Praktikum: Dynamische Simulation von Mehrkörpersystemen

Dokumentation

Igor Achieser, Tobias Becker

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Inhaltsverzeichnis

1 Einführung

Das entwickelte Projekt ist entstanden im Sommersemester 2012 an der TU Darmstadt. Der Kurs wurde geleitet von Prof. Jan Bender. Das Projekt ist erreichbar über SVN : http://subversion.assembla.com/svn/dsmks/

Das Ziel des Praktikums war ein funktionierendes System für die Simulation von dynamischen Mehrkörpersystemen zu erstellen. Zu diesem Zwecke wurden fünf Aufgaben gegeben:

1. Modellierung von Starrkörpern und gedämpften Federn, Einbezug von externen Kräften.
2. Impulsbasierte Simulation von Kugelgelenken.
3. Erstellen eines Textilmodells und Implementierung des Normalisierungsalgorithmuss nach Provot [1].
4. Kollisionserkennung für Kugeln, Ebenen und Quader.
5. Impulsbasierte Kollisionsauflösung mittels Impulse.

Es wurde ein programmatisches Grundgerüst von Prof. Bender zur Verfügung gestellt. Dieses beinhaltete grundlegende Simulationslogik, Benutzerinteraktion mit Visualisierung sowie eine mathematische Bibliothek.

2 Systembeschreibung

Die Programmarchitektur legt besonderen Wert auf Trennung der Interessen und klare Struktur.

Die mathematischen Klassen, die zur Verfügung gestellt wurden, wurden im Laufe des Projekts immer wieder um Funktionalität und Laufzeitoptimierungen erweitert. Es ist jedoch nichts Grundlegendes geändert worden, weshalb in dieser Dokumentation nicht darauf eingegangen wird.

In den folgenden Abschnitten wird jeder Bereich der Implementation besprochen. Diese Bereiche korrespondieren sehr gut mit den gestellten Aufgaben des Praktikums.

2.1 Framework

Das Simulationsframework basiert insbesondere auf zwei Klassen: ISimulationObject und ISimulationModule. Der Zusammenhang wird in Grafik 2.1 verdeutlicht: ISimulationObject ist die Basisklasse für alle grundlegenden Simulationsklassen. Die Klasse ISimulationObject kapselt die Namen der Objekte und bietet Initialisierungs- und Aufräumfunktionalität, die über einfache Konstruktoren / Destruktoren nicht möglich ist. ISimulationObject schützt seinen eigenen Zustand: Es wird sichergestellt, dass das Objekt nur einmal initialisiert und nur aufgeräumt werden kann, falls es initialisiert ist. Dies ist praktisch für größere Vorausberechnungen oder das Laden von Daten.

ISimulationModule ist eine Subklasse von ISimulationObject und wird in verschiedenen Bereichen der Simulation verwendet um ISimulationObject-Instanzen, für die sie zuständig ist, zu gruppieren und während des Simulationschrittes korrekt zu verarbeiten. So kümmert sich z.B. der Dynamikalgorithmus um alle Gelenke der Simulation. ISimulationModules machen im Endeffekt die Simulation aus, weil sie die wesentliche Funktionalität beinhalten.

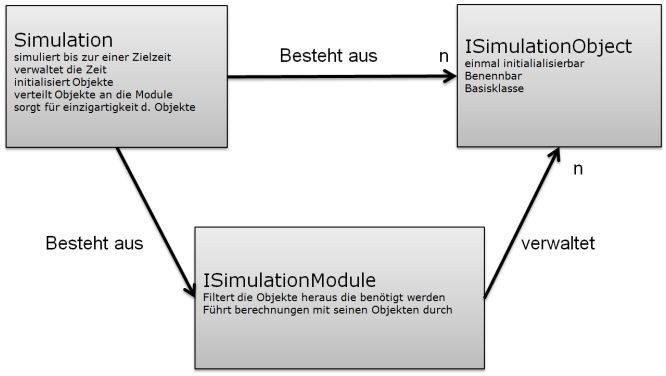

Abbildung 2.1: Zentrale Architektur des Frameworks

2.2 Kinematik - simulation/kinematics

Die erste Aufgabe bestand darin im 3D frei bewegliche Körper zu Simulieren. Hierzu wurde als erste Abstraktionsstufe das CoordinateSystem sowie KinematicBody gewählt. Ein Koordinatensystem besteht aus einer Position und einer Orientierung. Der KinematicBody erbt davon und stellt zudem noch die ersten zwei Ableitungen der Position und der Orientierung zur Verfügung: Die lineare bzw. angulare Geschwindigkeit bzw. Beschleunigung. Ein KinematicBody besitzt also alle kinetischen Parameter.

2.3 Integration - simulation/integration

Um tatsächlich Bewegung ins Spiel zu bringen, müssen die Differentialgleichungen des KinematicBody gelöst werden. Hierzu implementiert der KinematicBody das Interface IIntegrable, welches über die Methoden getState(), setState(), getDerivedState() auf den kompletten Zustand des KinematicBody in der Form eines Gleitkommazahl-Arrays zurückgreifen kann. Dieses generische Array wird verwendet für die Integration. Da es in der Simulation nicht nur einen Körper gibt, musste eine Möglichkeit geschaffen werden, auf den Zustand sowie die erste Ableitung des Zustandes des kompletten Simulationssystems zuzugreifen. Hier kommt die Klasse CompositeIntegrable ins Spiel. Sie erbt von ISimulationModule und aggregiert alle Simulationsobjekte, welche das IIntegrable Interface implementieren. Sie verwaltet ein großes Array aus Gleitkommazahlen, in dem für jeden Körper der Zustand gespeichert wird.

Neben dem Systemzustand und seiner Ableitung muss auch die Möglichkeit gegeben werden, den abgeleiteten Systemzustand aus einem gegebenen Zustand zu berechnen. Dafür wurde die Klasse ISystemFunction erstellt. Diese Klasse stellt die Methoden evaluate(...), preIntegration(...), sowie postIntegration(...) zur Verfügung, die von den verschiedenen

Algorithmen verwendet werden, um in der korrekten Phase (vor, während und nach der Integration) Berechnungen auszuführen. Die einzige Implementierung der ISystemFunction ist zum Ende des Praktikums die Klasse DynamicsAlgorithm, welche alle verschiedenen Berechnungen/Algorithmen vereint und in der korrekten Reihenfolge ausführt.

Das Zustandsarray und die Ableitung wird an den Integrator übergeben, welcher über ein Interval [a,b] integrieren kann (mit entsprechendem Startzustand).

In der ersten Aufgabe wurde gefordert, dass die zwei Einschrittverfahren ”Expliziter Euler” sowie ”Runge Kutta der Fehlerordnung 5” implementiert werden. Neben diesen beiden Verfahren wurde noch der ”Implizite Euler” (verwendet simple Fixpunktiteration für das Auflösen des nichtlinearen Gleichungssystems) sowie der ”RungeKuttaFehlberg45” (mit dynamischer Schrittweitensteuerung) implementiert. Die Implementierungen dieser Verfahren sind im Ordner simulation/integration/implementations zu finden.

2.4 Dynamik - simulation/dynamics

Masse hebt die Dynamik von der Kinematik ab. Die Aufspaltung der beiden Bereiche ist Software-Design-technisch sinnvoll, da der kinematische Körper sowie das Koordinatessystem noch an anderen Stellen des Simulationssystems verwendet werden (Geometrie und Visualisierung). Die Dynamik bzw. der DynamicBody und seine beiden Subklassen RigidBody und Particle besitzen die kinematischen Eigenschaften des KinematicBody sowie die Masseeigeneschaften, aus welchen zusammen die Möglichkeit folgt, externe Kräfte anzubringen und die Bewegungsgleichung zu lösen. Aus den extern wirkenden Kräften und unter Berücksichtigung des Körperzustandes (momentane Geschwindigkeit und Position) kann die neue Beschleunigung berechnet werden. Hierzu besitzt der DynamicBody die Methode calculateDynamics(). Neben dieser zentralen Funktionalität bietet der DynamicBody noch folgende Möglichkeiten:

- Setzen der Masse
- Setzen/Hinzufügen/Zurücksetzen von externen Kräften und Drehmomenten
- Das Anbringen eines Impulses an einer beliebigen Stelle
- Das Berechnen der Matrix K für die spätere Berechnung von Impulsen.

Der Unterschied zwischen den Klassen Particle und RigidBody besteht in der Orientierung. Partikel sind Objekte mit 3 translatorischen Freiheitsgraden, während der RigidBody alle 6 Freiheitsgrade (translatorisch und rotatorisch) des Raumes beherrscht und somit physikalisch korrekt ist.

Das DynamicBodyModule ist das Simulationmodul, welches die dynamischen Körper sammelt und deren Berechnung anstößt.

2.5 Verbindung - simulation/dynamics/connection

Um dynamische Körper mit Federn, Gelenken oder anderem zu verbinden, wurde das Verbindungstück entwickelt (Klasse Connector). Dieses bietet die Möglichkeit einen beliebigen Punkt im lokalen Koordinatensystem eines dynamischen Körpers zu speichern und als Angriffspunkt für Kräfte, Impulse oder Ähnliches zu verwenden. Für Starrkörper und Partikel wurden eigene Implementierungen des Connectors entwickelt, welche in RigidBodyConnector und ParticleConnector zu finden sind. Ein ParticleConnector ist eine degenerierte Version des RigidBodyConnectors, da ein ParticleConnector mangels Ausdehnung nur an den Lokalen Ursprung verbunden werden kann.

Da häufig Verbindungsstücke für die verschiedenen dynamischen Körpertypen erstellt und gelöscht werden, wurde die Klasse ConnectorFactory erstellt, welche sich um die Speicherverwaltung der Connectorobjekte kümmert.

2.6 Kräfte und Federn- simulation/force

Das Kräftemodell ist relativ einfach gestalltet. Das Kräftemodul ForceModule filtert sich alle Force-Objekte sowie DynamicBody-Objekte heraus. In jedem dynamischen Körper ist ein Kraftakkumulator, welcher über DynamicBody::addExternalForce/Torque externe Kräfte bzw. Drehmomente hinzugefügt bekommt. Dieser wird am Anfang des Simulationsschrittes zurückgesetzt auf Null. Danach wird jede einzelne Kraft angewendetet. Jede Kraft kann dabei eine Funktion sein, die auf alle, einige oder einzelne Dynamischen Körper wirkt. Dies ermöglicht beispielsweise die Modellierung von Schwerkraft - in der Lösung in der Klasse Gravity - oder Federn. Die zentrale Methode bei den Kräften ist Force::act(vector<DynamicBody*> & bodies, Real time, Real stepSize). Es ist also möglich beliebige Zeitabhängige Kraftfelder auf alle Körper wirken zu lassen

Das Federmodell, gefunden in der Klasse DampenedSpring, modelliert eine Feder mit Federkonstante F_C und Dämpfungskonstante F_d. Um die Federkräfte auf Partikel und Starrkörper anzuwenden wird hier jedoch der Umweg über die Verbindungstücke gegangen.

2.7 Mehrkörperdynamik - simulation/multibody

Körper interagieren miteinander unter anderem über Gelenke. Ein Gelenk in einem impulsbasierten Simulationssystem definiert einen Positions- und einen Geschwindigkeits-Constraint, die mit Impulsen realisiert werden müssen. Deshalb definiert die Schnittstelle der Klasse Joint die entsprechenden Methoden: correctPosition() und correctVelocity(). Die Klasse, die den impulsbasierten Mehrkörperdynamikalgorithmus enthält, ImpulseBasedDynamicsAlgorithm, ruft correctPosition vor jedem Simulationsschritt auf und correctVelocity nach jedem Simulationsschritt. correctPosition wird dabei in einer Schleife aufgerufen, da man den erforderlichen Impuls nicht direkt exakt berechnen kann.

In diesem Praktikum wurden Kugelgelenke (Klasse BallJoint) implementiert. Sie halten zwei dynamische Körper in bestimmten Punkten (Connectors, vgl. 2.5) zusammen.

2.8 Textilsimulation - simulation/textile

Textilien werden in der Klasse TextileModel als Partikelmengen modelliert, zwischen denen ein Netz aus gedämpften Federn aufgespannt ist. Es gibt dabei 3 Arten von Federn: Federn, die der Ausdehnung, der Scherung und der Beugung des simulierten Stoffes entgegenwirken (vgl. [1])

Nach jedem Simulationsschritt wird in der Methode normalize() Normalisierung durchgeführt: entstandene Überdehnungen werden aufgelöst, indem die Partikel an den Enden einer Feder entlang dieser Feder aufeinander zu bewegt werden, damit die Maximalüberdehnung nicht überschritten wird. Auf diese Weise wirken die Textilien nicht unnatürlich elastisch.

2.9 Geometrie - simulation/geometry

Um die folgenden Abschnitte über Kollisionen zu ermöglichen mussten Geometrische Objekte erstellt werden. Hierzu wurde die Klasse Geometry eingeführt welche zum einen ein CoordinateSystem besitzt das die Position und Orientierung eines geometrischen Gebildes beschreibt. Nebst dieser Hauptfunktionalität hat Geometrie einige virtuelle Methoden um einen Körper auf eine Achse zu projezieren diese werden verwednet um beispielsweise die Größe eines Hüllkörpers zu berechnen.

Ein weitere wichtiger Punkt der später wichtig ist, ist dass man Kugeln und Quader klassifizieren kann als Innerhalb, Außerhalb oder auf dem Rand der Geometrie.

Die eigentliche Geometrie wird jedoch in den Subklassen von Geometry definiert. Es gibt

- Die Kugel (Klasse Sphere)
- Die Ebene (Klasse Plane)
- Das Polygon (Klasse Polygon)

Die Kugel und die Ebene sind einfache Gebilde. Das Polygon verwendet eine Half-Edge Datenstrukture für die Interne Rerpäsentation. Es durch die Polygonklasse möglihc beliebige Geometrien zu erstellen. Zum Beispiel wurden Subklassen von Polygon erstellt: Hexahedron, Pyramid, Triangle, Rectangle, und PlyMesh. Wobei das letztere beliebige Gitter aus dem Ply Format von Stanford laden kann.

Weiterhin wurde eine Hüllgeometrie erstellt namens BoundingOctree. Diese nähert beliebige Geometrien an. Das Bottom-Up Verfahren zum Aufbau der Octreehierarchy geschieht durch das Überziehen der Geometrie mit einem Gitternetz(mit beliebiger Auflösung) und anschließender geometrischer Klassifizierung der einzelnen Zellen als innerhalb, außerhalb oder auf dem Rand der Geometrie. Im darauffolgenden Schritt werden jeweils 8 Zellen die nur Innerhalb oder nur Außerhalb des Octrees sind rekursiv zusammengefasst zu einer 8 mal so großen Zelle bis die oberste Ebene erreicht ist.

2.10 Kollisionserkennung - simulation/collision/detection

Die Basisklasse für die heist CollisionDetector diese nimmt Objekte vom Typ Collidable entgegen. Collidable Objekte wiederum hat eine Referenz auf ein beliebiges ISimulationObjekt welches dann auf Kollisionen getestet wird. Weiterhin wird dem Collidable mitgeteilt ob es an Kollisionen teilnimmt. Die Kollision selbst besteht aus den beiden beteiligten Simulationsobjekten sowie einer beliebigen Zahl an Kontakten (da ein Körper an mehreren Stellen gleichzeitig kollidieren kann). Ein Kontakt ist in der Klasse Contact modelliert. Er besteht aus den beiden Kollisionspunkten, der Kontaktnormalen und der Durchdringungstiefe der beiden beteiligten Objekten.

CollisionDetector arbeitet noch unoptimiert. Dies bedeutet, dass jede potentielle Kollision überprüft wird was eine quadratische Anzahl an Testaufrufen resultiert. Hier besteht auf jeden fall noch großes Optimierungspotential und diese änderung ist vorgesehen in dem Klassendesign.

Für jedes Kombination von Collidableobjekten wird ein Kollisionstest gesucht (abhängig von den Klassentypen). Es gibt folgende Kollisionstests:

- SphereSphere - Kugel gegen Kugel
- SpherePlane - Kugel gegen Ebene
- OctreeOctree - Zwei Octree Hüllkörper gegeneinander
- SphereOctree
- PlaneOctree
- ReverseTest - dreht einen Test herum

Diese sind abrufbar über die Klasse CollisionTestRepository.

Fall einer diese Tests Kollisionen erkennt erzeugt er für jeden Kontaktpunkt ein Contact Objekt und speichert diese in dem Kollisionobjekt.

Nach jedem Simulationsschritt werden die gefundenen Kollisionen gelöscht. Hierbei wurde darauf geachtet, dass die gelöschten Objekte wieder verwendet werden, da die Initialisierung sehr lange dauert.

2.11 Kollisionsbehandlung - simulation/collision/handling/impulsebased

Die Kollisionsbehandlung wird im Anschluss an die Kollisionserkennung vor dem Simulationsschritt durchgeführt. In der Klasse ImpulseBasedCollisionHandler werden in der Methode handleCollisions() alle erkannten Kollisionen in einer Schleife behandelt, bis keine Kollisionen mehr auftreten. Bei der Behandlung einer Kollision werden die Kontakte, aus denen sie besteht, zu einem Kontakt zusammengefasst und in der Methode handleContact behandelt.

Bewegen sich die Kontaktpunkte aufeinander zu, wird ein Rückstoß mittels eines Impulses simuliert. Handelt es sich um einen bleibenden Kontakt, wird ein künstliches Gelenk (Instanz der Klasse ContactJoint) erzeugt, welches dafür sorgt, dass die Körper sich nicht durchdringen und welches dynamische und statische Reibung simulieren kann. Die entsprechenden Constraints werden durch Impulse in Richtung der Kontaktnormalen (Nichtdurchdringung) bzw. -tangente (Reibung) realisiert.

Literaturverzeichnis

[1] X. Provot. Deformation constraints in a mass-spring model to describe rigid cloth behavior. In *IN GRAPHICS INTERFACE*, pages 147–154, 1995.