindert unnötige Aufrufe von Kollisionserkennungsroutinen. Durch Überladen können Beschleunigungsverfahren realisiert werden.

```

virtual t_Bool checkIntersections()

```

Überprüft die Existenz von Durchdringungen. Schnitte zweier Objekte unendlicher Masse werden ignoriert.

```

static

```

```

void detectAndResponse (const t_PhysicalObjectPtr& phyObject1,
                       const t_PhysicalObjectPtr& phyObject2)

```

Kollisionserkennung und -reaktion zweier Objekte.

```

static void detect (const t_PhysicalObjectPtr& phyObject1,
                   const t_PhysicalObjectPtr& phyObject2)

```

Ruft die registrierte Kollisionserkennungsroutine auf. Wird eine Kollision erkannt, so muß diese True an `detected` zurückgeben.

```

static void response (const t_PhysicalObjectPtr& phyObject1,
                     const t_PhysicalObjectPtr& phyObject2)

```

Ruft die registrierte Kollisionsreaktionsroutine auf.

```

static t_Bool detected()
static void detected (t_Bool detected)
    Setzen und Abfragen des letzten Kollisionstestergebnisses.

static t_Bool crossed()
static void crossed (t_Bool crossed)
    Kollisionserkennungsroutinen können an dieser Stelle vermerken, daß sich zwei Ob-
    jekte durcheinander hindurch bewegt haben.

static t_Id registerClass (t_Id parentMDId)
    Registriert eine Klasse, die von der Klasse parentMDId abgeleitet wurde, für das
    Multiple-Dispatching und gibt deren neue ID zurück.

static void registerDetectFunction (t_Id class1, t_Id class2,
    t_CollisionDispatch::func detectFunction)
    Registriert detectFunction für das Multiple-Dispatching.

static void registerResponseFunction (t_Id class1, t_Id class2,
    t_CollisionDispatch::func responseFunction)
    Registriert responseFunction für das Multiple-Dispatching.

static int bSphereCalls()
static int detectCalls()
    Anzahl der im letzten Step gemachten Tests.

protected:
static unsigned long s_bSphereCalls
static unsigned long s_detectCalls
    Die Testzählvariablen, die von abgeleiteten Klassen hochgezählt und zu Beginn eines
    Steps zurückgesetzt werden müssen.

```

A.5 t_CollisionDispatch

```

class t_CollisionDispatch
    Multiple-Dispatching mit zwei vertauschbaren t_PhysicalObject-Argumenten.
    Die Hierarchie der registrierten Klassen ist static. Verschiedene Instanzen können
    Kollisionserkennungs- bzw. -reaktionsroutinen verwalten.

public:
t_CollisionDispatch()
    Konstruktor

static t_Id registerClass (t_Id parentMDId)
    Registriert eine Klasse, die von der Klasse parentMDId abgeleitet wurde, für das
    Multiple-Dispatching und gibt deren neue ID zurück. t_PhysicalObject hat die
    ID 0.
    Vor der ersten Klassenregistrierung müssen alle t_CollisionDispatch-Instan-
    zen existieren.
    Bei der Registrierung erbt die Klasse alle registrierten Funktionen von parentMDId.

```

```
typedef void (*func) (const t_PhysicalObjectPtr& phyObject1,
                    const t_PhysicalObjectPtr& phyObject2)
```

Typ einer Multiple-Dispatching Funktion.

```
void registerFunction (t_Id MDId1, t_Id MDId2, func function)
```

Registriert die Funktion `func` für die Klassen `MDId1` und `MDId2` sowie für deren abgeleitete Klassen, sofern dort noch keine spezielleren Funktionen registriert sind.

```
void execFunction (const t_PhysicalObjectPtr& phyObject1,
                 const t_PhysicalObjectPtr& phyObject2)
```

Führt eine Multiple-Dispatching Funktion für die beiden Objekttypen aus.

```
private:
```

```
class item
```

```
public:
```

```
    t_Id parent
```

```
    t_Id firstChild
```

```
    t_Id nextSibling
```

Hierarchielistenelement

```
static item s_class [MAX_MDID]
```

Hierarchielisten-Array

```
func v_dispatchArray [MAX_MDID][MAX_MDID]
```

Multiple-Dispatching Funktionen-Array

A.6 t_EarthGravity

```
class t_EarthGravity : public t_ForceField
```

Erdanziehung, realisiert als `t_ForceField`.

```
public:
```

```
t_EarthGravity (t_Real g = 9.80665)
```

Initialisiert Erdanziehung mit Beschleunigungskonstante `g`.

```
void earthAcceleration (t_Real g)
```

```
t_Real earthAcceleration() const
```

Setzen und Abfragen der Erdbeschleunigungskonstante.

```
static t_Real earthAccConst()
```

Liefert Naturkonstante $g = 9.80665 \frac{\text{m}}{\text{s}^2}$

```
virtual t_3DVector acceleration (const t_3DVector& position,
                               t_Real mass) const
```

Liefert immer `t_3DVector(.0, -g, .0)`, unabhängig von Position und Masse. Dies entspricht dem Fallverhalten von Objekten auf der Erdoberfläche, unter Vernachlässigung der Luftreibung.

A.7 t_ForceField

```
class t_ForceField
```

Basisklasse zur Implementierung von Kraftfeldern im Sinne eines Partikelsystems.

```
public:
```

```
t_ForceField()
```

Konstruktor

```
virtual ~t_ForceField()
```

Entfernt Instanz aus interner Liste.

```
virtual
```

```
t_3DVector acceleration (const t_3DVector& position,
                        t_Real          mass) const = 0
```

Liegt an Punkt *position* die Kraft \mathbf{F} an, so wird gemäß des Aktionsprinzips die Beschleunigung $\mathbf{a} = \frac{\mathbf{F}}{m}$ zurückgegeben.

```
static t_3DVector allAcc (const t_3DVector& position,
                        t_Real          mass)
```

Summe der Beschleunigungen aller Instanzen an Punkt *position*.

A.8 t_Invisible

```
class t_Invisible : public t_SurfaceObject
```

Unsichtbares *t_SurfaceObject* (wird von Begrenzungsobjekten verwendet).

```
public:
```

```
t_Invisible()
```

Konstruktor

```
virtual t_Bool intersect (t-Ray& ray)
```

Liefert immer False

A.9 t_LimitBox

```
class t_LimitBox : public t_Invisible
```

Ein unsichtbarer Quader bestehend aus sechs Begrenzungsrechtecken in Hauptachsenlage.

```
public:
```

```
t_LimitBox (const t_3DVector& p1,
            const t_3DVector& p2,
            t_Real          friction)
```

Erzeugt sechs Instanzen vom Typ *t_LimitQuadrangle*.

A.10 t_LimitHalfSpace

```
class t_LimitHalfSpace : public t_PhysicalObject
```

Unsichtbarer Halbraum unendlicher Masse zur Szenenbegrenzung. Die Hüllkörper sind ebenfalls unendlich, was Kollisionserkennungsbeschleunigung erschwert.

```
public:
```

```
t_LimitHalfSpace (const t_3DVector&      point,
                  const t_3DVector&      normal,
                  t_Real                  friction,
                  const t_SurfaceShaderPtr& srfshd =
                                          (t_SurfaceShaderPtr)NULL)
```

Der Halbraum ist durch einen Punkt und den Normalenvektor gegeben.

```
virtual t_BVol boundingVolume() const
```

Liefert leeren Hüllkörper, was von der Kollisionserkennung als unendlich zu interpretieren ist.

```
virtual t_3DVector point() const
```

```
virtual t_3DVector normal() const
```

Die geometrische Spezifikation physikalischer Objekte muß public sein, um Kollisionserkennung zu ermöglichen.

```
virtual t_Id MDId()
```

```
static t_Id sMDId()
```

Liefert Multiple-Dispatching ID.

```
protected:
```

```
virtual void physicalFunction (t_Real timeStep)
```

Ohne Funktion (verhindert die Bewegung des Halbraumes).

```
virtual void calcBSphere()
```

Liefert unendlich große Hüllkugel (negativer Radius).

```
private:
```

```
static void halfSpaceCollisionDetect (
                                const t_PhysicalObjectPtr& halfSpace,
                                const t_PhysicalObjectPtr& phyObject)
```

Kollisionserkennungsroutine für Kollisionen mit den Hüllkugeln beliebiger Objekte. Wird bei t_CollisionDetection registriert.

A.11 t_LimitPlane

```
class t_LimitPlane : public t_PhysicalObject
```

Unsichtbare Ebene unendlich großer Masse zur Szenenbegrenzung. Objekte, die die Ebene durchquert haben werden zurückgelassen.

Der Hüllkörper ist ebenfalls unendlich, was Kollisionserkennungsbeschleunigung erschwert.

```
public:
t_LimitPlane (const t_3DVector&      point,
              const t_3DVector&      normal,
              t_Real                  friction,
              const t_SurfaceShaderPtr& srfshd =
                                      (t_SurfaceShaderPtr)NULL)
```

Die Ebene ist durch einen Punkt und ihre Normale gegeben.

```
t_Boolean inLetBackList      (const t_PhysicalObjectPtr& object)
void      addToNewLetBackList (const t_PhysicalObjectPtr& object)
```

Objekte, die die Ebene durchquert haben, werden in die letBackList aufgenommen, um eine erneute Kollisionserkennung bei der Rückkehr zu verhindern.

```
virtual t_BVol boundingVolume() const
```

Liefert leeren Hüllkörper, was von der Kollisionserkennung als unendlich zu interpretieren ist.

```
virtual t_3DVector point() const
virtual t_3DVector normal() const
```

Die geometrische Spezifikation physikalischer Objekte muß public sein, um Kollisionserkennung zu ermöglichen.

```
virtual t_Id MDId()
static t_Id sMDId()
```

Liefert Multiple-Dispatching ID.

```
protected:
```

```
virtual void physicalFunction (t_Real timeStep)
```

Ohne Funktion (verhindert die Bewegung der Ebene).

```
virtual void calcBSphere()
```

Liefert unendlich große Hüllkugel (negativer Radius).

```
class item
```

```
public:
```

```
t_PhysicalObjectPtr object
item*                next
```

Element der letBackList

```
item* v_letBackList
item* v_newLetBackList
```

Aktuelle und neue letBackList

```
void clearLetBackList()
```

Löscht letBackList

```
private:
```

```
static void limitPlaneCollisionDetect (
                                      const t_PhysicalObjectPtr& plane,
                                      const t_PhysicalObjectPtr& phyObject)
```

Kollisionserkennungsroutine für Kollisionen mit den Hüllkugeln beliebiger Objekte. Wird bei t_CollisionDetection registriert.

A.12 t_LimitQuadrangle

```
class t_LimitQuadrangle : public t_LimitHalfSpace
```

Unsichtbares Begrenzungsviereck unendlich großer Masse.

Identisch zu `t_LimitHalfSpace`, jedoch mit endlichen Hüllkörpern. Kollisionen der Ebene außerhalb des Hüllkörpers `boundingVolume` sollten vermieden werden. Wird von `t_LimitBox` verwendet.

```
public:
```

```
t_LimitQuadrangle (const t_3DVector&      p1,
                  const t_3DVector&      p2,
                  const t_3DVector&      p3,
                  const t_3DVector&      p4,
                  const t_3DVector&      normal,
                  t_Real                  friction,
                  const t_SurfaceShaderPtr& srfshd
                  = (t_SurfaceShaderPtr)NULL)
```

Konstruktor

```
virtual t_BVol boundingVolume() const
```

Kleinster Hüllkörper der `p1` bis `p4` umfaßt.

```
virtual t_3DVector p1() const
```

```
virtual t_3DVector p2() const
```

```
virtual t_3DVector p3() const
```

```
virtual t_3DVector p4() const
```

Die geometrische Spezifikation physikalischer Objekte muß `public` sein, um Kollisionserkennung zu ermöglichen.

```
virtual t_Id MDId()
```

```
static t_Id sMDId()
```

Liefert Multiple-Dispatching ID.

```
protected:
```

```
virtual void calcBSphere()
```

Kleinste Hüllkugel, die `p1` bis `p4` einschließt.

A.13 t_PhysicalObject

```
class t_PhysicalObject : public t_RefObject
```

Basisklasse aller physikalischer Objekte. Fügt geometrischen Objekten Masse, Geschwindigkeit, Rotation und Reibung hinzu.

Die Ableitung von `t_RefObject` erlaubt u.a. Translation und Rotation.

Abgeleitete Objekte müssen ihre geometrische Beschreibung `public` zur Verfügung stellen, um Kollisionserkennung zu ermöglichen.

```

public:
t_PhysicalObject (const t_3DVector&      massCenter,
                  t_Real                  mass,
                  const t_3DVector&      velocity,
                  const t_3DVector&      rotVelocity,
                  t_Real                  friction,
                  const t_SurfaceShaderPtr& srfshd
                  = (t_SurfaceShaderPtr)NULL)

```

Die Einheit der Masse ist kg. $mass < 0$ bedeutet unendliche Masse.

$velocity$ ist die Startgeschwindigkeit in $\frac{m}{s}$. Die Länge von $rotVelocity$ ist die Startwinkelgeschwindigkeit in $\frac{1}{s}$, seine Richtung die Rotationsachse. $velocity$ und $rotVelocity$ werden bei unendlicher Masse ignoriert, um Kollisionen unendlicher Massen zu verhindern.

$friction \in [0, 1]$ ist der Reibungskoeffizient. Je größer er ist desto „haftender“ ist die Fläche.

```

virtual ~t_PhysicalObject()

```

Löscht das Objekt aus der internen Liste.

```

t_Real      mass      () const
void        mass      (t_Real m)
t_3DVector  massCenter () const
void        massCenter (const t_3DVector& c)
t_3DVector  velocity   () const
void        velocity   (const t_3DVector& v)
t_3DVector  rotVelocity () const
void        rotVelocity (const t_3DVector& r)
t_Real      friction   () const
void        friction   (t_Real f)

```

Alle geometrischen und physikalischen Eigenschaften, die möglicherweise zur Kollisionserkennung benötigt werden, müssen `public` sein.

```

virtual t_Real inertia      () const
virtual t_Real inertia      (const t_3DVector& rotAxis)

```

Liefert das Trägheitsmoment J bezüglich der aktuellen Rotationsachse, bzw. bezüglich `rotAxis`. In der Basisklasse entspricht das Trägheitsmoment der Hüllkugel und ist somit für alle Achsen gleich.

```

t_3DVector  impulse      () const
void        impulse      (const t_3DVector& p)

```

Setzen und Abfragen des Impulses $p = m \cdot v$. Setzen des Impulses verändert die Geschwindigkeit.

```

t_3DVector  rotImpulse    () const
void        rotImpulse    (const t_3DVector& L)

```

Setzen und Abfragen des Drehimpulses $L = J \cdot w$. Setzen des Drehimpulses verändert die Winkelgeschwindigkeit.

```

t_Real      linEnergy  () const
void        linEnergy  (t_Real wlin)
t_Real      rotEnergy  () const
void        rotEnergy  (t_Real wrot)

```

Setzen und Abfragen der linearen kinetischen Energie $W_{lin} = \frac{1}{2} \cdot m \cdot v^2$ und der Rotationsenergie $W_{rot} = \frac{1}{2} \cdot J \cdot \omega^2$. Setzen der Energie verändert die Geschwindigkeit bzw. die Winkelgeschwindigkeit. Diese darf wegen der Richtungsbestimmung nicht 0 sein.

```

virtual t_Real volume      () const

```

Liefert das Volumen in m^3 , bei Verwendung der Basisklasse das Volumen der Hüllkugel.

```

t_Real      density      () const
void        density      (t_Real d)

```

Setzen und Abfragen der Dichte (Masse pro Volumen) in $\frac{kg}{m^3}$. Setzen der Dichte verändert die Masse.

```

static void physicalFrame (t_Real timeStep,
                           int      physicalSteps)

```

timeStep wird in physicalSteps Schritte zerlegt. Die physicalFunction aller physikalischer Objekte sowie die Kollisionserkennung und -reaktion werden für jeden Schritt aufgerufen.

Alle Kollisionen eines Schrittes werden als gleichzeitig aufgefaßt.

```

static void collisionDetectionMode (t_CollisionDetection* mode)

```

Setzt die Kollisionserkennungs und -reaktionsroutine, die von physicalFrame aufgerufen wird.

Wird keine bessere gesetzt, so wird die generische Routine verwendet, die jedes Objekt mit jedem testet. Vor einem speziellen Test wird jedoch ein Hüllkugeltest durchgeführt.

```

static t_Real physicalStep()

```

Dauer eines Schrittes in Sekunden s.

```

void      colPoint  (const t_3DVector& point)
t_3DVector colPoint  () const
void      colNormal (const t_3DVector& normal)
t_3DVector colNormal () const

```

Setzen und Abfragen des Kollisionspunktes P_{col} und der Kollisionsnormalen N_{col} .

```

t_3DVector colRadius() const

```

Liefert den Kollisionsradius $r = P_{col} \Leftrightarrow \text{massCenter}$.

```

t_3DVector colPointVelocity() const

```

Liefert die Kollisionspunktgeschwindigkeit $v_{P_{col}} = v + (\omega \times r_{col})$.

```

t_3DVector colPointImpulse() const

```

Liefert den Kollisionspunktimpuls $p_{P_{col}} = p + \frac{J}{r^2} \cdot (\omega \times r_{col})$.

```
void          colMailboxId (t_IdGenType id)
t_IdGenType colMailboxId () const
```

Dient zum Hinterlassen einer ID, die wiederholtes Kollisionstesten verhindert.

```
virtual t_BVol boundingVolume() const
```

Hüllkörper für rotierte und verschobene Objekte. Deformationen könnten Schwierigkeiten machen.

```
t_BSphere boundingSphere() const
```

Hüllkugel, die für die Basisfunktionalität der physikalischen Objekte und zur Kollisionstestbeschleunigung dient.

Vor dem ersten *Step* nicht definiert! Kann mit `checkintersections` initialisiert werden.

```
static t_Bool checkIntersections()
```

Überprüft, ob Durchdringungen in der Szene existieren. (Aktualisiert die Hüllkugeln).

```
virtual t_Id MDId()
```

```
static t_Id sMDId()
```

Liefert immer 0. Abgeleitete Klassen müssen ihre Multiple-Dispatching ID zurückgeben. Sowohl die `virtual`-, als auch die `static`-Funktion muß vorhanden sein.

```
static t_PhysicalObjectPtr first()
```

```
t_PhysicalObjectPtr next()
```

Liste der physikalischen Objekte.

```
protected:
```

```
virtual void physicalFunction (t_Real timeStep)
```

Ändert den physikalischen Zustand des Objektes auf den Zustand nach dem Intervall der Länge `timeStep`.

In der Basisklasse sind Geschwindigkeit und Rotation unter Berücksichtigung aller Instanzen von `t_ForceField` und `t_VelocityField` realisiert.

```
virtual void calcBSphere()
```

```
void boundingSphere(const t_BSphere& sphere)
```

Berechnung der Hüllkugel.

Wird von `physicalStep` und `checkIntersections` aufgerufen.

```
virtual void translate (const t_3DVector& vector)
```

```
virtual void transform (const t_4x3Matrix& matrix)
```

Transformation von `masscenter` und Aufruf der `t_RefObject`-Funktionen.

Überladene Funktionen müssen die Basisfunktionen aufrufen.

```
private:
```

```
void calcRotMatrix (t_Real timeStep)
```

Berechnung der Rotationsmatrix.

```
static void
genericCollisionDetect (const t_PhysicalObjectPtr& phyObject1,
                       const t_PhysicalObjectPtr& phyObject2)
```

Kollisionserkennungsroutine für Kollisionen der Hüllkugeln beliebiger Objekte. Da ein vorheriger Hüllkugelttest vorausgesetzt wird, wird immer True geliefert. Zuvor werden Kollisionspunkt und -normale berechnet.

Wird bei `t_CollisionDetection` registriert.

```
static void
genericCollisionResponse (const t_PhysicalObjectPtr& phyObject1,
                         const t_PhysicalObjectPtr& phyObject2)
```

Realisiert den einpunktigen Stoß starrer Körper. Reibung wird berücksichtigt. Stöße mit Objekten unendlicher Masse werden unterstützt.

Wird bei `t_CollisionDetection` registriert.

A.14 `t_RigidPolyhedron`

```
class t_RigidPolyhedron : public t_PhysicalObject
```

Basisklasse für Objekte, die exakt durch ihre `t_BRep`-Repräsentation gegeben sind. Kollisionen werden mit Hilfe einer ähnlichen Struktur erkannt, die `t_RPRep` heißt. Die `t_RPRep`-Repräsentation muß im Konstruktor der abgeleiteten Klasse erzeugt werden. Dies ist der Funktion `approxShape` ähnlich.

```
public:
t_RigidPolyhedron (const t_3DVector&          massCenter,
                  t_Real                  mass,
                  const t_3DVector&          velocity,
                  const t_3DVector&          rotVelocity,
                  t_Real                  friction,
                  const t_SurfaceShaderPtr& srfshd
                  = (t_SurfaceShaderPtr)NULL)
```

Konstruktor.

```
t_RPRep RPBoundary()
```

Liefert die `t_RPRep`-Repräsentation.

```
t_Id MDId()
```

```
static t_Id sMDId()
```

Liefert Multiple-Dispatching ID.

```
protected:
```

```
t_RPRep v_rpRep
```

`t_RPRep`-Repräsentation, die vom Konstruktor der abgeleiteten Klasse erzeugt werden muß.

```
virtual void translate (const t_3DVector& vector);
```

```
virtual void transform (const t_4x3Matrix& matrix);
```

Transformiert `t_RPRep`-Repräsentation und ruft Basisfunktionen auf.

```
private:
static void
polyhedronCollisionDetect (const t_PhysicalObjectPtr& poly,
                           const t_PhysicalObjectPtr& phyObject)
```

Kollisionserkennungsroutine für Kollisionen mit den Hüllkugeln beliebiger Objekte.
Wird bei t_CollisionDetection registriert.

```
static void polyHalfSpaceCollisionDetect (
                           const t_PhysicalObjectPtr& poly,
                           const t_PhysicalObjectPtr& halfSpace)
```

Kollisionserkennungsroutine für Kollisionen mit t_LimitHalfSpace-Objekten.
Wird bei t_CollisionDetection registriert.

```
static
void polyPolyCollisionDetect (const t_PhysicalObjectPtr& poly1,
                              const t_PhysicalObjectPtr& poly2)
```

Kollisionserkennungsroutine für Kollisionen mit t_RigidPolyhedron-Objekten.
Wird bei t_CollisionDetection registriert.

A.15 t_RPRep

```
class t_RPVertex
public:
    t_RPVertex (const t_3DVector& p)
    t_3DVector  point()
    void        point(const t_3DVector& p)
    t_RPVertex* next()
    void        next(t_RPVertex* v)
```

Element der Eckpunktliste von t_RPRep.

```
class t_RPEdge
public:
    t_RPEdge (t_RPVertex* v1, t_RPVertex* v2)
    t_3DVector  p1()
    t_3DVector  p2()
    t_RPEdge*   next()
    void        next(t_RPEdge* e)
```

Element der Kantenliste von t_RPRep.

```
class t_RPFace
public:
    t_RPFace (t_RPVertex* v1, t_RPVertex* v2, t_RPVertex* v3)
    t_3DVector  p1()
    t_3DVector  p2()
    t_3DVector  p3()
    t_3DVector  normal()
    t_RPFace*   next()
    void        next(t_RPFace* f)
```

Element der Flächenliste von t_RPRep.

```

class t_RPRep
    „RigidPolyhedronRepräsentation“, ähnlich zu t_BRep.
    Führt eine Eckpunkt- eine Kanten- und eine Flächenliste.

public:
t_RPRep()
    Konstruktor

void translate (const t_3DVector& vector)
void transform (const t_4x3Matrix& matrix)
    Transformation aller Eckpunkte.

t_RPVertex* addVertex (const t_3DVector& point)
    Erzeugt Eckpunktlistenelement und fügt es der Liste hinzu.

t_RPEdge* addEdge (t_RPVertex* v1, t_RPVertex* v2)
    Erzeugt Kantenlistenelement und fügt es der Liste hinzu.

t_RPFace* addFace (t_RPVertex* v1,
                  t_RPVertex* v2,
                  t_RPVertex* v3)
    Erzeugt Flächenlistenelement und fügt es der Liste hinzu.

t_RPVertex* firstVertex()
t_RPEdge*   firstEdge()
t_RPFace*   firstFace()
    Zeiger auf die Listenköpfe.

```

A.16 t_RigidQuadrangle

```

class t_RigidQuadrangle: public t_PhysicalObject
    Starres Viereck.

public:
t_RigidQuadrangle (const t_3DVector&      p1,
                  const t_3DVector&      p2,
                  const t_3DVector&      p3,
                  const t_3DVector&      p4,
                  t_Real                  mass,
                  const t_3DVector&      velocity,
                  const t_3DVector&      rotation,
                  t_Real                  friction,
                  const t_SurfaceShaderPtr& srfshd
                  = (t_SurfaceShaderPtr)NULL )

    Konstruktor

```

```
virtual t_3DVector p1() const
virtual t_3DVector p2() const
virtual t_3DVector p3() const
virtual t_3DVector p4() const
```

Die geometrische Spezifikation physikalischer Objekte muß public sein, um Kollisionserkennung zu ermöglichen.

```
virtual void translate (const t_3DVector& vector)
virtual void transform (const t_4x3Matrix& matrix)
```

Transformiert Eckpunkte und ruft Basisfunktionen auf.

```
virtual t_Id MDId()
static t_Id sMDId()
```

Liefert Multiple-Dispatching ID.

```
protected:
virtual void calcBSphere()
```

Kleinste Hüllkugel, die p1 bis p4 einschließt.

```
private:
static void
quadHalfSpaceCollisionDetect (const t_PhysicalObjectPtr& quad,
                               const t_PhysicalObjectPtr& space)
```

Kollisionserkennungsroutine für Kollisionen mit t_LimitHalfSpace-Objekten.
Wird bei t_CollisionDetection registriert.

```
static void
quadrangleCollisionDetect (const t_PhysicalObjectPtr& quad,
                           const t_PhysicalObjectPtr& phyObject)
```

Kollisionserkennungsroutine für Kollisionen mit den Hüllkugeln beliebiger Objekte.
Wird bei t_CollisionDetection registriert.

A.17 t_RigidSphere

```
class t_RigidSphere: public t_PhysicalObject
```

Starre Kugel.

```
public:
t_RigidSphere (const t_3DVector&          center,
               t_Real                radius,
               t_Real                mass,
               const t_3DVector&        velocity,
               const t_3DVector&        rotation,
               t_Real                friction,
               const t_SurfaceShaderPtr& srfshd
               = (t_SurfaceShaderPtr)NULL )
```

Konstruktor

```
virtual t_3DVector center() const
virtual t_Real radius() const
```

Die geometrische Spezifikation physikalischer Objekte muß `public` sein, um Kollisionserkennung zu ermöglichen.

```
virtual t_BVol boundingVolume() const
```

Der Hüllkörper kann schneller als in der Basisklasse berechnet werden.

```
virtual t_Id MDId()
static t_Id sMDId()
```

Liefert Multiple-Dispatching ID.

```
protected:
```

```
virtual void calcBSphere()
```

Die Hüllkugel ist mit der Kugel selbst identisch.

A.18 t_RigidTetrahedron

```
class t_RigidTetrahedron : public t_RigidPolyhedron
```

Starrer Tetraeder.

```
public:
```

```
t_RigidTetrahedron (const t_3DVector& p0,
                   const t_3DVector& p1,
                   const t_3DVector& p2,
                   const t_3DVector& p3,
                   t_Real mass,
                   const t_3DVector& velocity,
                   const t_3DVector& rotVelocity,
                   t_Real friction,
                   const t_SurfaceShaderPtr& srfshd
                   = (t_SurfaceShaderPtr)NULL)
```

Konstruktor

```
t_3DVector p0()
t_3DVector p1()
t_3DVector p2()
t_3DVector p3()
```

Die geometrische Spezifikation physikalischer Objekte muß `public` sein, um Kollisionserkennung zu ermöglichen.

```
virtual t_Id MDId()
static t_Id sMDId()
```

Liefert Multiple-Dispatching ID.

```
protected:
```

```
virtual void calcBSphere()
```

Kleinste Hüllkugel, die `p1` bis `p4` einschließt.

```
virtual void translate (const t_3DVector& vector)
virtual void transform (const t_4x3Matrix& matrix)
```

Transformiert Eckpunkte und ruft Basisfunktionen auf.

A.19 t_Velocityfield

```
class t_VelocityField
```

Basisklasse zur Realisierung von Strömungsfeldern. Die Viskosität ist `static`.

```
public:
```

```
t_VelocityField()
```

Konstruktor

```
virtual ~t_VelocityField()
```

Entfernt Instanz aus interner Liste.

```
static void viscosity (t_Real v)
```

```
static t_Real viscosity()
```

$v \in [0, 1]$. 0 entspricht leerem Raum, 1 einer sehr zähen Flüssigkeit.

Die Reaktion von Objekten kann auch von deren Form abhängen.

```
virtual
```

```
t_3DVector velocity (const t_3DVector& position) const = 0
```

Strömungsvektor an Punkt `position`.

```
static t_3DVector allVel (const t_3DVector& position)
```

Summe der Strömungsvektoren aller Instanzen.

B mrtphys-Klassenreferenz**B.1 t_AnimationDialog**

```
class t_AnimationDialog
```

Realisiert das Animationsmenü, das ständig im Hauptfenster sichtbar ist.

```
public:
```

```
t_AnimationDialog (t_PhysicalAnimation* physAnim)
```

physAnim wird für die Animationssteuerungsparameter und die Displaywerte benötigt.

```
void initDialogObjects()
```

Muß nach dem Parsen aufgerufen werden.

```
void update()
```

Aktualisiert die Displays.

```
private:
```

```
friend class t_StopGoAction
```

```
friend class t_StepAction
```

```
friend class t_RealtimeOffAction
```

```
friend class t_RealtimeOnAction
```

```
friend class t_RealtimeForcedAction
```

```
friend class t_BSphereAction
```

```
friend class t_RegGridAction
```

```
friend class t_AdaptiveAction
```

```
friend class t_FpsActionSlider
```

```
friend class t_PspfActionSlider
```

```
friend class t_RegGridActionSlider
```

```
friend class t_AdaptiveActionSlider
```

Klassen der Penguin-Aktionen und -Schieberegler.

B.2 t_ForcesDialog

```
class t_ForcesDialog
```

Realisiert das *Forces*-menü.

```
public:
```

```
t_ForcesDialog (t_EarthGravity* gravity)
```

Konstruktor

```
void initDialogObjects()
```

Muß nach dem Parsen aufgerufen werden.

```
private:
```

```
friend class t_GravityResetAction
```

```
friend class t_GravityActionSlider
```

```
friend class t_ViscosityActionSlider
```

Klassen der Penguin-Aktionen und -Schieberegler.

B.3 t_PhysicalAnimation

```
class t_PhysicalAnimation
```

Steuerung der Animation.

```
public:
```

```
t_PhysicalAnimation (unsigned int    framesPerSecond,
                    unsigned int    physicalStepsPerFrame,
                    t_SceneWindowPtr scene,
                    t_IdlePtr       idle,
                    t_Bool          on = False)
```

scene bezeichnet das Animationsfenster.

idle wird für die Aufrechterhaltung der Hauptschleife des Programms benötigt.

on gibt an, ob die Animation sofort starten soll.

```
t_Bool nextFrame()
```

Überprüft ob aufgrund des *Realtime*-Modus und der fortgeschrittenen Zeit ein neuer *Frame* erzeugt werden muß. Wenn ja wird der *Frame*-Zähler hochgezählt, `t_PhysicalObjekt::physicalFrame` wird aufgerufen und `True` wird zurückgegeben.

```
unsigned int framesPerSecond() const
```

```
void framesPerSecond (unsigned int framesPerSecond)
```

Setzt die gewünschte *Frame*-Rate und setzt den Zähler zurück.

```
unsigned int physicalStepsPerFrame() const
```

```
void physicalStepsPerFrame (unsigned int physicalStepsPerFrame)
```

Setzt die *Steps per Frame*-Rate ohne den Zähler zu verändern.

```
t_Bool on() const
```

```
void on (t_Bool on)
```

Starten und Stoppen der Animation.

```
void step()
```

Auslösung eines Einzelbildes.

```
t_Bool realtime() const
```

```
void realtime (t_Bool on)
```

Bei abgeschaltetem *Realtime*-Modus läuft die Animation so schnell wie möglich.

```
t_Bool realtimeForced() const
```

```
void realtimeForced (t_Bool on)
```

Im *Forced*-Modus wird, wenn nötig, Echtzeit durch Überspringen von *Frames* erzwungen (Nur im *Realtime*-Modus).

```
int framesSkipped() const
```

Im *Forced*-Modus wird die Zahl der übersprungenen *Frames* geliefert.

```
t_Real fpsReached() const
```

Liefert die zuletzt erreichte *Frame*-Rate.

```
const char* timeCodeString(char* string)
```

Liefert den Timecode-String im Format:

<Stunden> h: <Minuten> ': <Seconden> ": <Frames> f

B.4 t_RenderDialog

```
class t_RenderDialog
```

Realisiert das *Render*-menü.

```
public:
```

```
t_RenderDialog (t_SceneWindowPtr scene)
```

scene wird für die sofortige Umsetzung der Eingaben benötigt.

```
void initDialogObjects()
```

Muß nach dem Parsen aufgerufen werden.

```
private:
```

```
friend class t_ShadingAction
```

```
friend class t_IlluminationAction
```

```
friend class t_RenderingAction
```

```
friend class t_TriQualityActionSlider
```

Klassen der Penguin-Aktionen und -Schieberegler.

B.5 t_SceneWindow

```
enum t_rendering {DefaultRendering, WireframeRendering,
                  ScanlineZBuffer, ZBuffer, File}
```

Aufzählungstyp für Renderingmodi.

```
class t_SceneWindow : public t_CgiGlyph
```

Öffnet ein *t_CgiGlyph*-Fenster als CGI-Ausgabegerät zur Darstellung der Animation mittels CGI-3D.

Durch die *t_CgiGlyph*-Funktionalität ist Interaktion in diesem Fenster möglich. Diese wird zur Manipulation der Betrachterposition genutzt.

Nach dem Öffnen des Fensters wird die Szene gemäß der Kommandozeilenparameter dargestellt. Es werden jedoch nur approximative Darstellungsmodi unterstützt.

```
public:
```

```
t_SceneWindow (int argc, char** argv, t_Penguin* pen,
               t_Bool& success, char* inName = "")
```

Ruft open auf.

```
virtual ~t_SceneWindow()
```

Ruft nicht close auf!

```
t_Bool open (char* inName)
```

```
void close()
```

Öffnen und Schließen der Szene

```
t_shading      shading()      const
t_illumination illumination() const
t_rendering    rendering()    const
t_Real         triQuality()   const
void shading   (t_shading shading)
void illumination (t_illumination illumination)
void rendering  (t_rendering rendering)
void triQuality (t_Real triQuality)
```

Setzen und Abfragen der Renderingparameter.

```
void checkSetup()
```

Ändert nicht unterstützte Renderingwerte.

```
private:
```

```
static void glyphDrawFunction      (t_Cgi* cgi,
                                     const t_2DVector& lower,
                                     const t_2DVector& upper)
static void glyphPressFunction      (t_CgiGlyphPtr glyph,
                                     const t_EventMousePtr event)
static void glyphDragLeftFunction   (t_CgiGlyphPtr glyph,
                                     const t_EventMousePtr event)
static void glyphDragMiddleFunction(t_CgiGlyphPtr glyph,
                                     const t_EventMousePtr event)
static void glyphDragRightFunction (t_CgiGlyphPtr glyph,
                                     const t_EventMousePtr event)
```

CgiGlyph-Funktionen zur Manipulation der Betrachterposition.

Literatur

- [Bar89] BARAFF D.: Analytical Methods for Dynamic Simulation of Non-penetrating Rigid Bodies, in ACM SIGGRAPH '89, vol. 23, no.3, Boston, 1989.
- [Ben97] BENDELS H.: *Eine topologische Datenstruktur und ihre Anwendung im 3D-Graphiksystem MRT*, Diplomarbeit, Institut für Informatik, Rheinische Friedrich-Wilhelms-Universität Bonn, 1997.
- [Fel92] FELLNER D. W.: *Computergrafik*, 2 ed., vol. 58 of *Reihe Informatik*, B.I. Wissenschaftsverlag, Mannheim, 1992.
- [Fel96a] FELLNER D. W.: Extensible Image Synthesis, in *Object-Oriented and Mixed Programming Paradigms*, Wisskirchen P. (Ed.), Focus on Computer Graphics. Springer, pp 7-21, Feb. 1996.
- [Fel96b] FELLNER D. W.: MRT: Design Issues and Brief Reference, available via <http://hyperg.cs.uni-bonn.de/TeachingMaterial.CG.API>, Jan 1996.
- [FFW93] FELLNER D. W., FISCHER M., WEBER J.: *CGI-3D - A 3D Graphics Interface*, Tech. Rep. IAI-TR-95-x, University of Bonn, Dept. of Computer Science, Bonn, Germany, August 1993.
- [Fis95] FISCHER, M.: *Runtime Type Information for Class Hierarchies*, Tech. Rep. IAI-TR-95-11, University of Bonn, Dept. of Computer Science, Bonn, Germany, July 1995.
- [Gla95] GLASSNER A.S.: *Principles Of Digital Image Synthesis*, Morgan Kaufmann Publisher, San Francisco, 1995.
- [Gro91] GROSSMANN S.: *Mathematischer Einführungskurs für die Physik*, Teubner Studienbücher, Stuttgart, 1991.
- [GV93] GERTHSEN C., VOGEL H.: *Physik*, 17 ed., Springer, 1993.
- [Ha88] HAHN J. K.: Realistic Animation of Rigid Bodies, in ACM SIGGRAPH '88, vol. 22, no.4, Atlanta, 1988.
- [Ham78] HAMEL G.: *Theoretische Mechanik*, Berichtigter Reprint von Band 57 der *Grundlehren der mathematischen Wissenschaften* von 1949, Springer, 1978.
- [Hau92] HAUMANN D., WEICHERT J.: Choreography, in ACM SIGGRAPH '92, Course Notes vol. 16, pp. 2-1 - 2-29, Chicago, 1992.
- [Mül97] MÜLLER G.: *Beschleunigung strahlbasierter Rendering-Algorithmen*, Diplomarbeit, Institut für Informatik, Rheinische Friedrich-Wilhelms-Universität Bonn, Mai 1997.
- [MW88] MOORE M., WILHELMS J.: Collision Detection and Response for Computer Animation, in ACM SIGGRAPH '88, vol. 22, no.4, Atlanta, 1988.
- [SEYA96] SCHECHTER G., ELLIOT C., YEUNG R., ABI-EZZI S.: Functional 3D Graphics in C++ - with an Object-Oriented, Multiple Dispatching Implementation, in *Object-Oriented and Mixed Programming Paradigms*, Wisskirchen P. (Ed.), Focus on Computer Graphics. Springer, pp 172-193, Feb. 1996.
- [Str91] STROUSTUP B.: *The C++ Programming Language*, 2 ed., Addison Wesley, Reading, Massachusetts, 1991.

Versicherung

Hiermit versichere ich an Eides Statt, daß ich die vorliegende Arbeit „Kollisionserkennung und -reaktion physikalischer Objekte in virtuellen Umgebungen“ selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Köln, den 3. Februar 1998