JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

A Unified Framework for Rigid Body Dynamics

MAGISTERARBEIT

zur Erlangung des akademischen Grades

DIPLOMINGENIEUR

im Magisterstudium

INFORMATIK

Angefertigt am *Institut für Graphische und Parallele Datenverarbeitung*

Eingereicht von:

Helmut Garstenauer

Betreuung:

o. Univ.-Prof. Dr. Jens Volkert

Mitbetreuung:

Dipl.-Ing. Dr. Gerhard Kurka

Linz, März 2006

Abstract

Modern 3d applications convince through models with high polygon counts, detailed textures, and advanced lighting techniques. The results are closer to photographic realism than ever before. But so often this illusion of a virtual world is lost as soon as objects start to move and interact.

One method to improve the believability of moving, interacting objects is simulation of physics. Idealization of real world objects helps to simplify and speed up the simulation. One important idealization is a *rigid body* – a body that never changes its shape. The majority of objects in interactive 3d applications can be treated as rigid bodies or systems of connected rigid bodies. Examples are barrels, bricks, rocks, cupboards with movable doors, and even humans and animals with movable limbs.

These rigid bodies can collide with each other and are subject to forces, like gravity and friction, and constraints. Constraints are conditions that restrict the movement of bodies. For example, a joint constraint connects two bodies.

This thesis summarizes the fundamentals of mechanics that are necessary to build a simulation of rigid body dynamics. A modular overview of the simulation process is given, and the state of the art simulation methods will be examined. With this knowledge a framework for the simulation of rigid body dynamics in interactive 3d applications is presented. The designed framework is independent of special simulation methods and allows incorporating all examined simulation methods and physical effects.

Kurzfassung

Moderne 3D-Anwendungen überzeugen durch Modelle mit hohen Polygonzahlen, detaillierten Texturen und fortgeschrittenen Beleuchtungstechniken. Die Ergebnisse sind fotorealistischer als jemals zuvor. Doch oft geht die Illusion einer virtuellen Welt verloren, sobald Objekte anfangen sich zu bewegen und zu interagieren.

Eine Methode, um die Glaubwürdigkeit von sich bewegenden, interagierenden Objekten zu verbessern, ist die Simulation von Physik. Die Idealisierung von Objekten der realen Welt hilft die Simulation zu vereinfachen und zu beschleunigen. Eine wichtige Idealisierung ist ein *Starrkörper* – ein Körper, der seine Gestalt niemals verändert. Die Mehrheit der Objekte in interaktiven 3D-Anwendungen lassen sich als Starrkörper oder Systeme von verbundenen Starrkörpern beschreiben. Beispiele sind Fässer, Ziegel, Steine, Kästen mit beweglichen Türen, sogar Menschen und Tiere mit beweglichen Gliedern.

Diese Starrkörper können miteinander kollidieren und werden von Kräften, wie Schwerkraft und Reibung, und Zwangsbedingungen beeinflusst. Zwangsbedingungen sind Bedingungen, die die Bewegung von Körper einschränken. Ein Beispiel ist ein Gelenk, das zwei Körper miteinander verbindet.

Diese Magisterarbeit fasst die Grundlagen der Mechanik zusammen, die notwendig sind um eine Simulation von Starrkörpern zu realisieren. Sie gibt einen modularen Überblick über den Simulationsprozess und untersucht die aktuellen Simulationsmethoden. Basierend auf diesem Wissen wird ein Framework für die Simulation von Starrkörper-Dynamik in interaktiven 3D-Anwendungen entwickelt. Das Framework vereinheitlicht bestehende Simulationsmethoden und ermöglicht die Verwendung aller untersuchten Simulationsverfahren und physikalischen Effekte.

Acknowledgments

I want to thank those individuals that have made my studies and this work possible through their technical, financial or personal support.

Special thanks deserve those two beloved persons who have supported me for my whole live – my parents Stefanie and Konrad Garstenauer.

Most of all I want to thank my twin brother Martin Garstenauer. His suggestions were crucial to improve this work.

Of course I want to thank o.Univ-Prof. Dr. Jens Volkert and Dipl.-Ing. Dr. Gerhard Kurka for their advice and for supervising this work.

To Dipl.-Inf. (FH) MSc. Christoph Anthes I am grateful for the great cooperation in several projects and for proofreading this thesis.

And I would like to thank everyone who reads this work and thus imbues it with meaning. I hope this work is your shortcut on your path to the dynamics of rigid bodies.

Danksagung

Ich möchte all jenen danken, die durch ihre fachliche, finanzielle oder persönliche Unterstützung mein Studium und diese Arbeit ermöglicht haben.

Besonderer Dank gebührt jenen zwei geliebten Menschen, die mich mein Leben lang unterstützt haben – meinen Eltern Stefanie und Konrad Garstenauer.

Am meisten danke ich meinem Zwillingsbruder Martin Garstenauer. Seine Vorschläge haben diese Arbeit entscheidend verbessert.

Natürlich möchte ich mich bei Herrn o.Univ-Prof. Dr. Jens Volkert und Herrn Dipl. Ing. Dr. Gerhard Kurka für ihren Rat und die Betreuung meiner Diplomarbeit bedanken.

Herrn Dipl.-Inf. (FH) MSc. Christoph Anthes bin ich dankbar für die tolle Zusammenarbeit in zahlreichen Projekten und für das Korrekturlesen dieser Arbeit.

Und ich danke alle jenen, die diese Arbeit lesen werden und ihr damit einen Sinn geben. Ich hoffe, diese Arbeit ist Ihre Abkürzung auf dem Pfad zur Dynamik von starren Körpern.

Table of Contents

Abstract ... **iii**

Kurzfassung ... **v**

Acknowledgments ... **vii**

Danksagung ... **ix**

1 Introduction and Overview ... **1**
1.1 Goals and Contributions ... 2
1.2 Chapter Overview ... 2

2 Basics ... **5**
2.1 Scope of this Work ... 5
2.2 Rotations ... 7
2.2.1 Euler Angles ... 7
2.2.2 Rotation Matrices ... 8
2.2.3 Quaternions ... 9
2.3 Newton's Laws of Motion ... 10
2.4 Particles ... 11
2.4.1 Equations of Motion of Particles ... 11
2.4.2 Angular Motion ... 11
2.4.3 Momentum and Impulse ... 13
2.4.4 Energy ... 14
2.4.5 Systems of Particles ... 14
2.5 Rigid Bodies ... 15
2.5.1 World Space and Body Space ... 15
2.5.2 Velocity and Acceleration ... 16
2.5.3 Mass Moment of Inertia ... 17
2.5.4 Time Derivatives of Rotation Matrices and Quaternions ... 20
2.5.5 Equations of Motion of Rigid Bodies ... 22
2.5.6 Applying Forces and Impulses ... 22
2.5.7 Energy ... 23
2.5.8 Systems of Rigid Bodies ... 23

2.6 Forces ... 25
2.6.1 Gravity ... 25
2.6.2 Fluid Dynamic Drag ... 26
2.6.3 Springs and Dampers ... 26
2.6.4 Friction ... 27
2.6.5 Other forces ... 29
2.7 The Linear Complementarity Problem ... 29
2.7.1 Problem Definition ... 29
2.7.2 Mixed LCP ... 30
2.7.3 Dependent Limits ... 31
2.7.4 Solving the LCP ... 31

3 Simulation Overview ... 33
3.1 Scope of Simulation ... 33
3.1.1 Rigid Bodies vs. Deformable Bodies ... 33
3.2 Simulation Modules ... 34
3.3 Force Computation ... 35
3.4 Velocity and Position Update ... 36
3.4.1 Numerical Integration ... 37
3.4.2 Implementation Issues ... 38
3.5 Collision Detection ... 39
3.5.1 Reduced Contact Set ... 39
3.5.2 Contact Information ... 40
3.5.3 Algorithms ... 40
3.6 Collisions, Contacts, and Constraints ... 40
3.6.1 Constraint Solver ... 41
3.7 Error Correction ... 43
3.8 Time Step Methods ... 43
3.8.1 Fixed Time Step Methods ... 43
3.8.2 Adaptive Time Step Methods ... 46
3.8.3 Other Time Step Methods ... 47
3.9 Sleeping ... 47
3.10 Conclusion ... 48

4 Constraint Solver ... 49
4.1 Constraint Equations ... 49
4.1.1 Constraint Functions ... 49
4.1.2 Time Derivatives of Constraint Functions ... 50
4.1.3 Systems of Constraints ... 51
4.2 Constraint Geometry ... 52
4.3 General Constraints ... 53
4.3.1 Ball-and-Socket Joints ... 53
4.3.2 Hinge Joints ... 53
4.3.3 Linear Limits ... 54
4.3.4 Other Constraints ... 54
4.4 Collisions and Contacts ... 55
4.4.1 Collisions ... 56
4.5 Sequential Methods ... 58
4.5.1 Sequential Force-Based Methods ... 58

4.5.2 Sequential Impulse-Based Methods ... 59
4.6 Simultaneous Methods ... 64
4.6.1 Constraint Forces ... 64
4.6.2 Simultaneous Force-based Methods ... 65
4.6.3 Simultaneous Impulse-based Methods ... 68
4.7 Conclusion ... 71

5 Error Correction ... 73
5.1 Error Reduction Parameter ... 73
5.2 Baumgarte's Constraint Stabilization Method ... 74
5.2.1 Sequential Impulse-based Methods ... 74
5.2.2 Simultaneous Force-based Methods ... 75
5.2.3 Simultaneous Impulse-based Methods ... 75
5.3 Position-based Methods ... 75
5.3.1 Sequential Position-based Methods ... 76
5.3.2 Simultaneous Position-based Methods ... 76
5.4 Conclusion ... 77

6 Other Simulation Methods ... 79
6.1 Velocity-less Formulations ... 79
6.2 Generalized Coordinates and Lagrangian Dynamics ... 80
6.3 Other Methods ... 81

7 A Unified Framework ... 83
7.1 Requirements ... 83
7.2 Used Technologies ... 84
7.3 Design Issues ... 84
7.4 Simulation Objects ... 84
7.4.1 Rigid Bodies ... 85
7.4.2 Forces ... 89
7.4.3 Constraints ... 90
7.5 Simulation Strategies ... 93
7.5.1 Integrators ... 93
7.5.2 Collision Detection ... 94
7.5.3 Constraint Solvers ... 94
7.5.4 Time Step Strategies ... 96
7.5.5 Sleeping ... 96
7.5.6 Validation Rules ... 98
7.5.7 Simulation ... 99
7.6 Conclusion ... 102

8 Evaluation ... 103
8.1 Viewer ... 103
8.2 Example: Seesaw ... 104
8.3 Example: Parallel Simulation ... 106
8.4 Example: Error Correction ... 107
8.5 Example: Ragdoll ... 108
8.6 Lessons Learned ... 110
8.6.1 Problem Sources ... 110
8.6.2 Repeatability ... 111

8.6.3 Visible errors ... 111
8.6.4 Bad performance ... 111
8.6.5 Believability ... 112

9 Conclusion and Future Work ... 113
9.1 Conclusion ... 113
9.2 Future Work ... 114

A Notation ... 115

B Analogy Translation – Rotation ... 119

C Mass Moment of Inertia ... 121

List of Figures ... 123

Bibliography ... 125

Index ... 133

Eidesstattliche Erklärung ... 137

Lebenslauf ... 139

1 Introduction and Overview

"The illusion and immersive experience of the virtual world, so carefully built up with high polygon models, detailed textures and advanced lighting, is so often shattered as soon as objects start to move and interact."

– Gary Powell, MathEngine Plc

Interactive 3d applications are closer to photographic realism than ever before. Modern graphics algorithms and graphics hardware yield amazing results. The next challenge, which is still not solved satisfyingly, is believable motion.

The motion of objects in applications can be predefined or computed at runtime. In both cases the laws of physics have to be obeyed. Simulation of physics is a tool that relieves an animator from needless work. Manual animation is still mandatory for certain situations, for example for animation of objects that convey emotions. But the majority of animations result from objects that are subject to physical effects – clearly, a collapsing wall is nothing one wants to animate by hand.

The next challenge after motion is interaction, like collisions and contacts of objects. This is obviously a field in which simulation of physics is indispensable. Physics is difficult to overlook, and this thesis focuses on a small part: the dynamics of rigid bodies. Rigid bodies are an idealization of real bodies: Bodies are assumed to have a fixed, unchangeable shape. This idealization is a very practical one, since the majority of objects in interactive 3d applications can be treated as rigid bodies[1].

Rigid bodies are subject to forces, such as gravity, friction, and damped springs. These forces cause motion, translation and rotation, which have to be computed. The key elements of object interactions are collisions. Collisions have to be detected, and the post-collision motions have to be found. Joints can connect rigid bodies, and the allowed motion of bodies can be restricted. These conditions are called constraints. Complex systems of rigid bodies can be assembled using constraints; for example humans with movable limbs, or cars with turnable wheels and movable doors. Constraints can even be used as motors to drive the wheels.

Implementing dynamics in own applications is a difficult task: A lot of knowledge from mechanics

1 Typical examples for non-rigid bodies are clothes, hair, or fluids.

and other fields is necessary. Most research focuses on different selected methods and use different formalisms, which makes them difficult to understand and hard to compare.

1.1 Goals and Contributions

The goal of this thesis is to build a framework for the simulation of rigid body dynamics. The desired framework should exhibit these qualities:

- It should be suitable for usage in *interactive 3d applications* on standard PCs. So only fast simulation methods that qualify for interactive and real-time applications are examined.
- The simulation results should be *believable*. Physical correctness is not the main goal, as long as the results are felt to be realistic by a user of the application. This requires the simulation to be *stable* and *robust*. Here, stability means that unrealistic motion due to simulation errors is avoided. Robustness means that the simulation is able to recover from user- or simulation-induced errors.
- The framework should be *easy to use*. The effort to understand and integrate rigid body dynamics in an application should be low.
- And the framework should be *independent* of existing simulation methods or specific physical effects – for several reasons: Without being an expert in this field it is impossible to decide which simulation method is superior. An independent design facilitates the understanding and comparison of different simulation methods. And hopefully it leads to an improved design, which can be extended easily with new simulation methods and physical effects.

With this goal at the horizon, the general contributions of this thesis can be summarized as:

- *Basics* – The fundamental and advanced concepts of mechanics, necessary to understand and implement rigid body dynamics simulation, are summarized.
- *Big Picture* – A modular overview of the simulation process and an overview of state of the art simulation methods are given.
- *Detailed View* – The important simulation methods are explained.
- *Implementation* – A design of a unified (simulation method independent) framework is suggested.

1.2 Chapter Overview

This thesis is divided into the following chapters:

- Chapter 2, "**Basics**", summarizes the basics of physics, which are necessary to simulate rigid bodies. Additionally the linear complementarity problem is introduced, which is a key problem in rigid body dynamics.
- Chapter 3, "**Simulation Overview**", shows a modular overview of the simulation process.

- Chapter 4, "**Constraint Solver**", explains how collisions, resting contact and other constraints, which restrict the motion of rigid bodies, are realized in various simulation methods.
- Chapter 5, "**Error Correction**", describes how a simulation can be made robust by adding error correction.
- Chapter 6, "**Other Simulation Methods**", lists related work in the field of rigid body dynamics that introduce other methods or ideas than those presented in this thesis.
- Chapter 7, "**A Unified Framework**", describes the design of a simulation method independent framework for interactive 3d applications.
- Chapter 8, "**Evaluation**", concludes the description of the implementation by showing several example scenarios that have been simulated. This chapter demonstrates what rigid body dynamics can be used for. It finishes with several insights gained during the implementation process.
- Chapter 9, "**Conclusion and Future Work**", summarizes the contributions of this work and lists issues that remain for future work.

2 Basics

"When I use a word, it means just what I choose it to mean - neither more nor less."

– Humpty Dumpty, in "Through the Looking Glass"

This chapter will tackle the first hurdle in building a rigid body simulation: the understanding of the fundamentals of physics and mathematics.

Not all books on the basics handle all important issues for simulation of rigid body dynamics. Most papers only present knowledge of a specific simulation method, often expecting that the basics are well understood. Learning from these sources leaves a novice in a vast landscape of inconsistent formalisms – making the introduction to this field a tedious task. This chapter presents the relevant terms and fundamental knowledge in a consistent manner, together with all formulas used in the implementations and with derivations, which are important for understanding.

First it will be shown how rotations can be parameterized. Then the relevant dynamics equations for particles will be derived. Next, systems of particles and rigid bodies as special systems of particles will be discussed. A rigid body is a system of particles in which the distance between the particles is constant – or let's say "rigid". The following section explains how systems of rigid bodies can be described. Different types of forces are discussed afterwards. The chapter closes with a crucial problem of mathematical programming: the linear complementarity problem.

A solid understanding of these concepts is important. A good introduction into classical mechanics in general is given in Goldstein et al. [GoPS02], or Kuypers [Kuyp03] for German readers. Literature especially for computer scientists is Baraff [Bara97a] and Eberly [Eber04].

The notation that is used in this thesis is summarized in Appendix A, "Notation".

2.1 Scope of this Work

This thesis deals with *rigid body dynamics*, a problem that has a variety of names in the literature: *rigid*

body simulation, *rigid body physics*, *dynamic simulation of multi-body systems*, *physically based modeling*, etc. Likewise a piece of software that deals with this problem is known as *physics engine*, *dynamics engine*, *dynamic simulation SDK*, etc. Confusion of notions makes it necessary to have a look at the important terms and their distinction. The definitions given here follow the definitions given in Encarta [Enca04] and Wikipedia [Wiki05].

Physics is the science of the natural world in the broadest sense. In more detail: *Physics* is the natural science of the inanimate world, its structure and its laws which are accessible via measurement, experimental verification and mathematical representation. It is difficult to clearly distinct physics from other natural sciences, like *biology*, *chemistry*, *geology*, etc. Sometimes *physics* is said to be the fundamental science because all other natural sciences deal with systems that obey the laws of physics.

Classical Physics may be divided into subfields like *acoustics*, *mechanics*, *thermodynamics*, *electricity*, *magnetism,* etc. *Modern physics* deals with *astrophysics*, *cosmology*, *quantum theory*, the *general* and *special theory of relativity,* and more.

Mechanics as a subfield of physics can be split into *classical mechanics* and *quantum mechanics*. This work is concerned with *classical mechanics,* which is the study of the motions of bodies and the forces that cause these motions. Classical mechanics not only deals with solid bodies, therefore it may be divided into *solid mechanics* and *fluid dynamics* (including *hydrodynamics*, *aerodynamics*, *pneumatics*, etc.). Classical mechanics may also be separated into *kinematics*, *statics* and *dynamics*.

Kinematics studies the motion of bodies without regard to its causes. That means, it studies the relation between position, speed, and acceleration without considering forces that cause the motion of bodies.

Statics is concerned with bodies in static equilibrium. A body is in *static equilibrium* if no force is acting on the body, or all forces cancel each other. Bodies in static equilibrium are at rest or moving at constant velocity. *Statics* is important in architectural and structural engineering to calculate quantities like stress and pressure.

Dynamics is the study of motion of bodies and the forces that cause and affect this motion. Another term for dynamics is *kinetics*. Some sources (like Goldstein et al. [GoPS02]) use the term *dynamics* synonymously with *classical mechanics*.

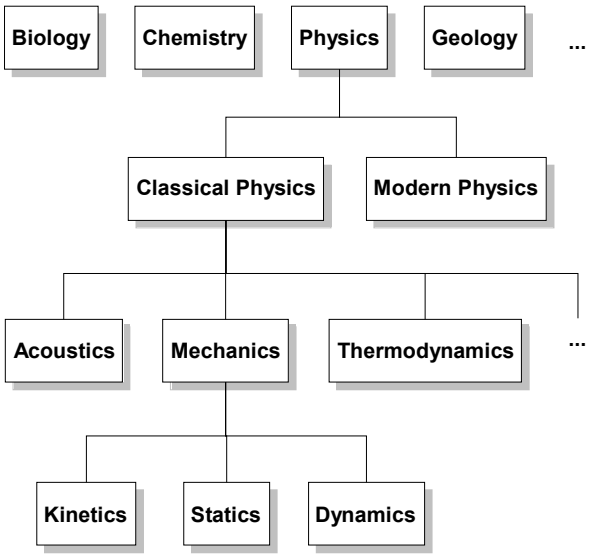

Figure 2.1: Term overview.

The relation of these terms is illustrated in Figure 2.1.

Classical mechanics is a vast field and this thesis deals with the dynamics of bodies that never change shape: *rigid bodies*. A rigid body is an idealization. Even if a big force is acting on the rigid body, it will not alter its form; it also never penetrates the area of another rigid body. In reality, no body is a rigid body. But this idealization leads to simplifications that allow us to simulate various scenarios in a computationally efficient manner.

The opposite of a rigid body is called *deformable body* or *soft body*. A force that is acting on a deformable body will lead to elastic or plastic deformation – the body will change its shape.

2.2 Rotations

Rotations play a critical role in rigid body motion. Therefore a mathematically suitable representation is needed. This section will present common methods to represent 3d rotations.

Given the center of rotation, a rotation in 2d has one degree of freedom and thus can be described by one scalar value: the rotation angle. A rotation in 3d has three degrees of freedom, so at least three coordinates are necessary to describe it.

Euler's theorem says: *The general displacement of a rigid body with one point fixed*[1] *is a rotation about some axis*. This theorem suggests that the rotation around a point can be described by the rotation axis and the amount of rotation, the rotation angle. This could be represented by a vector that points into the direction of the rotation axis with a magnitude equal to the rotation angle. This description is intuitive but not very useful to apply, combine, or interpolate rotations.

There are three other common methods to parameterize rotations: *Euler angles*, *rotation matrices*, and *quaternions*. In rigid body dynamics rotation matrices or quaternions are the weapons of choice.

2.2.1 Euler Angles[2]

The three degrees of freedom are parameterized by three angles that define three successive rotations in a specific order around defined axes. The angles of rotation are often called *pitch*, *roll,* and *yaw*[3].

The first rotation changes the coordinate system with the axes xyz to the new rotated coordinate system with the axes $x_{(1)}y_{(1)}z_{(1)}$. The second rotation changes the new coordinate system to $x_{(2)}y_{(2)}z_{(2)}$, and the third rotation transforms this coordinate system to the final coordinate system $x_{(3)}y_{(3)}z_{(3)}$.

The rotations can be performed around any fixed combination of these axes, for example: xyz (the first rotation is about x, the second about y, the third about z). Also intermediate axis may be chosen. For example, $zx_{(1)}z_{(2)}$ is frequently used in mechanics. The only restriction is that no two successive rotations can be around the same axis.

1 The fixed point could be the center of mass, for example.
2 Also called *Eulerian angles*.
3 Other names are *attitude*, *bank,* and *heading*.

Advantages and Shortcomings

Euler angle representation is intuitive, but it is sometimes difficult to setup an arbitrary rotation. The Euler angle representation of a rotation is not unique, which means, the same rotation can be parameterized with more than one Euler angle triple. It is also difficult to interpolate rotations with Euler angles. And there is a problem called *gimbal lock*. Gimbal lock is a characteristic problem where one degree of freedom is lost when setting up a sequence of rotations. This problem is discussed in Watt and Watt [WaWa92] in detail.

2.2.2 Rotation Matrices

A rotation in 3d can be described by a 3×3 matrix $\boldsymbol{R}$. A vector $\vec{v}$ can be rotated by multiplying with this matrix:

$$\vec{v}_{rotated} = \boldsymbol{R}\,\vec{v} \tag{2.1}$$

A 3×3 matrix has nine elements and therefore describes nine degrees of freedom. Additional conditions are necessary to restrict these to the three degrees of freedom of a rotation. Therefore, a rotation matrix has to be an *orthogonal matrix*. *Orthogonal matrix* means:

- The matrix is quadratic, the row vectors[1] are normalized, and they are all at right angles to each other.
- The row vectors and column vectors form an orthonormal basis.
- The inverse of the matrix is equal to the transpose of the matrix: $\boldsymbol{R}^{-1} = \boldsymbol{R}^{T}$
- The transpose multiplied by the inverse matrix is the identity matrix: $\boldsymbol{R}^{T}\boldsymbol{R} = \mathbf{1}_3$

All of these statements are equivalent. To describe a valid rotation the matrix has to be orthogonal and the determinant of the matrix has to be +1: $|\boldsymbol{R}| = +1$. More information on the properties of rotation matrices is given in Goldstein et al. [GoPS02].

During the simulation the matrix can loose the orthogonal property due to numerical inaccuracies – this is termed as *numerical drift*. A non-orthogonal matrix can cause shearing and scaling. To avoid this, the rotation matrix has to be re-orthogonalized regularly. The standard textbook method for this is the *Gram-Schmidt orthonormalization,* which is explained in most books on linear algebra (for example in Strang [Stra03]). Another method is described in Raible [Raib90]. How to set up a rotation matrix for an arbitrary rotation can be found in Foley et al. [FDFH05].

Advantages and Shortcoming

One advantage is that matrix operations are well-known, and software libraries supporting matrix operations are widely available. A disadvantage is that this representation of rotations is not intuitive. One can hardly tell which rotation is specified by just looking at the matrix. Regular re-orthogonalization is necessary because of numerical errors, and nine values are used to represent only three degrees of freedom. Another disadvantage – which is a concern in animation but not in dynamics – is that interpolating rotations is difficult with matrix representation.

1 This statement is also valid for the column vectors of an orthogonal matrix.

2.2.3 Quaternions

Quaternions were developed over 100 years ago by William Hamilton. They were introduced into computer graphics by Shoemake [Shoe85]. A quaternion is an extended complex value $q = w + \boldsymbol{i}\,x + \boldsymbol{j}\,y + \boldsymbol{k}\,z$ that has one real and three imaginary parts. w, x, y, z are real values and following equations hold for $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$:

$$\begin{aligned} \boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = -1, \\ \boldsymbol{ij} = \boldsymbol{k} = -\boldsymbol{ji} \end{aligned} \tag{2.2}$$

Here, another interpretation is used: A quaternion consists of one scalar w and an imaginary vector $\vec{v}$. It is typically written as a 4-dimensional vector :

$$\vec{q} = \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} w \\ \vec{v} \end{pmatrix} \tag{2.3}$$

Most interesting are unit quaternions. Unit quaternions have a magnitude of 1:

$$|\vec{q}| = \sqrt{w^2 + x^2 + y^2 + z^2} = 1 \,. \tag{2.4}$$

A quaternion describes a rotation by an angle α about an axis given by the unit vector $\vec{u}$. The corresponding unit quaternion is

$$\vec{q} = \begin{pmatrix} \cos(\frac{\alpha}{2}) \\ \sin(\frac{\alpha}{2})\vec{u} \end{pmatrix} \tag{2.5}$$

Unit quaternions can be used effectively to represent rotations in a simulation. A unit quaternion has to be renormalized if it loses its unit magnitude because of numerical errors. But since a quaternion has only four values, the drift of a unit quaternion is less than the drift of a rotation matrix. Thus, re-orthogonalization of matrices has to be done more often.

All quaternions used in the rest of this work are unit quaternions. The following equations may not be valid for non-unit quaternions.

Important Quaternion Operations

Quaternions have their own arithmetic. Most operations are defined on quaternions, for example addition, multiplication, logarithm, etc. This section covers the operations that are important for rigid body simulation. How to convert quaternions into rotation matrices and back, and more details can be found in Eberly [Eber02] and Shoemake [Shoe94].

Multiplying a quaternion with a scalar s is the same as a scalar-vector multiplication:

$$s\,\vec{q} = \begin{pmatrix} s\,w \\ s\,\vec{v} \end{pmatrix} \tag{2.6}$$

Multiplying two quaternions is defined as

$$\vec{q}_1 * \vec{q}_2 = \begin{pmatrix} w_1 w_2 - \vec{v}_1 \vec{v}_2 \\ \vec{v}_1 \times \vec{v}_2 + w_1 \vec{v}_2 + w_2 \vec{v}_1 \end{pmatrix}, \tag{2.7}$$

where * is used to denote a quaternion multiplication. Note that quaternion multiplication is non-commutative.

Sometimes a vector has to be multiplied by a quaternion. This is done by converting the vector into a quaternion with zero scalar value:

$$\vec{v} * \vec{q} = \begin{pmatrix} 0 \\ \vec{v} \end{pmatrix} * \vec{q} \tag{2.8}$$

Rotations can be combined by multiplying quaternions. If $\vec{q} = \vec{q}_1 * \vec{q}_2$, then $\vec{q}$ represents the same rotation as a rotation by $\vec{q}_2$ followed by $\vec{q}_1$. This order resembles the order of combined rotation matrices.

The inverse of a unit quaternion can be found by reversing the rotation axis:

$$\vec{q}^{-1} = \begin{pmatrix} w \\ -\vec{v} \end{pmatrix} \tag{2.9}$$

With these operations a vector $\vec{v}$ is rotated by a quaternion $\vec{q}$ with

$$\vec{v}_{rotated} = \vec{q} * \vec{v} * \vec{q}^{-1}. \tag{2.10}$$

This notation, which is commonly used, is a bit vague because the result of quaternion multiplication is another quaternion. But the scalar value will be zero again, and so the result can be interpreted as a simple vector.

Advantages and Shortcomings

Quaternions have only four elements, instead of not nine elements like rotation matrices. Quaternions have their own arithmetic and interpolation of rotations with quaternions is possible, but the math behind is not very intuitive. Regarding software implementations: Not all libraries support quaternion operations.

2.3 Newton's Laws of Motion

A discussion of rigid body physics should begin with Sir Isaac Newton's now-famous laws, which he stated in his *Philosophiae Naturalis Principia Mathematica*. They can be summarized like this:

- **Law I – Law of Inertia:**
  A body tends to remain at rest or continue to move in a straight line at constant velocity unless it is acted upon by an external force.
- **Law II – Fundamental Law of Dynamics:**
  The acceleration of an object equals the total force acting on the body, divided by its (constant) mass.
- **Law III – Law of Reciprocal Actions:**
  Whenever a body exerts force upon a second body, the second body exerts an equal opposite force upon the first body.

Especially the second law is of great importance for the study of dynamics.

2.4 Particles

A *particle* is the simplest object which allows studying motion. It is an object that has a constant mass *m*, a position in the world, and is able to move. A particle is also called *mass point*. It has no extent – it is an idealization. In the following, only systems with constant masses are examined.

2.4.1 Equations of Motion of Particles

The position of a particle can be described by a vector that varies over time: $\vec{x}(t)$. The velocity $\vec{v}(t)$ and the acceleration $\vec{a}(t)$ of the particle are defined as

$$\vec{v}(t)=\frac{d\vec{x}(t)}{dt}, \tag{2.11}$$

$$\vec{a}(t)=\frac{d\vec{v}(t)}{dt}=\frac{d^2\vec{x}(t)}{dt^2}. \tag{2.12}$$

Newton's second law of motion claims that forces acting on this particle cause acceleration:

$$\vec{F}(t)=m\vec{a}(t), \tag{2.13}$$

where $\vec{F}(t)$ is the *total force* (or *net force*), which is the vector sum of all forces acting on this particle:

$$\vec{F}(t)=\sum_i \vec{F}_i(t) \tag{2.14}$$

Equations (2.11 - 2.13) can be rewritten as

$$\begin{aligned}\dot{\vec{x}}(t)&=\vec{v}(t),\\ \dot{\vec{v}}(t)&=\frac{\vec{F}(t)}{m}.\end{aligned} \tag{2.15}$$

These are the *equations of motion* of a single particle. If the acting forces are known, the simulation can compute the translation of particles with these equations.

2.4.2 Angular Motion

So far we were concerned with linear motion in 3d, not rotations. The previous formulas resemble the known 2d formulas with the only difference that these vectors have three components. Rotational motion in 3d adds more complexity. To describe a rotation in 2d, only the point about which the rotation will happen has to be known. In contrast, in 3d a rotation can take place about an axis, which can have any position and orientation in the 3d world.

Angular Motion in Two Dimensions

Let $\Omega(t)$ be the angle of rotation around a point O. The angle in radian is defined as arc length $b(t)$ divided by radius r

$$\Omega(t)=\frac{b(t)}{r}. \tag{2.16}$$

The angular velocity in 2d is a scalar and defined as

$$\omega(t)=\frac{d\,\Omega(t)}{dt}. \tag{2.17}$$

Its relation to linear velocity is

$$v(t)=\omega(t)r=\frac{d\,\Omega(t)}{dt}r=\frac{d\,\Omega(t)r}{dt}=\frac{d\,b(t)}{dt}, \tag{2.18}$$

which gives the velocity along the circular path of the particle.

Angular Motion in Three Dimensions

In 3d angular velocity $\vec{\omega}(t)$ is a vector. It points into the direction of the axis of rotation and its magnitude is a measure for the velocity. The relation between linear velocity and angular velocity in 3d is

$$\vec{v}(t)=\vec{\omega}(t)\times\vec{r}(t), \tag{2.19}$$

where $\vec{r}(t)$ is the *radius vector* from O to the particle that is rotating around O (see Figure 2.2). Considering the properties of the vector cross product, the meaning is easier to spot: The resulting vector $\vec{v}(t)$ is normal to the rotation axis and $\vec{r}(t)$. The magnitude of $\vec{v}(t)$ is

$$|\vec{v}(t)|=|\vec{\omega}(t)||\vec{r}(t)|\sin\theta, \tag{2.20}$$

where θ is the angle between $\vec{\omega}(t)$ and $\vec{r}(t)$. $|\vec{r}(t)|\sin\theta$ is the part of $\vec{r}(t)$ that is normal to the axis of rotation and so the magnitude of $\vec{v}(t)$ is similar defined as in the 2d case in equation (2.18).

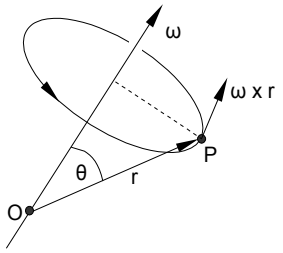

Figure 2.2: Angular motion of a particle.

Angular acceleration is defined as the vector $\vec{\alpha}(t)$ with

$$\vec{\alpha}(t)=\frac{d\,\vec{\omega}(t)}{dt}. \tag{2.21}$$

And its relation to linear acceleration is

$$\vec{a}(t)=\vec{\alpha}(t)\times\vec{r}(t). \tag{2.22}$$

The *torque* or *moment of force* (or simply *moment*) about a point $\vec{O}$ that is created by a force $\vec{F}(t)$ is

$$\vec{\tau}(t)=\vec{r}(t)\times\vec{F}(t)\,. \tag{2.23}$$

Torque $\vec{\tau}(t)$ is a vector that is normal to $\vec{r}(t)$ and $\vec{F}(t)$. It points in the direction of the axis of the rotation caused by this torque. Its magnitude is

$$|\vec{\tau}(t)|=|\vec{r}(t)||\vec{F}(t)|\sin\theta \tag{2.24}$$

$|\vec{r}(t)|$ is the distance to the rotation axis. $|\vec{F}(t)|\sin\theta$ is the part of the force that is normal to $\vec{r}(t)$. This resembles the well known 2d formula: *torque = force · lever arm*.

The analogy of linear and angular quantities is summarized in Appendix B, "Analogy Translation – Rotation".

2.4.3 Momentum and Impulse

The *linear momentum* $\vec{p}(t)$ is defined as the product of mass and velocity:

$$\vec{p}(t)=m\,\vec{v}(t) \tag{2.25}$$

A force that is acting on a particle causes a change in velocity, which also means a change of the linear momentum:

$$\vec{F}(t)=m\,\vec{a}(t)=\frac{m\,d\,\vec{v}(t)}{dt}=\frac{d\,m\,\vec{v}(t)}{dt}=\frac{d\,\vec{p}(t)}{dt} \tag{2.26}$$

The change in linear momentum is called *linear impulse*[1] $\vec{J}$. It is defined as

$$\vec{J}=\int\vec{F}\,dt=\int m\,\vec{a}\,dt=\int m\,\frac{d\,\vec{v}}{dt}\,dt=\int m\,d\,\vec{v}=\int d\,\vec{p} \tag{2.27}$$

This seems rather complicated. In simulation of rigid bodies it is most of the time sufficient to assume that a constant force $\vec{F}$ is acting over the time Δt. So $\vec{J}$ is simply

$$\vec{J}=\vec{F}\,\Delta t=m\,\Delta\vec{v}=\Delta p\,. \tag{2.28}$$

This can be viewed from another perspective: Applying an impulse changes the velocity and the linear momentum; the result is equivalent to a force that is acting over the time Δt.

Angular momentum $\vec{L}(t)$ is similarly to linear momentum. The relations to torque and linear momentum are

$$\vec{\tau}=\frac{d\,\vec{L}(t)}{dt}\,, \tag{2.29}$$

$$\vec{L}(t)=\vec{r}(t)\times\vec{p}(t)\,. \tag{2.30}$$

The rotational equivalent to linear impulse is *angular impulse* $\Delta\vec{L}$, although this quantity seems to be less important because it has no own variable letter in most textbooks on classical mechanics. The following equations still hold:

1 For German readers: "Momentum" in German literature is "Impuls". And the quantity "impulse" is "Kraftstoß" or "Impulsänderung" in German. Thus the English "impulse" and the German "Impuls" refer to different physical quantities!

$$\vec{\tau}\,\Delta t = I\,\Delta\vec{\omega} = \Delta\vec{L}\,. \tag{2.31}$$

The role of I will be explained in Section 2.5.3, "Mass Moment of Inertia".

Linear and angular momentum are two important quantities because both are conserved in the absence of external forces and torques:

- **Conservation theorem for the linear momentum:**
  If the total force $\vec{F}$ is zero, then the linear momentum $\vec{p}$ is conserved.
- **Conservation theorem for the angular momentum:**
  If the total torque $\vec{\tau}$ is zero, then the angular momentum $\vec{L}$ is conserved.

Momentum is even conserved in collisions. We will use this knowledge to derive formulas to compute the results of collisions. Impulses will be used by the simulation to alter the velocities of bodies.

2.4.4 Energy

If a force is acting on a particle and moves it from position *a* to position *b*, it does *work*. By definition this work is:

$$W_{ab} = \int_a^b \vec{F}\cdot d\,\vec{x} \tag{2.32}$$

From this definition we can derive other important quantities.

Energy is the capability to do work, or simply "stored work". Of special interest for rigid body simulation is *kinetic energy*. *Kinetic energy* of a particle is

$$E = \frac{m|\vec{v}|^2}{2}\,. \tag{2.33}$$

Kinetic energy is conserved in totally elastic collisions. In plastic collisions a part of the kinetic energy is consumed by deformation.

2.4.5 Systems of Particles

In a system of several unconnected particles the *total mass m* is the sum of all particle masses m_i:

$$m = \sum_i m_i \tag{2.34}$$

The *center of mass* $\vec{x_{CM}}(t)$ is the sum of the particle positions weighted in proportion to their mass:

$$\vec{x_{CM}}(t) = \frac{\sum_i m_i\vec{x}_i(t)}{\sum_i m_i} = \frac{\sum_i m_i\vec{x}_i(t)}{m} \tag{2.35}$$

The center of mass has outstanding relevance: If there are several external forces acting on the particles, then the center of mass moves as if the total external force is acting on the total mass of the system concentrated at the center of mass:

$$\begin{aligned}\vec{F}(t) &= \sum_i \vec{F}_i(t) = \sum_i m_i \vec{a}_i(t) = \sum_i m_i \frac{d^2 \vec{x}_i(t)}{dt^2} = \frac{d^2}{dt^2} \sum_i m_i \vec{x}_i(t) \\ &= \frac{d^2}{dt^2} m \vec{x}_{CM}(t) = m \vec{a}_{CM}(t),\end{aligned} \tag{2.36}$$

where $\vec{F}_i(t)$ is an external force acting on particle *i*.

The *total linear momentum* of the system is

$$\vec{p}(t) = \sum_i \vec{p}_i(t) = \sum_i m_i \vec{v}_i(t) = m \vec{v}_{CM}(t) . \tag{2.37}$$

Likewise, the *total angular momentum* is the vector sum of the angular momentums of all particles:

$$\vec{L}(t) = \sum_i \vec{L}_i(t) \tag{2.38}$$

2.5 Rigid Bodies

After having discussed systems of particles, this section cares about a special system of particles: a rigid body. A rigid body can be defined as a system of particles in which the distances between the particles are fixed and cannot vary over time. This is another definition than the more intuitive one given in Section 2.1, "Scope of this Work". In this definition a rigid body is viewed as a system of many, many mass elements, and this definition allows deriving the properties of a rigid body by starting with the properties of systems of particles.

What makes a rigid body more exciting and more complex than a single particle is the fact that it has an extent and can rotate around itself – a particle rotating around itself is not very fascinating. A rigid body has six degrees of freedom, three translational and three rotational, whereas a single particle has only three translational degrees of freedom.

A rigid body has the total constant mass *m*, which is the sum of all particle masses. This definition describes a rigid body that is made up of a discrete number of particles. In a continuous definition the rigid body is a volume filled with mass, and the total mass is calculated with a density function and the integral over the volume of the body:

$$m = \int_V \rho(\vec{x}) \, dV \tag{2.39}$$

$\rho(\vec{x})$ is the density function, which returns the density of the body at position $\vec{x}$.

In the continuous definition of the center of mass the volume integral replaces the sum:

$$\vec{x}_{CM} = \frac{\int_V \rho(\vec{x}) \vec{x} \, dV}{m} \tag{2.40}$$

2.5.1 World Space and Body Space

Two coordinate systems are of special interest: One coordinate system is fixed on the rigid body and rotates with the body. It is called *body coordinate system*, *body system*, *body space,* or *body frame*. The

other coordinate system is fixed in the world outside and is called *world coordinate system*, *world system*, *world space,* or *world frame*. Both coordinate systems are displayed in Figure 2.3.

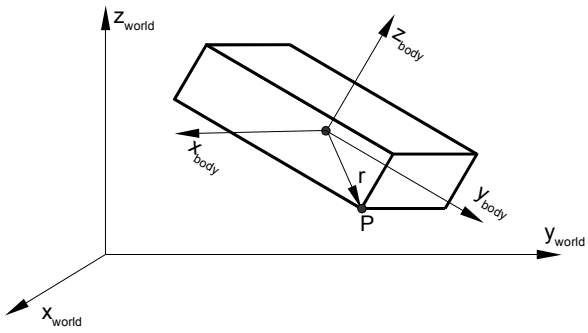

Figure 2.3: A rigid body with local body coordinate system in world space.

A coordinate system where Newton's second law of motion is valid is called an *inertial system*, *inertial frame,* or *Galilean system*. World space is an inertial system. Body space is no inertial system because the body could move at rising speed in the world; an observer sitting on the origin of body space would think she is sitting still and she would see other bodies that accelerate, even if no force is acting on them.

In the following considerations all vectors (and tensors) are given in world space. Exceptions will be explicitly marked. m denotes the total constant mass of the rigid body, and the center of mass is identical with the origin of body space.

How to Change Between World Space and Body Space

Given the body space coordinate of a point P fixed on the rigid body, what are the coordinates of this point in world space? The position of P in body space is given by the vector $\vec{r}_{body}$, which is constant in body space. The orientation of the rigid body can be described by a rotation matrix $\boldsymbol{R}(t)$ that rotates a vector from body to world space. The vector $\vec{r}$ in world space is given by

$$\vec{r}_{world}(t) = \boldsymbol{R}(t)\vec{r}_{body}. \tag{2.41}$$

$\vec{x}(t)$ is the position of the center of mass in world space. The appropriate position $\vec{x}_P(t)$ of point P in world space is

$$\vec{x}_P(t) = \vec{r}_{world}(t) + \vec{x}(t) = \boldsymbol{R}(t)\vec{r}_{body} + \vec{x}(t). \tag{2.42}$$

Converting from world space to body space is simple too because the rotation matrix $\boldsymbol{R}(t)$ is orthogonal, so there is an inverse matrix for $\boldsymbol{R}(t)$, which is the transpose:

$$\vec{r}_{body} = \boldsymbol{R}(t)^T \vec{r}_{world}(t) = \boldsymbol{R}(t)^T(\vec{x}_P(t) - \vec{x}(t)) \tag{2.43}$$

Foley et al. [FDFH05] discuss 3d transformations and changes of coordinate systems in detail.

2.5.2 Velocity and Acceleration

Chasles' theorem says that *the most general displacement of a rigid body is a translation plus a rotation*. This theorem gives the crucial hint that it is possible to split the problem of rigid body motion into two separate phases. One phase calculates the translational motion, the second one the rotational

motion. In other words: To get the total motion we compute the translational motion and add the angular motion.

It is best to place the origin of body space in the center of mass. In this way calculating the translational motion of the rigid body is the same problem as calculating the motion of the center of mass – the motion of a single particle with mass *m*.

In the following we will describe the state of a rigid body by the center of mass $\vec{x}(t)$, its orientation, which is given as a rotation matrix $\boldsymbol{R}(t)$ or a quaternion $\vec{q}(t)$, and its linear and angular velocity. Position and orientation are sometimes referred to as the *spatial variables* of the rigid body. The linear velocity of the center of mass is $\vec{v}(t)$, and $\vec{\omega}(t)$ is the angular velocity of the rigid body for rotations around the center of mass. These values can be seen in Figure 2.4.

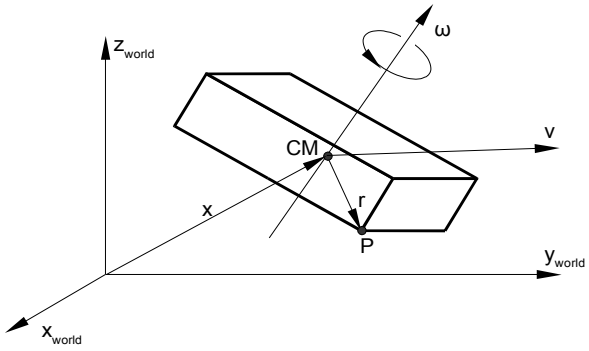

Figure 2.4: A rigid body with linear and angular velocity.

It is important to notice that in general $\vec{\omega}(t)$ is changing its direction continuously regarding world space and body space(!) even if no force is acting on the body.

The velocity of a body-fixed point *P* is the sum of the velocity of the center of mass and the velocity of the rotational motion:

$$\vec{v}_P(t)=\vec{v}(t)+\vec{\omega}(t)\times\vec{r}(t) \tag{2.44}$$

The acceleration of these points is

$$\vec{a}_P(t)=\vec{a}(t)+\vec{\alpha}(t)\times\vec{r}(t)+\vec{\omega}(t)\times(\vec{\omega}(t)\times\vec{r}(t)) . \tag{2.45}$$

The first term is the acceleration of the center of mass. The second term is the acceleration due to the angular acceleration about the center of mass; this is a vector in tangential direction to the rotation. The last term is the acceleration caused by the centripetal force. This is the force that keeps the particle at position $\vec{x}_P(t)$ in its orbit around the center of mass. Without it, the particle would change its distance to the center of mass, which is not possible in a rigid body. A derivation of this equation can be found in Baraff [Bara97b].

2.5.3 Mass Moment of Inertia

In 2d it is well known that angular momentum can be calculated as

$$L(t)=I\,\omega(t) \quad \text{with} \quad I=mr^2 , \tag{2.46}$$

where *I* is a scalar, called the *mass moment of inertia*.

Mass m describes "inertia" for a translational motion and I describes "inertia" for rotational motion. In 3d a single scalar quantity is not enough to describe the inertia behavior of a rigid body because its behavior depends on the shape of the body and the axis of rotation. For example, a long thin stick will behave different if it rotates around is longitudinal axis than if it rotates around a perpendicular axis.

The steps to derive the mass moment of inertia in 3d for a system of particles like a rigid body are sketched here:

First it is assumed that the body rotates around the origin of the coordinate system and $\vec{r}_i(t)=(x_i, y_i, z_i)^T$ is the position of particle i. The sum of all angular momentums of the particles is

$$\begin{aligned}\vec{L}=\sum_i \vec{L}_i=\sum_i \vec{r}_i\times\vec{p}_i=\sum_i \vec{r}_i\times(m\vec{v}_i)=\sum_i m_i\vec{r}_i\times(\vec{\omega}_i\times\vec{r}_i)= \\ =\sum_i m_i\begin{pmatrix}x_i\\y_i\\z_i\end{pmatrix}\times\left(\begin{pmatrix}\omega_x\\\omega_y\\\omega_z\end{pmatrix}\times\begin{pmatrix}x_i\\y_i\\z_i\end{pmatrix}\right).\end{aligned} \tag{2.47}$$

Expanding the cross-products leads to

$$\vec{L}=\sum_i m_i\begin{pmatrix}(y_i^2+z_i^2)\omega_x-x_i y_i\omega_y-x_i z_i\omega_z\\-y_i x_i\omega_x+(x_i^2+z_i^2)\omega_y-y_i z_i\omega_z\\-z_i x_i\omega_x-z_i y_i\omega_y+(x_i^2+y_i^2)\omega_z\end{pmatrix}. \tag{2.48}$$

In the next step the vector $\vec{\omega}$ is extracted, and the equation is reordered:

$$\begin{aligned}\vec{L}=&\begin{pmatrix}\sum_i m_i(y_i^2+z_i^2) & -\sum_i m_i y_i x_i & -\sum_i m_i z_i x_i\\-\sum_i m_i x_i y_i & \sum_i m_i(x_i^2+z_i^2) & -\sum_i m_i z_i y_i\\-\sum_i m_i x_i z_i & -\sum_i m_i y_i z_i & \sum_i m_i(x_i^2+y_i^2)\end{pmatrix}\begin{pmatrix}\omega_x\\\omega_y\\\omega_z\end{pmatrix}= \\ =&\begin{pmatrix}I_{xx} & -I_{yx} & -I_{zx}\\-I_{xy} & I_{yy} & -I_{zy}\\-I_{xz} & -I_{yz} & I_{zz}\end{pmatrix}\vec{\omega}=\boldsymbol{I}\,\vec{\omega}\end{aligned} \tag{2.49}$$

Equation (2.49) shows that the mass moment of inertia $\boldsymbol{I}$ is a 3×3 matrix. The diagonal elements are called *moment of inertia coefficients*:

$$\begin{aligned}I_{xx}&=\sum_i m_i(y_i^2+z_i^2),\\I_{yy}&=\sum_i m_i(x_i^2+z_i^2),\\I_{zz}&=\sum_i m_i(x_i^2+y_i^2)\end{aligned} \tag{2.50}$$

I_{xx} describes the inertia for a rotation about the x-axis. I_{yy} and I_{zz} do the same for rotation around the y-axis respectively the z-axis

The off-diagonal elements are called *products of inertia*:

$$\begin{aligned} I_{xy} &= \sum_i m_i x_i y_i, \\ I_{xz} &= \sum_i m_i x_i z_i, \\ I_{yz} &= \sum_i m_i y_i z_i \end{aligned} \tag{2.51}$$

These formulas are for a rigid body that is composed of discrete particles. For continuous bodies the discrete sum has to be replaced by an integral over the volume of the body, and the particle masses become the mass density:

$$\begin{aligned} I_{xx} &= \int_V \rho(\vec{x})(y_i^2 + z_i^2)\,dV, \\ I_{yy} &= \int_V \rho(\vec{x})(x_i^2 + z_i^2)\,dV, \\ I_{zz} &= \int_V \rho(\vec{x})(x_i^2 + y_i^2)\,dV, \\ I_{xy} &= \int_V \rho(\vec{x}) x_i y_i\,dV, \\ I_{xz} &= \int_V \rho(\vec{x}) x_i z_i\,dV, \\ I_{yz} &= \int_V \rho(\vec{x}) y_i z_i\,dV \end{aligned} \tag{2.52}$$

$\boldsymbol{I}$ is a matrix that describes the rigid body's inertia in rotational motion. This type of quantity is called a *tensor of second rank*. $\boldsymbol{I}$ depends on the material (mass density) and the shape of the body. It is constant if it is calculated in body space, but it is time-dependent in world space. Another important quality is: If a body is symmetric regarding an axis, then the corresponding product of inertia is zero.

Appendix C, "Mass Moment of Inertia", shows how to calculate the moment of inertia for simple shapes, like boxes and spheres.

Parallel Axis Theorem (Transfer of Axis Theorem)

$\boldsymbol{I}$ is always calculated for a rotation around a specific point. This point does not need to be the center of mass. The moment of inertia coefficients are smallest if the axis of rotation goes through the center of mass. But sometimes it is useful to calculate $\boldsymbol{I}$ for another center of rotation (as an example see Section 7.4.1, "Rigid Bodies"). The parallel axis theorem can be used to find the inertia tensor for another coordinate system, where axes are parallel to the original coordinate system.

Let's assume that the moment of inertia tensor for the center of mass (which lies in the origin of the coordinate system) is given and looks like this:

$$\boldsymbol{I}_{CM} = \begin{pmatrix} I_{CMxx} & -I_{CMyx} & -I_{CMzx} \\ -I_{CMxy} & I_{CMyy} & -I_{CMzy} \\ -I_{CMxz} & -I_{CMyz} & I_{CMzz} \end{pmatrix} \tag{2.53}$$

Then the moment of inertia tensor $\boldsymbol{I}$ for a rotation around point D is wanted. If the position of D is $\vec{d} = (d_x, d_y, d_z)^T$, the new components of the inertia tensor can be computed with

$$
\begin{aligned}
I_{xx} &= I_{CMxx} + m(d_y^2 + d_z^2), \\
I_{yy} &= I_{CMyy} + m(d_x^2 + d_z^2), \\
I_{zz} &= I_{CMzz} + m(d_x^2 + d_y^2), \\
I_{xy} &= I_{CMxy} + m d_x d_y, \\
I_{xz} &= I_{CMxz} + m d_x d_z, \\
I_{yz} &= I_{CMyz} + m d_y d_z.
\end{aligned}
\tag{2.54}
$$

Change of Coordinate System

The inertia matrix is constant in body space and time dependent in world space. The relation of the inertia matrix in body space and the corresponding inertia matrix in world space can be found with

$$
\begin{aligned}
\vec{L}_{world}(t) &= \boldsymbol{R}(t)\vec{L}_{body}(t) = \boldsymbol{R}(t)(\boldsymbol{I}_{body}\omega_{body}(t)) = \boldsymbol{R}(t)(\boldsymbol{I}_{body}(\boldsymbol{R}(t)^T\vec{\omega}_{world}(t))) = \\
&= (\boldsymbol{R}(t)\boldsymbol{I}_{\boldsymbol{body}}\boldsymbol{R}(t)^T)\vec{\omega}_{world}(t) = \boldsymbol{I}_{\boldsymbol{world}}(t)\vec{\omega}_{world}(t).
\end{aligned}
\tag{2.55}
$$

From this it can be seen that the inertia matrix in world space can be found by similarity transformation:

$$
\boldsymbol{I}_{\boldsymbol{world}}(t) = \boldsymbol{R}(t)\boldsymbol{I}_{\boldsymbol{body}}\boldsymbol{R}(t)^T \tag{2.56}
$$

This is practical for two reasons: First, the angular momentum in world space can be computed with the constant body space inertia matrix. Second, equation (2.56) shows how to compute the inertia matrix for a rotated coordinate system if the inertia matrix for the unrotated coordinate system is given.

Some simulation methods prefer to compute everything in body space because the inertia matrix is constant in body space, whereas the inertia tensor in world space has to be computed by similarity transformation in each new time step. This work uses world space coordinates since, when dealing with several rigid bodies, it is more intuitive to use a common coordinate system for all bodies.

Principal Axis Transformation

The inertia matrix is symmetric, so it is always possible to find a new rotated coordinate system in which the inertia matrix is a diagonal matrix (the products of inertia are all zero). The procedure of finding such a diagonal inertia matrix is known as *principal axis transformation,* and the axes of the rotated coordinate systems are the *principal axes*. The principal axes will correspond to the axes of symmetry. Even for arbitrary shaped bodies principal axes can be found so that all products of inertia are zero.

2.5.4 Time Derivatives of Rotation Matrices and Quaternions

The relation between the linear velocity $\vec{v}(t)$ of a rigid body and the position of the center of mass $\vec{x}(t)$ is known as $\dot{\vec{x}}(t) = \vec{v}(t)$. The relation between angular velocity and the orientation of the rigid body is not that easy to write down since the orientation is given as a rotation matrix $\boldsymbol{R}(t)$ or a quaternion $\vec{q}(t)$. This section shows how these quantities are related.

Time Derivative of a Rotation Matrix

Here we observe a rigid body that is positioned at the origin and can only make rotations ($\vec{x}(t)=\vec{0}$ and $\vec{v}(t)=\vec{0}$). The position of a point *P* on this rigid body is given by a vector $\vec{r}_{world}(t)$. The time derivative of *P*'s position is *P*'s velocity:

$$\vec{v}_P(t)=\frac{d\vec{r}_{world}(t)}{dt}=\vec{\omega}(t)\times\vec{r}_{world}(t) \tag{2.57}$$

The vector cross-product can be replaced by an equivalent operation: The multiplication with the *skew symmetric matrix* (also called *cross-product matrix*) built from $\vec{\omega}(t)=(\omega_x(t),\omega_y(t),\omega_z(t))^T$:

$$skew(\vec{\omega})=\begin{pmatrix}0 & -\omega_z & \omega_y\\ \omega_z & 0 & -\omega_x\\ -\omega_y & \omega_x & 0\end{pmatrix} \tag{2.58}$$

This is helpful for many situations because transforming equations with matrices is easier than transforming equations with vector cross-products. Thus, the cross product in equation (2.57) can be substituted with

$$\frac{d\vec{r}_{world}(t)}{dt}=skew(\omega(t))\vec{r}_{world}(t)\,. \tag{2.59}$$

Inserting equation (2.41) yields

$$\frac{d\boldsymbol{R}(t)\vec{r}_{body}}{dt}=skew(\vec{\omega}(t))\boldsymbol{R}(t)\vec{r}_{body}\,. \tag{2.60}$$

$\vec{r}_{body}$ is constant for a rigid body in body space, so it can be extracted from the derivative on the left hand side. Eliminating $\vec{r}_{body}$ from both sides leaves the demanded relation between the rotation matrix and the angular velocity:

$$\dot{\boldsymbol{R}}(t)=skew(\vec{\omega}(t))\boldsymbol{R}(t)\,. \tag{2.61}$$

Time Derivative of a Quaternion

The time derivative of quaternion $\vec{q}(t)$ is a bit longer to derive. The full derivation can be found in Baraff [Bara97b] and yields[1]

$$\dot{\vec{q}}(t)=\frac{1}{2}\vec{\omega}(t)*\vec{q}(t)\,. \tag{2.62}$$

* is a quaternion multiplication; the result is a quaternion. Again it would be very convenient if we could replace the quaternion product by a matrix operation, which would be easier to handle. Fortunately this can be done with a matrix $\boldsymbol{Q}$ built from the quaternion $\vec{q}$ like this:

$$\boldsymbol{Q}=\frac{1}{2}\begin{pmatrix}-x & -y & -z\\ w & z & -y\\ -z & w & x\\ y & -x & w\end{pmatrix}. \tag{2.63}$$

1 If $\vec{\omega}(t)$ is given in body space instead of world space, this equation changes to $\dot{\vec{q}}(t)=\frac{1}{2}\vec{q}(t)*\vec{\omega}(t)$.

Then equation (2.62) can be written with a matrix multiplication as

$$\dot{\vec{q}}(t) = \boldsymbol{Q}(t)\vec{\omega}(t)\,. \tag{2.64}$$

2.5.5 Equations of Motion of Rigid Bodies

One important ingredient for the rigid body's equations of motion is missing so far: The relation between torque and angular acceleration. The linear motion analogue is given by Newton's second law $\vec{F}(t) = m\,a(t)$. For angular motion this is more complicated since the inertia tensor is not constant:

$$\vec{\tau}(t) = \frac{d\,\vec{L}(t)}{dt} = \frac{d\,\boldsymbol{I}(t)\vec{\omega}(t)}{dt} = \frac{d\,\boldsymbol{I}(t)}{dt}\vec{\omega}(t) + \boldsymbol{I}(t)\frac{d\,\vec{\omega}(t)}{dt} \tag{2.65}$$

Baraff [Bara97b] and Goldstein et al. [GoPS02] show how to compute $\frac{d\,\boldsymbol{I}(t)}{dt}\vec{\omega}(t)$. The result is *Euler's Equation*:

$$\vec{\tau}(t) = \vec{\omega}(t) \times (\boldsymbol{I}(t)\vec{\omega}(t)) + \boldsymbol{I}(t)\frac{d\,\vec{\omega}(t)}{dt} \tag{2.66}$$

The term $\vec{\omega}(t) \times (\boldsymbol{I}(t)\vec{\omega}(t))$ is called *coriolis term* or *inertial torque*.

Now, combining linear and angular motion yields the so called *Newton-Euler equations* of rigid body motion:

$$\begin{aligned} \dot{\vec{x}} &= \vec{v} \\ \dot{\vec{v}} &= \frac{\vec{F}}{m} \\ \dot{\vec{q}} &= \boldsymbol{Q}\,\vec{\omega} \\ \dot{\vec{\omega}} &= \boldsymbol{I}^{-1}(\vec{\tau} - \vec{\omega} \times (\boldsymbol{I}\,\vec{\omega})) \end{aligned} \tag{2.67}$$

All quantities except m are time-dependent. These equations will be used to compute the motion of the rigid bodies.

2.5.6 Applying Forces and Impulses

As input for the equations of motion the total force on the center of mass and the torque for a rotation around the center of mass have to be known.

Section 2.6, "Forces", will show how forces due to gravity, drag, springs, and friction can be computed. Gravity and drag act directly on the center of mass. Springs and friction are forces that act on another point of the rigid body, for example on the point P in Figure 2.4. These forces can be split into a force on the center of mass and a torque.

If a force $\vec{F}_P(t)$ is acting on point P, the center of mass will be accelerated as if the force would act directly on the center of mass – this is one of the fabulous qualities of the center of mass. In addition, the force will cause a torque, which will start to rotate the body. The force $\vec{F}(t)$ and torque $\vec{\tau}(t)$ regarding the center of mass created by a force $\vec{F}_P(t)$ acting on point P are

$$\begin{aligned} \vec{F}(t) &= \vec{F}_P(t), \\ \vec{\tau}(t) &= r(t) \times \vec{F}_P(t). \end{aligned} \tag{2.68}$$

Forces that act only on the center of mass will not cause an angular acceleration.

Rigid body simulation often has to cope with constant forces and constant torques that act over a time Δt. This can be described by a linear and an angular impulse, as it was discussed in Section 2.4.3, "Momentum and Impulse". A linear impulse applied at the center of mass changes the linear velocity; an angular impulse changes the angular velocity. Impulses in the simulation can be used to instantly change the velocity. In reality this would not be possible because an instantaneous change in velocity would require an infinite force if Δt goes to zero.

A linear impulse $\vec{J}_P$ that is applied at a point P can be split into a linear impulse $\vec{J}$ and an angular impulse $\Delta \vec{L}$ regarding the center of mass:

$$\begin{aligned} \vec{J} &= \vec{J}_P, \\ \Delta \vec{L} &= r \times \vec{J}_P \end{aligned} \tag{2.69}$$

2.5.7 Energy

The *translational kinetic energy* of the center of mass is

$$E_{trans} = \frac{m|\vec{v}|^2}{2}. \tag{2.70}$$

The *rotational kinetic energy* of the rotational motion of the rigid body around the center of mass is

$$E_{rot} = \frac{\vec{\omega}^T \boldsymbol{I} \vec{\omega}}{2}. \tag{2.71}$$

Adding both gives the *total kinetic energy* of a rigid body:

$$E = E_{trans} + E_{rot} = \frac{m|\vec{v}|^2}{2} + \frac{\vec{\omega}^T \boldsymbol{I} \vec{\omega}}{2} \tag{2.72}$$

A derivation of this formula is given in Baraff [Bara97b].

2.5.8 Systems of Rigid Bodies

A simulation will have to handle several rigid bodies, and therefore it is convenient to combine all equations of motions of the single bodies. The notation in this chapter follows the notation suggested in Sauer and Schömer [SaSc98].

First the four equations of motion of a single body *i*

$$\begin{aligned}\dot{\vec{x}}_i &= \vec{v}_i \\ \dot{\vec{v}}_i &= \frac{\vec{F}_i}{m_i} \\ \dot{\vec{q}}_i &= \boldsymbol{Q}_i \vec{\omega}_i \\ \dot{\vec{\omega}}_i &= \boldsymbol{I}_i^{-1}(\vec{\tau}_i - \vec{\omega}_i \times (\boldsymbol{I}_i \vec{\omega}_i))\end{aligned} \tag{2.73}$$

will be combined into two equations. Therefore the positions and orientations are gathered in one 7-dimensional vector $\vec{s}_i$:

$$\vec{s}_i = \begin{pmatrix} \vec{x}_i \\ \vec{q}_i \end{pmatrix} \tag{2.74}$$

The velocities are combined to the 6-dimensional vector $\vec{u}_i$:

$$\vec{u}_i = \begin{pmatrix} \vec{v}_i \\ \vec{\omega}_i \end{pmatrix} \tag{2.75}$$

The forces, torques and coriolis terms are put into vector $\vec{f}_i$:

$$\vec{f}_i = \begin{pmatrix} \vec{F}_i \\ \vec{\tau}_i - \vec{\omega}_i \times (\boldsymbol{I}_i \vec{\omega}_i) \end{pmatrix} \tag{2.76}$$

Matrix $\boldsymbol{S}_i$ is a 7×6 matrix built from matrix $\boldsymbol{Q}_i$:

$$\boldsymbol{S}_i = \begin{pmatrix} \mathbf{1}_{3\times 3} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{Q}_i \end{pmatrix} \tag{2.77}$$

The mass properties of the rigid body are collected in a matrix $\boldsymbol{M}_i$:

$$\boldsymbol{M}_i = \begin{pmatrix} m & 0 & 0 & 0 & 0 & 0 \\ 0 & m & 0 & 0 & 0 & 0 \\ 0 & 0 & m & 0 & 0 & 0 \\ 0 & 0 & 0 & I_{xx} & -I_{xy} & -I_{xz} \\ 0 & 0 & 0 & -I_{xy} & I_{yy} & -I_{yz} \\ 0 & 0 & 0 & -I_{xz} & -I_{yz} & I_{zz} \end{pmatrix} = \begin{pmatrix} m\mathbf{1}_{3\times 3} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{I} \end{pmatrix} \tag{2.78}$$

After all this work the equations of motion for rigid body *i* can simply be written as

$$\begin{aligned}\dot{\vec{s}}_i &= \boldsymbol{S}_i \vec{u}_i, \\ \dot{\vec{u}}_i &= \boldsymbol{M}_i^{-1} \vec{f}_i.\end{aligned} \tag{2.79}$$

The position vector $\vec{s}_i$ has 7 elements and the velocity vector $\vec{u}_i$ 6 elements. Matrix $\boldsymbol{S}_i$ accomplishes the transition from two velocity vectors to one position vector and one quaternion.

This was the first step. Now the equations of motions of all n rigid bodies can be combined. Therefore the position and orientations are combined in a $7n$-dimensional *generalized position vector* $\vec{s}$. Likewise the velocities and forces are concatenated to the $6n$-dimensional *generalized velocity vector* $\vec{u}$ and the $6n$-dimensional *generalized force vector* $\vec{f}$:

$$\vec{s} = \begin{pmatrix} \vec{s}_1 \\ \vec{s}_2 \\ \vdots \\ \vec{s}_n \end{pmatrix}, \quad \vec{u} = \begin{pmatrix} \vec{u}_1 \\ \vec{u}_2 \\ \vdots \\ \vec{u}_n \end{pmatrix}, \quad \vec{f} = \begin{pmatrix} \vec{f}_1 \\ \vec{f}_2 \\ \vdots \\ \vec{f}_n \end{pmatrix} \tag{2.80}$$

The $\boldsymbol{S}_i$ matrices are collected in one $7n \times 6n$ matrix $\boldsymbol{S}$, and the mass matrices are gathered in the $6n \times 6n$ *generalized mass matrix* $\boldsymbol{M}$:

$$\boldsymbol{S} = \begin{pmatrix} \boldsymbol{S}_1 & \boldsymbol{0} & \cdots & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{S}_2 & \cdots & \boldsymbol{0} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{0} & \boldsymbol{0} & \cdots & \boldsymbol{S}_n \end{pmatrix}, \quad \boldsymbol{M} = \begin{pmatrix} \boldsymbol{M}_1 & \boldsymbol{0} & \cdots & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{M}_2 & \cdots & \boldsymbol{0} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{0} & \boldsymbol{0} & \cdots & \boldsymbol{M}_n \end{pmatrix} \tag{2.81}$$

Finally, the whole system of rigid body's equations of motion can be written in two short lines:

$$\begin{aligned} \dot{\vec{s}} &= \boldsymbol{S}\,\vec{u}\,, \\ \dot{\vec{u}} &= \boldsymbol{M}^{-1}\vec{f} \end{aligned} \tag{2.82}$$

These equations show well-known relations: The time derivative of position is velocity, and the time derivative of velocity is acceleration, which is force divided by mass. Matrix $\boldsymbol{S}$ has to be used to connect the 4-dimensional quaternions to 3-dimensional angular velocity vectors. And vector $\vec{f}$ additionally contains the coriolis terms $\vec{\omega} \times (\boldsymbol{I}\,\vec{\omega})$. Some works, like Catto [Catt05], simply ignore the coriolis terms. This is because the term is often small. When it is not small, it can lead to numerical instabilities. Neglecting the coriolis terms causes angular velocity to be conserved instead of angular momentum.

2.6 Forces

Forces cause acceleration and are the reason for motion. In the real world a lot of different forces appear. The most important ones are well studied and have to be handled in rigid body simulation.

2.6.1 Gravity

Unless the simulation handles a weightless environment, like outer space, *gravity* is a fundamental force to consider.

A body on earth is attracted to the ground with the force

$$\vec{F}_{gravity} = m\,\vec{g}\,, \tag{2.83}$$

where $\vec{g}$ is the *local acceleration of gravity* vector*;* it is simply pointing "down" and $|\vec{g}|$ is approximately $9{,}81\,m/s^2$ on earth. The gravity force acts on the center of gravity of the body, which is identical to the center of mass in uniform gravity fields.

2.6.2 Fluid Dynamic Drag

Fluid dynamic drag is a force that opposes the motion of a body that moves in a fluid (like air or water). It depends on a lot of factors: friction, relative speed, pressure variations, wave generation, shape of the body, etc. This topic is treated under several other names in the literature: Fluid friction, air friction, air resistance, viscosity, viscous drag, viscous damping, etc.

In a simple model the fluid dynamic drag force acts on the center of mass and depends on the fluid properties and the relative speed of the body in the fluid. For a low speed (laminar flow of the fluid) the force is

$$\vec{F}_D = -C_f \vec{v} \,. \tag{2.84}$$

For higher speed (turbulent flow of the fluid) the force is calculated as

$$\vec{F}_D = -C_f \vec{v}^2 \,. \tag{2.85}$$

The minus in the formulas shows that the force opposes the motion which is given by the velocity $\vec{v}$. C_f is the *drag coefficient* that describes the properties of the fluid and various other influences.

Fluid dynamic drag may also be considered in rotational movement, where it leads to a torque that opposes the angular motion:

$$\vec{\tau}_D = -C_f \vec{\omega} \quad \text{or} \quad \vec{\tau}_D = -C_f \vec{\omega}^2 \tag{2.86}$$

2.6.3 Springs and Dampers

Springs and dampers are structural elements that connect two particles (or points on a body) and apply equal opposite forces to both particles (or bodies). The force of a spring is defined by Hook's law:

$$|\vec{F}_S| = k_S |l - l_0| \tag{2.87}$$

k_S is the *spring constant*, l is the current stretched or compressed length of the spring, and l_0 is the rest length of the spring.

The damping force of a damper depends on the relative velocity of the two bodies and the *damping constant* k_D:

$$|\vec{F}_D| = k_D |\Delta v| \tag{2.88}$$

The spring force and the damping force are acting in the direction of the spring and the damper. The relative velocity has to be measured in the direction of the damper too. $\vec{E}$ is a unit vector that points from the position $\vec{x}_A$ of particle A to the position $\vec{x}_B$ of particle B:

$$\vec{E} = \frac{\vec{x}_B - \vec{x}_A}{|\vec{x}_B - \vec{x}_A|} \tag{2.89}$$

Then the relative velocity along the damper element is

$$\Delta v = (\vec{v}_B - \vec{v}_A)\, \vec{E} \,. \tag{2.90}$$

And the combined spring and damper force on point A is

$$\vec{F}_A = k_S(|\vec{x}_B - \vec{x}_A| - l_0)\vec{E} + k_D(\vec{v}_B - \vec{v}_A)\vec{E}\vec{E}\,. \tag{2.91}$$

Newton's third law of motion predicts that the force on point *B* is the opposite:

$$\vec{F}_B = -\vec{F}_A \tag{2.92}$$

2.6.4 Friction

If two touching surfaces move along each other, there is a force that resists this motion. This is *friction*. It is a contact force that is always parallel to the contacting surfaces. Friction may depend on the direction of movement on a surface, but here only isotropic friction is discussed, which is identical in all directions.

Normally friction is mathematically described with *Coulomb's law of friction*. In Coulomb's law of friction the *friction force* depends on the *normal force* that presses the surfaces together and the *coefficient of friction* μ. This coefficient describes the roughness of the surfaces. It cannot be found through calculation. It depends on the materials and has to be measured experimentally. For example, ice on ice has a low coefficient of friction, and it is different from ice on iron and iron on iron. A lot of other factors influence the coefficient of friction, for example if the surface is wet or dry. Most dry materials have a friction coefficient value between 0.1 and 0.6 (Beardmore [Bear05] is a good resource to look up coefficients of friction).

Figure 2.5 shows a contact between two bodies. The force that presses both bodies together is split into a component normal to the touching surfaces $\vec{F}_n$ and a component that acts tangential to the touching surfaces $\vec{F}_t$. The friction force $\vec{F}_{Friction}$ opposes the tangential force.

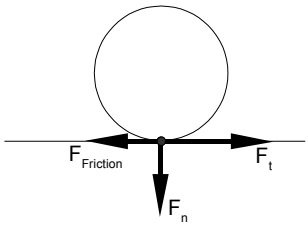

Figure 2.5: A frictional contact between two bodies in 2d.

In this model the size of the touching area is not important. Furthermore the friction is divided into *static friction* (also known as *stiction*) and *kinetic friction* (or *dynamic friction*).

Static friction

As long as the relative tangential motion of the bodies is zero, we speak of *static friction*. Static friction acts parallel to the surface and is as big as the part $\vec{F}_t$ of the external force that tries to move the surfaces parallel to each other. The maximum size of the static friction force is

$$|\vec{F}_{S,max}| = \mu_S |\vec{F}_n|\,, \tag{2.93}$$

where μ_S is the *coefficient of static friction*. If the external tangential force that tries to move the body is growing beyond $|\vec{F}_{S,max}|$, static friction switches to kinetic friction and the body starts to move.

Kinetic friction

If the relative tangential motion of the bodies is not zero, we speak of kinetic friction. The kinetic friction force opposes the movement and is

$$|\vec{F}_K| = \mu_K |\vec{F}_n| \ . \tag{2.94}$$

μ_K is the *coefficient of kinetic friction*. It is smaller than the coefficient of static friction and hence when the body starts to move, the friction force instantly changes from $\vec{F}_{S,max}$ to the smaller $\vec{F}_K$.

Kinetic friction can be further divided into *sliding friction* (objects are rubbing against each other) and *rolling friction* (one object rolls on the other object). The *coefficient of sliding kinetic friction* is greater than the *coefficient of rolling kinetic friction*.

The μ of kinetic friction is different from the μ of static friction, but often they are chosen to be equal, which simplifies friction calculation. For a more detailed discussion of friction see Gomez [Gome02].

Friction in Three Dimensions

Friction in 3d is a bit more complicated than in 2d. Figure 2.6 shows a contact between two bodies in 3d. The geometry at the contact point is described by a normal vector $\vec{n}$ and two tangential vectors $\vec{t}_x$ and $\vec{t}_y$.

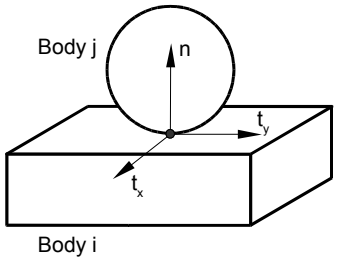

Figure 2.6: A contact between two bodies in 3d.

The external normal force can be written as $\vec{F}_n = \vec{n} f_n$.

$\vec{t}_x$ and $\vec{t}_y$ span the tangential plane. The friction force will lie in this tangential plane. It can be written as $\vec{F}_{Friction} = \vec{t}_x f_x + \vec{t}_y f_y$. The Coulomb model requires following inequality for the friction force:

$$f_x^2 + f_y^2 \le \mu^2 f_n^2 \tag{2.95}$$

This inequality can be interpreted geometrically as a cone and is called the *friction cone* (shown in Figure 2.7. The sum of the normal force and the friction force at the contact $\vec{F} = \vec{n} f_n + \vec{t}_x f_x + \vec{t}_y f_y$ has to lie within this cone to satisfy the Coulomb friction model.

In the case of kinetic friction, the direction of the friction force is opposite to the tangential velocity. In the static case the direction of the friction force is not known. The Coulomb friction model only requires that the friction force lies within the friction cone and is at least partially opposed to the tangential acceleration $\vec{a}_t = \vec{t}_x a_x + \vec{t}_y a_y$:

$$\vec{F}_{Friction} \vec{a}_t = f_x a_x + f_y a_y \le 0 \tag{2.96}$$

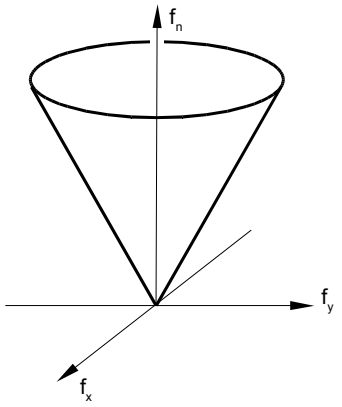

Figure 2.7: The friction cone.

Some simulation methods prefer to use impulses instead of forces. This is no problem, since multiplying both sides of the equations (2.93-2.96) with Δt turns the equations with forces into equations with impulses ($\vec{J} = \vec{F}\Delta t$). So friction impulses can be used instead of friction forces.

2.6.5 Other forces

The discussed forces are the ones that appear most frequently in interactive 3d applications, but others are not less interesting, for example: buoyancy (bodies swim in fluids), gravitation (mutual attraction of bodies), magnetism, etc. A detailed discussion of forces and applications, like projectiles, aircrafts, and cars, is given in Bourg [Bour02].

2.7 The Linear Complementarity Problem

A special problem that is encountered more than once in the field of rigid body dynamics is the *Linear Complementarity Problem* (LCP). LCPs are a special kind of *nonlinear programming problems*. A nonlinear programming problem is a problem of constraint optimization.

2.7.1 Problem Definition

Given a matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ and a vector $\vec{b} \in \mathbb{R}^n$, find $\vec{x} \in \mathbb{R}^n$ and $\vec{w} \in \mathbb{R}^n$ such that

$$\vec{w} = \boldsymbol{A}\vec{x} - \vec{b} \tag{2.97}$$

and for all i

$$w_i \geq 0\ ,\ x_i \geq 0 \text{ and} \tag{2.98}$$

$$w_i x_i = 0\ . \tag{2.99}$$

Equation (2.99) is the complementarity constraint, which can also be stated as

$$\vec{x}^T \vec{w} = \vec{0} \text{ or} \tag{2.100}$$

$$\vec{x}^T(\boldsymbol{A}\vec{x}-\vec{b})=\vec{0}\,. \tag{2.101}$$

This condition describes that either w_i or x_i has to be zero. Chapter 4, "Constraint Solver", will show several applications of the LCP problem and how it relates to "real" problems.

The equation (2.101) contains the term $\vec{x}^T\boldsymbol{A}\vec{x}$ which is quadratic in $\vec{x}$. The LCP can be formulated as a more general problem: a *quadratic programming problem* (QP). Thus a LCP can be solved by solving the corresponding QP.

2.7.2 Mixed LCP

In the traditional LCP the elements x_i are bound by $x_i \geq 0$. In a more general problem the elements x_i could be subject to specific bounds: $lo_i \leq x_i \leq hi_i$. This idea leads to a *Mixed Linear Complementarity Problem* (MLCP), which is:

Find $\vec{x}\in\mathbb{R}^n$ and $\vec{w}\in\mathbb{R}^n$ such that

$$\vec{w}=\boldsymbol{A}\vec{x}-\vec{b}\,, \tag{2.102}$$

$$\vec{lo}\leq\vec{x}\leq\vec{hi} \tag{2.103}$$

and for all i one of the three conditions below holds:

$$x_i=lo_i, w_i\geq 0\,, \tag{2.104}$$

$$x_i=hi_i, w_i\leq 0\,, \tag{2.105}$$

$$lo_i<x_i<hi_i, w_i=0 \tag{2.106}$$

$\vec{lo}$ is the vector of lower bounds, $\vec{hi}$ is the vector of upper bounds on $\vec{x}$. Figure 2.8 illustrates this problem. As long as x_i is within the bounds, w_i is zero. Only if x_i reaches a bound, w_i can be non-zero.

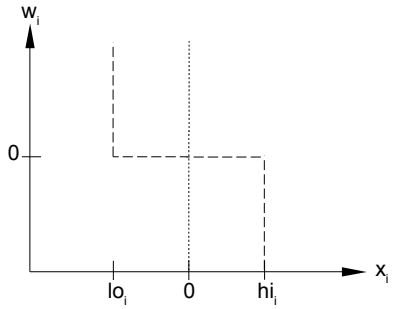

Figure 2.8: Illustration of the *i*'th variable of the MLCP.

A MLCP is a very flexible tool. With $\vec{lo}=\vec{0}$ and $\vec{hi}=\vec{\infty}$ the MLCP is the same as the traditional LCP. Setting the lower bound to $-\infty$ and the higher bound to ∞ removes all bounds on the unknown $\vec{x}$. This way the MLCP can be used to solve the linear set of equations: $\boldsymbol{A}\vec{x}=\vec{b}$. So the MLCP can solve equations with equalities and inequalities together.

2.7.3 Dependent Limits

Baraff [Bara94] introduced the idea that the limits of a variable can depend on the value of another variable. For example: The limits for x_i are lo_i and hi_i, and they could depend on x_j multiplied by a constants μ: $lo_i = -\mu x_j$ and $hi_i = +\mu x_j$. This is useful when calculating normal forces and friction forces of contacts with one MLCP as it will be done in Section 4.6.3, "Simultaneous Impulse-based Methods".

2.7.4 Solving the LCP

Determining if an LCP has a solution is NP-complete. Finding a solution to an LCP is NP-hard. In practice LCPs can be solved by efficient algorithms with exponential worst case behavior and with expected polynomial running time. These and other properties of LCPs are discussed in detail in Murty [Murt88] and Baraff [Bara92].

Murty also shows various methods to solve LCPs. Solution methods are often divided into *pivoting* (*direct*) *methods* and *iterative* (*splitting*) *methods*. Lacoursière [Laco03] compares several methods and states that there is no general solution method which is guaranteed to work for any given LCP – except for total enumeration, which has exponential complexity $O(2^n)$.

One of the first pivoting algorithms to solve LCPs was *Lemke's algorithm*, used in Baraff [Bara91]. Baraff [Bara94] describes *Dantzig's algorithm,* a pivoting algorithm to solve LCPs in the context of rigid body dynamics and how it can be extended to handle dependent limits.

Iterative solvers for sets of linear equations (see Press et al. [PTVF03]), like the *Jacobi method*, *Gauss-Seidel method*, *SOR* (s*uccessive over* r*elaxation*) *method*, or c*onjugate* g*radient method*, can be extended to solve MLCPs. Erleben [Erle04] and Catto [Catt05] describe in detail how an iterative solver for a MLCP can be implemented. Iterative algorithms have some advantages: They can stop early and return an approximate solution, whereas pivoting algorithms have no useful intermediate results. Iterative solvers seem to be the best solution for interactive 3d applications, where short execution time is preferred over accuracy.

For the implementation part of this thesis a MLCP solver based on Dantzig's method ([Bara94], [Smit05]), and an iterative solver Gauss-Seidel solver ([Erle04], [Catt05]) have been implemented.

3 Simulation Overview

"Computer Science is no more about computers than astronomy is about telescopes."

– E. W. Dijkstra

Dynamics simulation is a complex task. At first it is hard to overlook. Therefore the whole process is dissected into separate modules in this chapter. This allows studying and comparing the existing simulation methods. Understanding of the presented concepts is vital for the design and implementation of a rigid body simulation. Chapter 7, "A Unified Framework", shows how to implement these concepts.

3.1 Scope of Simulation

First of all it is crucial to know and decide which objects and physical phenomena will be modeled. The laws of physics presented in the previous chapter describe rigid body behavior and certain forces. They allow simulating frictional contacts, bodies connected with springs, or bodies connected by constraints. Even the fascinating behavior of a spinning top is the result of these laws. Though, several physical phenomena like general gravitation (mutual attraction of bodies), aerodynamics, or thermodynamics are not covered by these laws.

3.1.1 Rigid Bodies vs. Deformable Bodies

This work deals only with rigid bodies – not with deformable bodies. Non-rigid bodies introduce several difficulties: The body does not have a constant shape, and the center of mass is shifting, depending on the current mass distribution. And the mass moment of inertia changes with the shape of the body too. The shape of a deformable body cannot be described by a simple form, like

a box or a cylinder, any more. Due to shape deformation a body can also collide with itself, so self-collisions have to be detected and resolved. Furthermore inner forces, tension, deformation have to be calculated – a lot of additional work that we want to avoid.

The majority of objects in interactive 3d applications can be treated as rigid bodies. This leads to efficient and still visually appealing results. Examples are barrels, bricks, crates, rocks, furniture. Even complex objects can be modeled as connected systems of rigid bodies: humans, animals, chains, cupboards with doors that can swing open, etc.

The typical application for simulation of deformable bodies in interactive 3d applications is cloth simulation.

3.2 Simulation Modules

The simulation, as it is understood in this work, is part of an interactive 3d application. The modules of a simulation are illustrated in Figure 3.1.

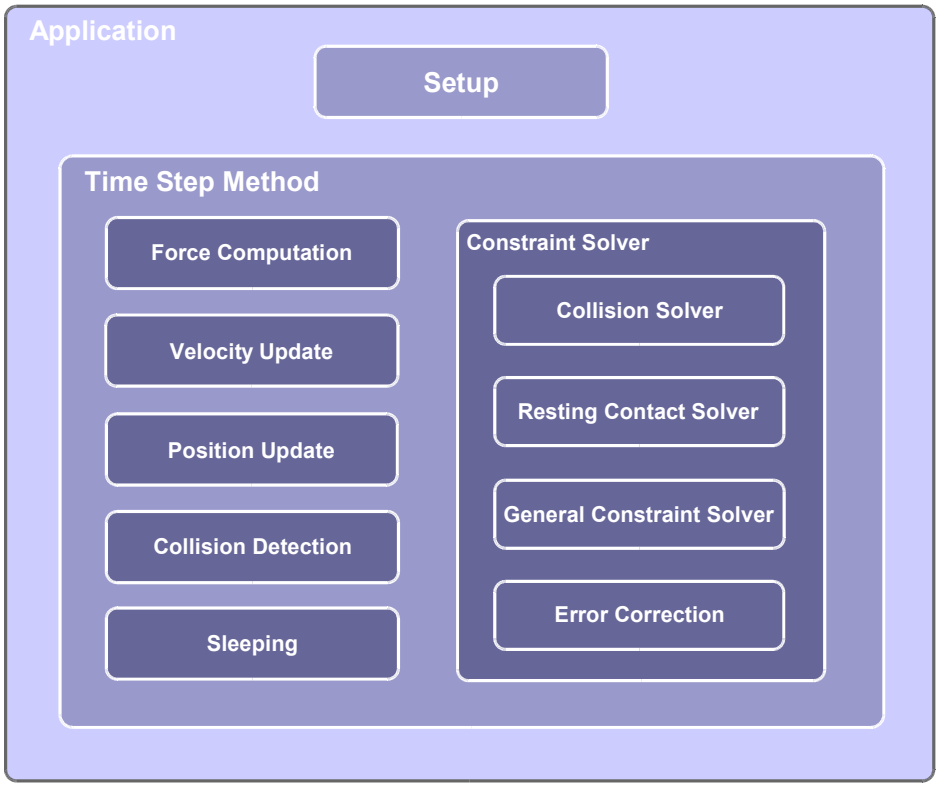

Figure 3.1: Modules of a rigid body simulation.

Let's sketch the principal process of a rigid body simulation in an application:

The first task that has to be done is the *setup*. The application has to set the simulation properties; it has to determine which forces, rigid bodies and constraints exist in the simulated world. For rigid bodies this involves calculating the mass properties and defining the shape. Instead of doing these steps when the application starts, they can be done in an offline process in advance. The predefined settings can be loaded when the application starts.

Then the application is running. It regularly calls the simulation and tells the simulation to advance by a time Δt - this is called a *time step*. After a time step the rigid bodies have changed positions and orientations. The application can read the new positions and use them, for example, to draw the bodies in their new positions. Usually the simulation makes a time step once per frame.

What happens in a time step? As illustrated in Figure 3.1 several tasks have to be done in each time step. These tasks will be discussed in this chapter. Here is a short summary in advance:

Module	***Description***
Force Computation	Calculates the total forces and torques that are acting on each body.
Velocity Update	Uses the total forces and torques to compute accelerations. From these the new velocities are found with numerical integration.
Position Update	Uses the velocities to determine the new positions and orientations with numerical integration.
Collision Detection	Finds all contacts between bodies.
Collision Solver	Handles colliding bodies: Computes post-collision velocities.
Resting Contact Solver	Computes forces between resting bodies.
General Constraint Solver	Handles constraints like joints, limits or motors.
Error Correction	Handles errors, like interpenetrations or violated constraints.
Sleeping	Deactivates bodies that have come to rest.

The *time step method* determines the sequence in which these modules are invoked.

At the end of each time step the simulation time is updated to $t_{new} = t_{old} + \Delta t$.

3.3 Force Computation

Job of the *force computation* is to sum up all forces and torques that act on the rigid bodies. For each body the total external force and total external torque due to forces like gravity, drag, springs, dampers have to be calculated. The interactive 3d application can also directly set additional forces to deliberately control the motion of bodies.

The force computation applies the laws presented in Section 2.6, "Forces". All forces are converted into forces and torques on the center of mass, as presented in Section 2.5.6, "Applying Forces and Impulses", and summed up.

Friction is different to the other forces because it is only computed when bodies come into contact. It is not handled in the force computation module, instead it is handled in the *collision solver* and *resting contact solver* modules. It is more difficult to compute – but very important as a few

examples show: A box on a horizontal plane in frictionless contact (without air drag) will slide endlessly if someone kicks it. A sphere sitting on an inclined plane without friction will slide down without rotating; with friction it will start to roll down.

Rolling friction and spinning friction are typically not handled in the simulation. This implies that a rolling sphere on a flat plane will roll forever. And a spinning top on a flat plane that touches the plane at only one point will spin forever. A viscous drag force is often used to counter this unrealistic behavior. Guendelman et al. [GuBF03] show how rolling and spinning friction can be added to the simulation.

3.4 Velocity and Position Update

The motion state of a rigid body *i* can be described by its position of the center of mass $\vec{x}_i(t)$, its orientation $\vec{q}_i(t)$ (or $\boldsymbol{R}_i(t)$), its linear velocity of the center of mass and $\vec{v}_i(t)$, and its angular velocity $\vec{\omega}_i(t)$. Instead of velocities, the linear momentum $\vec{p}_i(t)$ and the angular momentum $\vec{L}_i(t)$ could be used. Momentum has the nice quality of being constant if no external force and torque is acting. In contrast, angular velocity is generally not constant, even if no torque is acting on the body. So the current state of a body is given by a state vector

$$\begin{pmatrix} \vec{x}_i(t) \\ \vec{q}_i(t) \\ \vec{v}_i(t) \\ \vec{\omega}_i(t) \end{pmatrix} \text{ or } \begin{pmatrix} \vec{x}_i(t) \\ \vec{q}_i(t) \\ \vec{p}_i(t) \\ \vec{L}_i(t) \end{pmatrix}. \tag{3.1}$$

For the explanation in this section the first state vector with velocities is used because most readers are more familiar with velocity than with momentum. Section 2.5.8, "Systems of Rigid Bodies", showed that the positions and orientations of all bodies and the velocities of all bodies can be concatenated in two vectors, the generalized position vector $\vec{s}(t)$ and the generalized velocity vector $\vec{u}(t)$.

The simulation moves on in time by taking discrete time-steps of size Δt. The challenge of the *velocity* and *position update* is to take the state $\vec{u}(t)$ and $\vec{s}(t)$ at time t and compute the state $\vec{u}(t+\Delta t)$ and $\vec{s}(t+\Delta t)$ at time $t+\Delta t$. This is done using the forces and torques computed in the *force computation*.

The equations of motion of the system of rigid bodies are

$$\begin{aligned} \dot{\vec{s}} &= \boldsymbol{S}\,\vec{u} \\ \dot{\vec{u}} &= \boldsymbol{M}^{-1}\vec{f}\,. \end{aligned} \tag{3.2}$$

These are *ordinary differential equations* (ODEs). Solving for velocities and positions leads to these integrals:

$$\begin{aligned} \vec{u} &= \int \boldsymbol{M}^{-1}\vec{f}\,dt\,, \\ \vec{s} &= \int \boldsymbol{S}\,\vec{u}\,dt\,. \end{aligned} \tag{3.3}$$

All quantities in these equations depend on time *t*. In the following the time-dependency of the matrices will not be written explicitly to keep the equations readable. And to be exact, $\vec{f}$ depends

on the positions and velocities of bodies, for example, if they are connected by damped springs. Furthermore there are time dependent forces, $\vec{f}$ also depends explicitly on time; so

$$\vec{f} = \vec{f}(t, \vec{s}(t), \vec{u}(t)) . \tag{3.4}$$

No explicit formula for $\vec{f}(t)$ is known beforehand. Hence the integrals (3.3) cannot be solved analytically, they have to be computed with numerical integration. Due to these dependencies the update of a single body cannot be done independently for each body – the state of all bodies has to be considered.

3.4.1 Numerical Integration

This section shows several numerical integration methods to discuss the implications on the design of a rigid body simulation. A good introduction to numeric integration is given in Witkin and Baraff [WiBa97]. Press et al. [PTVF03] describe more numerical integration methods and how they can be implemented. A wealth of methods with special focus on rigid body simulation is presented in Eberly [Eber04]

Explicit Euler Integration

The simplest and most intuitive numerical integration method is *Euler's method* (*explicit Euler integration* also known as *forward Euler integration*). This method computes the new state like this:

$$\begin{aligned} \vec{u}(t+\Delta t) &= \vec{u}(t) + \Delta t\, \dot{\vec{u}}(t) = \vec{u}(t) + \Delta t\, \boldsymbol{M}^{-1} \vec{f}(t, \vec{s}(t), \vec{u}(t)) \\ \vec{s}(t+\Delta t) &= \vec{s}(t) + \Delta t\, \dot{\vec{s}}(t) = \vec{s}(t) + \Delta t\, \boldsymbol{S}\, \vec{u}(t) \end{aligned} \tag{3.5}$$

Explicit Euler integration uses the first derivation and evaluates it at the current time. This method is easy to understand and fast to compute, but the numerical error is large. Overall it is not the fastest method because the simulation has to take many small time steps Δt to avoid large errors.

Implicit Euler Integration

Another method is *implicit Euler integration* (also known as *backward Euler integration*), which evaluates the first time derivative in the future (at time $t + \Delta t$):

$$\begin{aligned} \vec{u}(t+\Delta t) &= \vec{u}(t) + \Delta t\, \dot{\vec{u}}(t+\Delta t) = \vec{u}(t) + \Delta t\, \boldsymbol{M}^{-1} \vec{f}(t+\Delta t, \vec{s}(t+\Delta t), \vec{u}(+\Delta t)) \\ \vec{s}(t+\Delta t) &= \vec{s}(t) + \Delta t\, \dot{\vec{s}}(t+\Delta t) = \vec{s}(t) + \Delta t\, \boldsymbol{S}\, \vec{u}(t+\Delta t) \end{aligned} \tag{3.6}$$

The problem is that in general $\vec{f}(t+\Delta t, \vec{s}(t+\Delta t), \vec{u}(+\Delta t))$ is not known because the future positions and velocities are not yet known. But they can be approximated, this involves solving a linear system and is described in Baraff [Bara97c].

Semi-Implicit Euler Integration

A simpler mixture of explicit and implicit Euler integration is often applied. Explicit Euler integration is used for the first equation, implicit Euler integration is done for the second integration with the newly computed velocity. This is called *semi-implicit Euler integration* (or *symplectic Euler*):

$$\begin{aligned}\vec{u}(t+\Delta t) &= \vec{u}(t)+\Delta t\,\dot{\vec{u}}(t) = \vec{u}(t)+\Delta t\,\boldsymbol{M}^{-1}\vec{f}(t,\vec{s}(t),\vec{u}(t))\\ \vec{s}(t+\Delta t) &= \vec{s}(t)+\Delta t\,\dot{\vec{s}}(t+\Delta t) = \vec{s}(t)+\Delta t\,\boldsymbol{S}\,\vec{u}(t+\Delta t)\end{aligned} \tag{3.7}$$

Other Integration Methods

The *midpoint method* is a method that computes an intermediate state at time $t+\Delta t/2$:

$$\begin{aligned}\vec{u}(t+\frac{\Delta t}{2}) &= \vec{u}(t)+\frac{\Delta t}{2}\dot{\vec{u}}(t) = \vec{u}(t)+\frac{\Delta t}{2}\boldsymbol{M}^{-1}\vec{f}(t,\vec{s}(t),\vec{u}(t))\\ \vec{s}(t+\frac{\Delta t}{2}) &= \vec{s}(t)+\frac{\Delta t}{2}\dot{\vec{s}}(t) = \vec{s}(t)+\frac{\Delta t}{2}\boldsymbol{S}\,\vec{u}(t)\\ \vec{u}(t+\Delta t) &= \vec{u}(t)+\Delta t\,\dot{\vec{u}}(t+\frac{\Delta t}{2}) = \vec{u}(t)+\Delta t\,\boldsymbol{M}^{-1}\vec{f}(t+\frac{\Delta t}{2},\vec{s}(t+\frac{\Delta t}{2}),\vec{u}(t+\frac{\Delta t}{2}))\\ \vec{s}(t+\Delta t) &= \vec{s}(t)+\Delta t\,\dot{\vec{s}}(t+\frac{\Delta t}{2}) = \vec{s}(t)+\Delta t\,\boldsymbol{S}\,\vec{u}(t+\frac{\Delta t}{2})\end{aligned} \tag{3.8}$$

This method evaluates the time derivatives twice. Other methods, like the *Runge-Kutta* methods, evaluate the time derivatives several times to get more accurate results.

Yet another popular method is *Verlet integration* [Verl67]. This method has gained popularity in the games industry due to Jakobsen [Jako01]. Kačić-Alesić et al [KaNB03] show how this scheme can be implemented for rigid body dynamics simulation. A lot of variations of this method exist, which are discussed in detail in Eberly [Eber04], for example: the *leap-frog method* and the *velocity-Verlet method.*

Better integration schemes may also involve adaptive control of the integration time step size.

3.4.2 Implementation Issues

Most rigid body simulations use explicit or semi-implicit Euler integration. The results are satisfying for most simulation scenarios. In scenarios with springs with high spring constants, explicit Euler integration leads to instability[1], because these are so called *stiff systems*. Implicit Euler integration can be used to solve stiff ODEs.

Applying explicit or semi-implicit Euler integration is easy. Force computation is done first, followed by the velocity and the position update. Velocity update and position update can be performed separately.

When using more sophisticated methods, like Runge-Kutta, the force computation has to be called several times by the numerical integration. It is also difficult to split up the velocity update and the position update.

In general, numerical integrators assume that the functions they are integrating are continuous. Collision handling and other actions in the simulation can cause discontinuities in the functions (like a collision causes an instantaneous change of velocity, as discussed below). If a more sophisticated integration scheme than the ones above is used, it should be restarted at the time of discontinuity with the new state as the initial state for integration [Bara89].

1 Instability in general means that the result of the integration is growing beyond all limits or is oscillating around the real solution.

3.5 Collision Detection

The modules *force computation*, *velocity update*, and *position update* are sufficient to compute the motion of the rigid bodies due to forces. But the bodies will move through each other, and bodies lying on the floor will sink through the ground. Therefore collisions need to be detected. The *collision detection* delivers the input for the modules that will handle the contacts between the bodies, the *collision solver* and the *resting contact solver*.

Two bodies are in *contact* if they touch or interpenetrate. Three cases have to be distinguished when two bodies meet: If their relative speed is zero (or within a given tolerance), then it is called a *resting contact*. If the bodies are approaching, it is termed a *colliding contact* or simply *collision*. A *separating contact* is present if the bodies are touching at the moment but the velocities indicate that they will move away from each other.

These terms are often mixed up. Care is taken to distinguish these terms in this thesis. The term collision detection is so common that it will be used in this thesis too, although all kinds of contacts have to be detected here, not only collisions.

Collision detection is a big, difficult, and important topic. It will not be treated here in detail. Instead this section defines what rigid body simulation requires from a collision detection system and which problems have to be considered.

3.5.1 Reduced Contact Set

If two bodies collide, there can be several simultaneous contact points, for example in edge-face or face-face contacts of two boxes. In these cases there are infinite contact points in the *contact region*, but a finite set is sufficient to compute the correct results [Bara89]. These finite set of contact points is called *reduced contact set*. Normally a reduced contact set is composed of points of the contact region that lie on edge-edge intersection. Figure 3.2 shows a contact region and a reduced contact set for a box-box contact example.

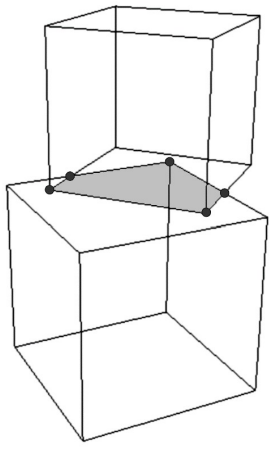

Figure 3.2: Reduced contact set with 5 contacts of a box-box contact

In general, there is not only one unique contact set that leads to correct results. Different contact sets may be chosen. Maybe it is even possible to choose a contact set with less contact points and

get correct simulation results. Moravánszky and Terdiman [MoTe04] discuss a method to reduce the number of contacts in the contact set to speed up the simulation.

3.5.2 Contact Information

For each single contact the simulation requires this information:

- a reference to the two colliding bodies,
- the position of the contact point,
- the normal vector, which is normal to the touching surfaces, and
- the penetration depth.

Contacts can be non-penetrating or penetrating. If a point of one body is inside the other body, it is called *penetrating contact* or *overlapping contact*. The *penetration depth* is the minimum length along the collision normal that the penetrating point has to move to leave the body. A contact with a penetration depth of zero (or a penetration depth within a given tolerance) is a *non-penetrating contact*.

3.5.3 Algorithms

Collision detection is usually divided into two phases. The task of the *broad phase* is to quickly decide which bodies may be in contact and which bodies cannot collide. The *narrow phase* then goes through the list of possibly colliding objects and determines the detailed contact information. Bergen [Berg04] gives a good collision detection overview.

Two very efficient algorithms for the narrow phase are Mirtich's *Voronoi Clip* (*V-Clip*) *algorithm* [Mirt98b] and the *Gilbert-Johnson-Keerthi* (*GJK*) *algorithm*[1] [GiJK88]. Both are explained in detail in Coutinho [Cout01]. Many algorithms for the narrow phase only return information about the pair of closest points of two bodies. This is not enough for the simulation as a whole contact set is needed, which often involves more than one contact. Mirtich [Mirt98c] shows how the V-Clip algorithm can be used to model contact regions.

Redon [Redo04] gives an overview on recent works on continuous collision detection methods that guarantee consistent simulations by computing the time of first contact.

3.6 Collisions, Contacts, and Constraints

In reality the motion of a rigid body is not free, it is restricted: Bodies may not pass through walls or other bodies, bodies might be connected to other bodies via joints, or bodies may even be restricted to follow a given path, like a bead on a wire. These restrictions on the motion are called *constraints*.

Collisions and resting contacts are special constraints. The resting contact constraint requires that

1 An enhanced version of the GJK algorithm was developed by Cameron [Came97].

a rigid body does not penetrate the space of another rigid body. Without this constraint the bodies would move through each other – this is called *unconstrained motion* in Baraff [Bara97a], in opposite to *constrained motion*. Collisions are like resting contacts, but additionally the bodies have to bounce off of each other.

In this work all other constraints than collisions and resting contacts are called *general constraints*. General constraints are versatile, nearly everything can be implemented.

Joints are constraints that connect two bodies. Bodies linked with joints are called *articulated*[1] *rigid body systems*. An example is given in Section 8.5, "Example: Ragdoll". Joint examples are a ball-and-socket joint and a hinge joint. The own body is a good subject to study joints. The hip joint, which connects the leg of a human to the rest of the body, can be approximated by a ball-and-socket joint. And the knee connects lower and upper leg via a hinge joint.

The range of motion at a joint is limited. The leg cannot be stretched much more than 180°. These constraints can be described by *limits*. An *angular limit* restricts the angular motion of two bodies. A *linear limit* restricts the linear motion. For example, a linear limit can be used to specify that two bodies cannot separate more than a specified distance – as if connected by an invisible rope.

But constraints can do much more: The orientation of a body can be forced to be fixed. The body may only be allowed to move in a flat plane or along a given path. And constraints can even create motion like a motor.

3.6.1 Constraint Solver

The *collision solver* module is responsible for treating collisions; the *resting contact solver* deals with resting contacts, and the *general constraint solver* handles all other constraints. These three modules are called *constraint solvers*.

Collisions, resting contacts, and general constraints can be treated uniformly as it will be shown in the next chapter. Nevertheless they are often treated separately. One reason is that collisions and resting contacts are detected by the collision detection, whereas general constraints are set by the user. A difference between collisions and resting contacts is that collisions occur at a certain time and are resolved instantly – at one moment the bodies are colliding and at the next moment they are separating. Resting contacts usually persist for a longer duration.

Some simulation methods use three separate constraint solvers for collisions, resting contact, and general constraints. Other methods use one solver for collisions and a second solver for resting contacts and general constraints. It is even possible to handle everything in a single constraint solver.

The task of the constraint solver is to "do something", so that the constraints are not violated. But what can a constraint solver perform to ensure constraints? The velocity and position update uses three inputs: positions (including orientations), velocities (linear and angular), and forces (including torques). These are three quantities with which a constraint solver can work.

Force/Acceleration-based Methods

Forces determine the acceleration. Acceleration of a body or a point on a body cannot be set directly, but forces can be applied at any point of the body. A constraints solver can compute an

1 articulated = consisting of segments held together by joints; (medical term) [FrDi05]

additional force per body that ensures constraints. This force is called *constraint force*. The constraint force has to be chosen such that the resulting acceleration of the body is consistent with the constraints.

Impulse/Velocity-based Methods

Another possibility is to directly change the velocity of the body. Directly changing the velocity is only possible for the center of mass – not for an arbitrary point on the rigid body. The velocity of a random point is the result of the combination of linear and angular velocity of a body.

An impulse is a tool to indirectly change the velocity of any point of the rigid body to any desired velocity. Linear impulse is $\vec{J} = m \Delta v$. Impulses can be applied at a point on the rigid body to cause any desired velocity change Δv at this position. A linear impulse that is applied at an arbitrary point can be split into a linear impulse and an angular impulse regarding the center of mass as shown in Section 2.5.6, "Applying Forces and Impulses".

Another way to look at the application of *constraint impulses* is to consider that applying a constant constraint force over a time Δt is similar to applying a constraint impulse directly since $\vec{J} = \vec{F} \Delta t$. A difference to a force-based solver is that an impulse-based solver immediately changes the velocity to a new value (directly or by applying an impulse). A constraint force of force-based solver has to go through the velocity and position update stage to have an effect.

Position-based Methods

The idea of the third method is simple: The bodies are not in the right positions? Then set the positions and orientations to the desired values.

The first difficulty of this method is that only the positions and orientations regarding the center of mass can be set directly. The position of another point of the rigid body cannot be set directly.

The second problem is that velocities should be corrected too. This is necessary for collisions: The bodies are approaching first, and after the collision solving the rigid bodies should have separating velocities. Position-based methods are not sufficient to handle collisions.

In resting contacts the velocity has to be re-adjusted as well. Only making position changes keeps the bodies in the right positions, but nothing hinders the velocity from growing into infinity in undesired directions – this would quickly cause numerical problems.

That is why position-based methods are mainly used additionally to other methods. Their main application is error correction.

Constraint Solvers in the Simulation

For now we can conclude that a constraint solver is a black box, which sets constraint forces, changes the velocities, or changes the positions. Chapter 4, "Constraint Solver", will open this black box and have a look at the inside.

The velocity update and position update may compute several intermediate states, for example, if a Runge-Kutta method is used. The force computation is called for each intermediate state. Likewise the collision detection and constraint solvers have to be called for each intermediate step. This sounds like a lot of work – and it is indeed because the collision detection and the constraint solvers are the two most time-consuming modules of a rigid body simulator.

Calling the collision detection and constraint solver for each intermediate step of the numerical integration is called *eager strategy* in Erleben [Erle04]. In the *moderate strategy* the collision detection is run once before the numerical integration. The contacts are reused in every call of the constraint solvers. In this thesis mainly the *lazy strategy* will be used: The collision detection and constraint solvers are only executed once before the velocity and position update; the constraint forces are assumed to be constant during one time step. The *lazy strategy* is not as accurate but faster and appropriate for interactive 3d applications.

3.7 Error Correction

During the simulation errors will occur. Errors appear in violations of constraints: Rigid bodies interpenetrate, or connected bodies drift apart and don't touch anymore. The task of the error correction is to remove these errors. This can be done by setting a force or changing the velocity, so that the error will be removed in the next time steps. Or the positions can be changed directly to the position enforced by the constraints.

The error correction can be implemented as another constraint solver. In contrast to the previously discussed constraint solvers this one *removes* errors. The other solvers only try to prevent that errors will occur. Error correction can be executed as a separate step, or be integrated with the other constraint solvers.

If the error correction uses a force/acceleration-based method or an impulse/velocity-based method, it will be called before the velocity and position update. This way the forces and velocities set by the error correction can take effect in the numerical integration. A position-based solver will usually be called after the position update to correct any remaining position errors.

3.8 Time Step Methods

And how do these modules fit together? This is determined by the *time step method*. The time step method calls the other modules in a certain order and advances the simulation time by Δt.

The time step size Δt will be set by the surrounding application. In real-time applications Δt will be proportional to the real time elapsed. Usually the application calls the simulation time step procedure once per frame and sets Δt to the time elapsed since the last frame was drawn.

3.8.1 Fixed Time Step Methods

Fixed time step methods proceed by making steps of size Δt and do not subdivide this time step. This implies that collisions are handled at the beginning of the time step and not at the exact collision time.

Explicit Time Step Method

The *explicit time step method* is perhaps the simplest fixed time step method. The exact time of

collision is not detected and in general penetrations will occur. This method proceeds like this:

```
ExplicitTimeStep(dt) {
    Collision Detection
    Collision Solver
    Resting Contact Solver
    General Constraint Solver
    Velocity and Position Update
    Error Correction
    t = t + dt
}
```

t is the current simulation time, and dt is the current time step Δt. Collision detection is called first to find the input for the collision and resting contact solver. Then the constraint solvers are executed. These could be separate solvers or only one single solver. Thereafter, the velocity and position update performs the numerical integration. The force computation is at least called once, but may be called several times. The error correction after the velocity and position update is optional. It could be included in the other constraint solvers. Finally the simulation time is advanced by Δt. Figure 3.3 illustrates the module order in an explicit time step. This figure also shows that the constraint solver may want to call the force computation too, depending on the applied constraint solver method.

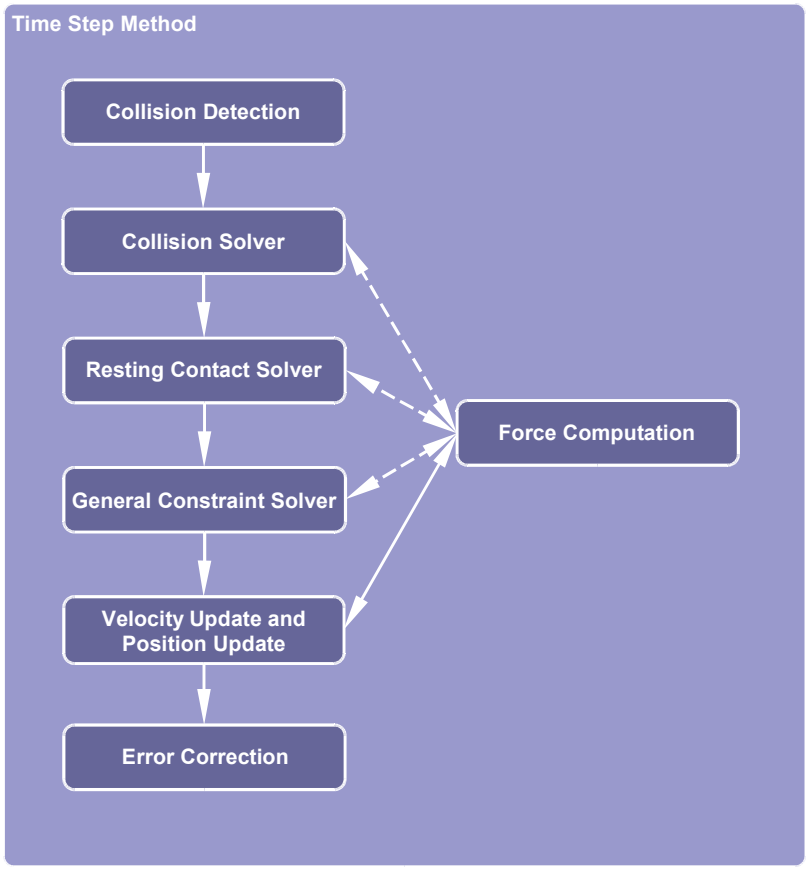

Figure 3.3: Explicit Time Step Method.

This method only detects all contacts at time t. But bodies could, for example, collide at time $t+\Delta t/2$. These contacts will not be handled and thus interpenetrations will occur at time $t+\Delta t$.

In the next time step these interpenetrations will be detected and treated as contacts with penetration errors.

(Semi-)Implicit Time Step Method

A *semi-implicit time step method* tries to detect in advance all contacts that may occur until the time $t + \Delta t$:

```
SemiImplicitTimeStep(dt) {
    Call Position Update to estimate positions
    Collision Detection with estimated positions
    Collision Solver
    Resting Contact Solver
    General Constraint Solver
    Velocity and Position Update
    Error Correction
    t = t + dt
}
```

First the positions at time $t + \Delta t$ are estimated by calling the position update module. Then collision detection uses the estimated positions to find contacts. The constraint solver takes these contacts to set constraint forces or change velocities. The rest is similar to the explicit time step method.

An *implicit time step method* estimates new velocities as the first step by numerical integration. This estimated velocity is used to estimate positions. The rest is the same as in the semi-implicit time step method.

Collision-Contact-Separated Time Step Method

Guendelman et al. [GuBF03] presented the idea that the update of velocities and positions can be separated. This leads to a time step method like this:

```
CollisionContactSeparatedTimeStep(dt) {
    Collision Detection
    Collision Solver
    Velocity Update
    Collision Detection
    Resting Contact Solver
    General Constraint Solver
    Position Update
    Error Correction
    t = t + dt
}
```

Guendelman et al. suggest detecting collisions at estimated positions as in the implicit time step method and the collision detection is additionally invoked after the velocity update. In their work the velocity and position update are easy to separate because they only use explicit Euler integration.

At first all detected contacts are treated as collisions in the collision solver. All non-separating contacts detected after the velocity update are considered to be resting contacts in the resting contact solver. This method does not need to distinguish colliding contacts and resting contacts in

the collision detection.

Shock Propagation

Many constraint solver methods do not give exact results, they just deliver approximations. Not all contacts are perfectly resolved. This is especially visible in body groups with a lot of constraints, such as a stack of boxes. In a stack contacts have to be resolved, and inaccurate constraint handling will result in unstable stacks.

Another new invention of Guendelman et al. is *shock propagation*. The idea is that the results of the constraint solvers will be more accurate if the constraints are resolved in the right order. For a stack the right order is bottom up, starting at the immovable ground plane.

In [GuBF03] shock propagation is done in an extra step after the resting contact solving in the collision-contact-separated time step. Erleben [Erle04] showed that shock propagation can be helpful in all simulations with iterative constraint solvers.

First a *contact graph* is built. The nodes of the graph are the rigid bodies, and the edges between the nodes are the contacts between the bodies. Then all bodies are sorted in levels, using a topological search starting at the immovable bodies. All bodies that touch the ground (or another immovable object) belong to level 1. All other bodies that touch immovable bodies or the bodies of level 1 belong to level 2, and so on. Bodies that rest on other objects are in higher levels.

In the shock propagation step the constraint solver is called for each level. In each call only constraints of bodies of the current level or lower levels are treated. After the constraint solver has handled a level, bodies of this level are changed to immovable bodies (mass and inertia are set to infinity and their inverses to zero). This way the bodies in higher levels will not push the bodies in lower levels down any more. The shock propagation ends after the constraint solver was called for the last level. Finally the bodies are changed to movable bodies again (the original mass and inertia are restored).

3.8.2 Adaptive Time Step Methods

Adaptive time step methods divide the time step Δt, in opposite to fixed time step methods. Adaptive time step methods can be split into *retroactive detection* and *conservative advancement (*Schmidl [Schm02]).

Retroactive Detection

In an exact simulation interpenetrations have to be avoided. Thus it is necessary to always find the exact time of collision. If the simulation finds interpenetrations, it has to step back to the time at which the first contact occurred. These methods are also called *backtracking methods*.

One example is the *bisection time step method* [Bara97b]. First a time step with size Δt is attempted. If no interpenetrations are found at the end of the time step, everything is ok. Otherwise the simulation is set back to the beginning of the time step. The time step size is cut into halves, and a step with $\Delta t/2$ is tried. Whenever interpenetrations are detected, the simulation is rewinded and the time step is cut into halves. If only non-penetrating contacts are found, collisions are handled and the simulation can try to make the next step forward.

The advantage of this scheme is that penetrations do not occur. But every time the simulation steps back, work is actually wasted. A method to avoid this was presented in Mirtich [Mirt00] and is discussed in Section 6.3, "Other Methods". Further problems are that this time step method is slower than an explicit time step method, and it is difficult to implement this method robustly because sometimes penetrations cannot be avoided (Erleben [Erle04]).

Baraff [Bara90] shows how the bisection search can be replaced by more sophisticated root finding methods that have faster convergence. The penetration depth, velocities, and accelerations of the bodies can be used to find better estimates for the collision time than cutting the time steps into halves.

Conservative Advancement

A conservative advancement method also tries to find the exact collision time. Instead of making a big step and then stepping back, this method makes only forward steps, but very carefully. Before each step an estimate for the next time of impact is made. This gives an upper bound for the time step. This method needs a sophisticated collision detection that can make good estimates for the next collision time.

A disadvantage of this method is that in some situations a lot of collisions in close temporal proximity will happen. Hence, the simulation is forced to take a lot of small time steps [Schm02].

3.8.3 Other Time Step Methods

Sauer and Schömer [SaSc98] use a *fix-point-iteration method*. This method is similar to the semi-implicit time step method, but the work of a single time step is repeated in a loop without advancing the simulation time t. Each iteration will find a better estimate for the new positions and the loop is repeated until the positions do not significantly differ from the positions computed in the last loop.

Erleben [Erle04] suggests a *fractioned time step*: The time step Δt is divided into two parts: $f \cdot \Delta t$ and $(1-f) \cdot \Delta t$ with $f \leq 1$. The first part of the time step is, for example, made with an explicit fixed time step method without error correction. And the second part of the time step is then performed with a shock-propagation method including error correction. This allows combining different time step methods.

3.9 Sleeping

The *sleeping* module has not been discussed yet. *Sleeping* is also known as *freezing* in Schmidl [Schm02] or *deactivating of objects*. It refers to the idea that sooner or later bodies will come to rest in the simulation. And if they do not move any more, they can be just fixed at their current place, and their state is not simulated any more. These fixed bodies are called *sleeping bodies*.

A sleeping method serves two purposes: Sleeping bodies can be ignored in many parts of the simulation. For example, two sleeping bodies do not need to be tested for contacts, and the states of sleeping bodies do not have to be numerically integrated. The second advantage is that sleeping bodies will not cause future errors because they don't move any more. So sleeping can save

computation time and make the simulation more stable.

The sleeping method has to detect resting bodies and set them to sleep. Sleeping bodies have to wake up on certain events, for example if another non-sleeping body is colliding with the sleeping body. Section 7.5.5, "Sleeping", will show how resting bodies can be detected and how sleeping can be implemented.

3.10 Conclusion

A running application that uses rigid body simulation calls the time step module in regular intervals. A time step method determines how the simulation is advanced by a time step Δt – especially in which order the simulation modules are invoked. There are many possible time step methods, too many to present all of them here. Advanced methods may call the collision detection several times in one time step. Care has to be taken to know which velocities and positions are used when: Some methods use the old velocities and positions in the collision detection and constraint solvers. Other methods use estimated velocities and positions in these modules. Interactive 3d applications should use fixed time step methods because they are faster, and in general small penetrations are tolerable.

The force computation modules delivers the currently acting forces. These values are the input for several other modules. The velocity update and the position update use numerical integration to change the state of the rigid bodies.

Contacts (found by the collision detection), user-defined constraints (like joints), and constraint errors (like interpenetrations), are handled by constraint solvers. Constraint solvers can use force-based, impulse-based, or position-based methods. It is possible to use dedicated solvers for collisions, resting contacts, general constraints, and errors. Though, methods that handle several tasks in one step are available.

4 Constraint Solver

"If I have seen farther than others, it is because I have stood on the shoulders of giants."

- Newton, referring to Galileo

This chapter reveals the inside of the constraint solver module. Several constraint solver methods are available and new methods are still being developed (see recent articles like [GuBF03] and [KaEP05]).

First it will be shown how equations can describe constraints, and several terms will be introduced. Important constraints will be detailed, for example: collisions, resting contacts, and some joints. After that, the important constraint solver methods will be discussed, which are divided into sequential methods and simultaneous methods.

4.1 Constraint Equations

This section gives a mathematical description, which helps to understand the topic and is crucial for simultaneous constraint solver methods.

4.1.1 Constraint Functions

To describe a constraint k a *constraint function* C_k is used. C_k is also called *behavior function* [Witk97a] or *deviation function* [BaBa88] because it describes the deviation from the valid position given by the constraint. C_k depends on time and the state of the rigid bodies:

$$C_k = C_k(t, \vec{s}(t), \vec{u}(t)) \tag{4.1}$$

A constraint can be expressed using the *constraint function* C_k in a *constraint equation*. If it is

independent of $\vec{u}$ and can be expressed in the form[1]

$$C_k(t, \vec{s}) = 0 , \tag{4.2}$$

then it is called a *holonomic constraint* (also called *equality constraint* [Bara93] or *bilateral constraint*). C_k reaches zero if the constraint is met. s independent holonomic constraints reduce the degrees of freedom of a system by s.

Examples for *nonholonomic constraints* are constraints, where the constraint function depends on velocities (and cannot be brought into holonomic form by integration)

$$C_k(t, \vec{s}, \vec{u}) = 0 , \tag{4.3}$$

or constraints that are expressed as inequalities (called *inequality constraints* [Bara93] or *unilateral constraints*)

$$C_k(t, \vec{s}) \geq 0 . \tag{4.4}$$

In this work we deal only with constraints functions that depend on the positions and rotations of two bodies. The constraint functions can be written as

$$C_k(\vec{s}_i, \vec{s}_j) , \text{ or} \tag{4.5}$$

$$C_k(\vec{x}_i, \vec{q}_i, \vec{x}_j, \vec{q}_j) . \tag{4.6}$$

Dynamics with constraints is difficult. Without constraints the equations of motion of all bodies are independent. But with constraints the equations of motion are no longer independent because the bodies' positions and rotations are connected by the constraint equations.

4.1.2 Time Derivatives of Constraint Functions

The time derivatives of the constraint function are important quantities. C_k depends on positions and rotations. The time derivative $\dot{C}_k$ will depend on velocities since $\dot{\vec{s}}$ is a linear function of $\vec{u}$. The second time derivative $\ddot{C}_k$ will depend on accelerations and indirectly on the acting forces and torques because of Newton's second law.

C_k is the deviation from the valid position. $\dot{C}_k$ describes the deviation velocity – the velocity with which the deviation from the valid position grows. And $\ddot{C}_k$ describes the acceleration of the deviation.

It is important to distinguish between *constraint function* and *constraint equation*. This work deals with equality constraints and inequality constraints. We call a constraint equations like $C_k = 0$ and $C_k \geq 0$ *position constraint equations*. Building the first time derivative yields the corresponding *velocity constraint equations* $\dot{C}_k = 0$ and $\dot{C}_k \geq 0$. The second order time derivative gives the *acceleration constraint equations* $\ddot{C}_k = 0$ and $\ddot{C}_k \geq 0$.

Let's think about the meaning of constraint functions and their time derivatives for equality constraints: If in one time step $C_k = 0$, then the constraint is satisfied at the moment. If $\dot{C}_k = 0$, then the constraint will be satisfied in the next time step too. But if $\dot{C}_k \neq 0$, the deviation will grow and the constraint will be violated in the next time step. If $C_k = 0$ and $\dot{C}_k = 0$, the constraint is

1 The time dependencies of the variables $\vec{s}$, $\vec{u}$, etc. will be left out in the equations for readability.

satisfied and will be satisfied in the next time step. $\dot{C}_k$ will stay zero as long as $\ddot{C}_k$ stays zero. With these considerations different strategies can be formed to deal with equality constraints:

- Idea 1: We try to keep $C_k=0$. If the constraint is violated, we compute a position change of all the bodies, so that the constraints are satisfied again.
- Idea 2: We assume that $C_k=0$ and try to keep $\dot{C}_k=0$. If $\dot{C}_k\neq 0$ in one time step, then we have to compute a change in the velocities (for example by applying impulses), so that $\dot{C}_k$ is corrected to zero.
- Idea 3: We assume that $C_k=0$ and $\dot{C}_k=0$ and try to keep $\ddot{C}_k=0$. This is done by calculating forces, called *constraint forces*. *Constraint forces* act in addition to external forces. Constraint forces have to ensure that $\ddot{C}_k$ stays zero.

These three ideas lead to the distinction in *position-based*, *impulse-based*, and *force-based* methods as discussed below. The same is true for inequality constraints, but there are a few differences. An inequality constraint is satisfied as long as $C_k\geq 0$. As long as $C_k>0$, $\dot{C}_k$ can take positive and negative values because the deviation is allowed to grow or shrink. Only if $C_k\leq 0$, $\dot{C}_k$ has to be zero or positive: $\dot{C}_k\geq 0$. This means that equality constraints have to be actively enforced all the time, whereas inequality constraints only have to be enforced if C_k reaches zero. If C_k is greater than zero, the constraint solver has nothing to do. Typical examples for inequality constraints are limits and contacts, which will be discussed in detail below. As long as the limit is not reached, the constraint solver does not have to care about the limit; and as long as the bodies do not touch, the constraint solver has nothing to do to ensure the non-penetration constraint.

4.1.3 Systems of Constraints

Normally the simulation has to deal with several constraints. This section shows how the constraint functions can be combined in one vector function.

A constraint function C_k is a function of the positions: $C_k=C_k(\vec{s}(t))$. $\dot{C}_k$ is a function of the velocities, which can be shown by the chain rule:

$$\dot{C}_k=\frac{d\,C_k}{dt}=\frac{\partial C_k}{\partial \vec{s}}\dot{\vec{s}}=\frac{\partial C_k}{\partial \vec{s}}\boldsymbol{Q}\vec{u}=\boldsymbol{J}_k\vec{u} \quad \text{with} \tag{4.7}$$

$$\boldsymbol{J}_k=\frac{\partial \vec{C}}{\partial \vec{s}}\boldsymbol{Q} \tag{4.8}$$

These equations illustrate that $\dot{C}_k$ is a linear combination of the components of the velocity vector $\vec{u}$. $\boldsymbol{J}_k$ is an $1\times 6n$ matrix, it can be split up into

$$\dot{C}_k=\boldsymbol{J}_{k1}\vec{u}_1+\boldsymbol{J}_{k2}\vec{u}_2+\cdots+\boldsymbol{J}_{kn}\vec{u}_n=\boldsymbol{J}_{ki}\vec{u}_i+\boldsymbol{J}_{kj}\vec{u}_j. \tag{4.9}$$

Each $\boldsymbol{J}_{kx}$ is a 1×6 matrix. Since all constraints only involve two bodies *i* and *j*, all matrices except $\boldsymbol{J}_{ki}$ and $\boldsymbol{J}_{kj}$ will be zero matrices[1].

1 A constraint may even involve only one body *i*, for example if the body's position is fixed relative to the simulation world. In this case $\boldsymbol{J}_{kj}$ is a zero matrix too. But this case can also be described as a two-body-constraint if the simulation world is represented by an abstract immovable rigid body, see 7.4.1 "Rigid Bodies".

Building the time derivative once more leads to

$$\ddot{C}_k = \boldsymbol{J}_k \dot{\vec{u}} + \dot{\boldsymbol{J}}_k \vec{u} = \boldsymbol{J}_k \dot{\vec{u}} + c_k \text{ with} \tag{4.10}$$

$$c_k = \dot{\boldsymbol{J}}_k \vec{u} \,. \tag{4.11}$$

Thus $\ddot{C}_k$ can be expressed as a matrix times the generalized acceleration vector and a term c_k.

A system normally deals with several constraint equations. A simple ball-and-socket joint removes three translational degrees of freedom and thus will already need three holonomic constraint equations.

All s constraint functions of a system can be combined into one constraint function vector $\vec{C}$. The matrices $\boldsymbol{J}_k$ and the scalars c_k are combined in the same way:

$$\vec{C} = \begin{pmatrix} C_1 \\ \vdots \\ C_s \end{pmatrix}, \quad \boldsymbol{J} = \begin{pmatrix} \boldsymbol{J}_1 \\ \vdots \\ \boldsymbol{J}_s \end{pmatrix} = \begin{pmatrix} \boldsymbol{J}_{11} & \boldsymbol{J}_{12} & \cdots & \boldsymbol{J}_{1n} \\ \vdots & \vdots & & \vdots \\ \boldsymbol{J}_{s1} & \boldsymbol{J}_{s2} & \cdots & \boldsymbol{J}_{sn} \end{pmatrix}, \text{ and } \quad \vec{c} = \begin{pmatrix} c_1 \\ \vdots \\ c_s \end{pmatrix}. \tag{4.12}$$

Thus the time derivatives of the constraint functions of the system are

$$\dot{\vec{C}} = \boldsymbol{J} \dot{\vec{u}} \text{ and} \tag{4.13}$$

$$\ddot{\vec{C}} = \boldsymbol{J} \dot{\vec{u}} + \dot{\vec{c}} \,. \tag{4.14}$$

$\boldsymbol{J}$ is called the *constraint Jacobian* of the system. Each constraint equation determines one row of this matrix.

4.2 Constraint Geometry

A constraint is often defined on two points, one point P_i on body *i* and one point P_j on body *j*, as shown in Figure 4.1. Most of the time these points are not identical with the centers of mass. These points are henceforth called *constraint anchor points*. The positions of the anchor points are given by two vectors $\vec{x}_{Pi}$ and $\vec{x}_{Pj}$. The distances of the points to the according centers of mass of the rigid bodies can be specified by two vectors $\vec{r}_i$ and $\vec{r}_j$.

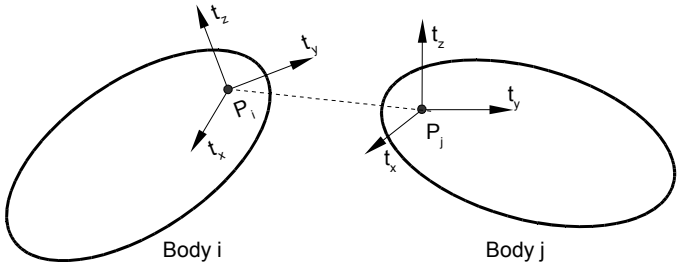

Figure 4.1: The constraint anchor points and local constraint axes.

A local coordinate system can be associated with each anchor point. The axes of each local coordinate systems can be described by three vectors $\vec{t}_x$, $\vec{t}_y$, and $\vec{t}_z$; these are called the *local constraint axes*. The orientation of this coordinate cross is fixed on the body and can be chosen by

the user – it does not have to be identical with the local coordinate system of the rigid body.

These definitions will help to discuss and implement constraints.

4.3 General Constraints

This section shows examples for various constraints. The constraint equations and time derivatives are described and brought into the general form

$$\dot{\vec{C}}_k = \boldsymbol{J}_{ki}\vec{u}_i + \boldsymbol{J}_{kj}\vec{u}_j \tag{4.15}$$

to show the structure of the Jacobian matrices. Many complex constraints require several constraint equations, so $\boldsymbol{J}_{ki}$ and $\boldsymbol{J}_{kj}$ are matrices with one row per constraint equation and 6 columns.

4.3.1 Ball-and-Socket Joints

A *ball-and-socket joint* (sometimes simply called *ball joint*) is an often needed joint. It forces the constraint anchor points to be at the same position, whereas the orientations are not restricted. The joint removes three translational degrees of freedom, hence it needs three holonomic constraint equations.

Its mathematical description is

$$\vec{C}_k = \vec{x}_{Pi} - \vec{x}_{Pj} = \vec{0}, \tag{4.16}$$

$$\dot{\vec{C}}_k = \dot{\vec{x}}_{Pi} - \dot{\vec{x}}_{Pj} = \vec{0}, \tag{4.17}$$

$$\ddot{\vec{C}}_k = \ddot{\vec{x}}_{Pi} - \ddot{\vec{x}}_{Pj} = \vec{0} \tag{4.18}$$

The velocity of a point is given by equations 2.44. This leads to

$$\dot{\vec{C}}_k = \vec{v}_i + \vec{\omega}_i \times \vec{r}_i - \vec{v}_j - \vec{\omega}_j \times \vec{r}_j = \vec{0} \tag{4.19}$$

Using $\vec{a}\times\vec{b} = -\vec{b}\times\vec{a}$ leads to

$$\begin{aligned}\dot{\vec{C}}_k &= \vec{v}_i - \vec{r}_i \times \vec{\omega}_i - \vec{v}_j + \vec{r}_j \times \vec{\omega}_j = \\ &= \begin{pmatrix}\mathbf{1}_3 & -skew(\vec{r}_i)\end{pmatrix}\begin{pmatrix}\vec{v}_i \\ \vec{\omega}_i\end{pmatrix} + \begin{pmatrix}-\mathbf{1}_3 & skew(\vec{r}_j)\end{pmatrix}\begin{pmatrix}\vec{v}_j \\ \vec{\omega}_j\end{pmatrix} = \vec{0}\end{aligned} \tag{4.20}$$

4.3.2 Hinge Joints

A *hinge joint* removes two more degrees of freedom than a ball-and-socket joint. Only one rotational degree of freedom is left. Hinge joints are often used to attach a wheel to a rotation axis.

A hinge joint forces the constraint anchor points to be at the same position, and the local constraint axes $\vec{t}_x$ have to be aligned. $\vec{t}_x$ is the hinge axis; only rotations around this axis are

allowed. The angular velocities of the bodies in directions orthogonal to $\vec{t}_x$ must be identical:

$$\begin{aligned} \vec{t}_y\vec{\omega}_i &= \vec{t}_y\vec{\omega}_j, \\ \vec{t}_z\vec{\omega}_i &= \vec{t}_z\vec{\omega}_j \end{aligned} \tag{4.21}$$

The local constraint axes $\vec{t}_x$ and $\vec{t}_y$ of body *i* or body *j* can be used in this equation. $\vec{t}_x$ of both bodies are identical. These conditions produce two more constraint equations in addition to the constraint equations of the ball-and-socket joint:

$$\dot{C}_k = \begin{pmatrix} \mathbf{1_3} & -skew(\vec{r}_i) \\ \vec{0}_3^T & \vec{t}_y^T \\ \vec{0}_3^T & \vec{t}_z^T \end{pmatrix} \begin{pmatrix} \vec{v}_i \\ \vec{\omega}_i \end{pmatrix} + \begin{pmatrix} \mathbf{-1_3} & skew(\vec{r}_j) \\ \vec{0}_3^T & -\vec{t}_y^T \\ \vec{0}_3^T & -\vec{t}_z^T \end{pmatrix} \begin{pmatrix} \vec{v}_j \\ \vec{\omega}_j \end{pmatrix} = \vec{0} \tag{4.22}$$

4.3.3 Linear Limits

Now, we will define a *linear limit* that forces the centers of mass of two bodies to be closer than a fixed distance *L*.

$$C_k = \frac{1}{2}(L^2 - (\vec{x}_i - \vec{x}_j)^2) \geq 0 \tag{4.23}$$

$$\dot{C}_k = -(\vec{x}_i - \vec{x}_j)(\vec{v}_i - \vec{v}_j) \geq 0\,. \tag{4.24}$$

With $\vec{d} = \vec{x}_j - \vec{x}_i$

$$\dot{C}_k = \begin{pmatrix} \vec{d}^T & \vec{0}_3^T \end{pmatrix} \begin{pmatrix} \vec{v}_i \\ \vec{\omega}_i \end{pmatrix} + \begin{pmatrix} -\vec{d}^T & \vec{0}_3^T \end{pmatrix} \begin{pmatrix} \vec{v}_j \\ \vec{\omega}_j \end{pmatrix} \geq 0\,. \tag{4.25}$$

The effect of this constraint is that the bodies behave as if they were connected by an invisible rope with length *L*. Of course, this linear limit can be extended to handle constraint anchor points that are not in the centers of mass.

This is an example of an inequality constraint. Equation (4.24) must only hold if the distance of the centers of mass reaches *L*.

4.3.4 Other Constraints

Shabana [Shab01], Smith [Smit04], and Erleben [Erle04] show how to formulate various other constraints. Making a new constraint in general involves these steps (Catto [Catt05]):

1. Determine the constraint equation with C_k as function of $\vec{s}_i$ and $\vec{s}_j$.
2. Differentiate the constraint equation with respect to time.
3. Identify the matrices $\boldsymbol{J}_{ki}$ and $\boldsymbol{J}_{kj}$.

Sometimes it is easy to directly form the velocity constraint equation with $\dot{C}_k$, as it was done in the hinge joint example.

The constraint equations and the structure of Jacobian matrices can be determined offline. The

simulation only needs to compute the elements of the Jacobian in every time step.

4.4 Collisions and Contacts

A contact constraint states that two bodies may not interpenetrate. The collision detection provides the contact points and the contact normal vectors. Figure 4.2 shows the constraint anchor points P_i and P_j, and the local constraint axes of body *i*, for a touching and a penetrating contact. Penetration as in Figure 4.2 (b) has to be avoided.

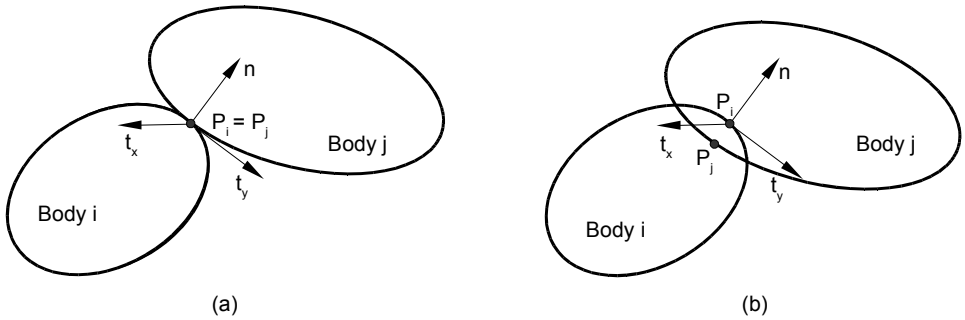

Figure 4.2: A touching contact (a) and a penetrating contact (b).

$\vec{t}_x$, $\vec{t}_y$ lie in the tangential plane of the contact, and $\vec{t}_z$ points in the normal direction. Here, it is called $\vec{n}$ instead of $\vec{t}_z$ to emphasis this meaning. We use the normal vector $\vec{n}$ that is pointing away from body *i* and formulate the contact constraint as

$$C_k = (\vec{x}_{Pj} - \vec{x}_{Pi})\,\vec{n} \geq 0\,. \tag{4.26}$$

C_k describes the separation distance[1] in the direction of $\vec{n}$. $\dot{C}_k$ can be found by differentiating with respect to time, where it has to be considered that the normal vector $\vec{n}$ depends on time *t* too since it is rotating with body *i*.

$$\dot{C}_k = (\vec{v}_{Pj} - \vec{v}_{Pi})\,\vec{n} + (\vec{x}_{Pj} - \vec{x}_{Pi})\,\vec{\omega}_i \times \vec{n} \geq 0\,. \tag{4.27}$$

We ignore the second term because we assume that the penetration at the contact is small. Using the vector identity $(\vec{a} \times \vec{b})\,\vec{c} = (\vec{b} \times \vec{c})\,\vec{a}$ this leaves

$$\begin{aligned} \dot{C}_k &= (\vec{v}_{Pj} - \vec{v}_{Pi})\,\vec{n} = (\vec{v}_j + \vec{\omega}_j \times \vec{r}_j - \vec{v}_i - \vec{\omega}_i \times \vec{r}_i)\,\vec{n} = \\ &= \begin{pmatrix} -\vec{n}^T & -(\vec{r}_i \times \vec{n})^T \end{pmatrix} \begin{pmatrix} \vec{v}_i \\ \vec{\omega}_i \end{pmatrix} + \begin{pmatrix} \vec{n}^T & (\vec{r}_j \times \vec{n})^T \end{pmatrix} \begin{pmatrix} \vec{v}_j \\ \vec{\omega}_j \end{pmatrix} \geq 0\,. \end{aligned} \tag{4.28}$$

$\dot{C}_k$ is the relative contact velocity in direction of $\vec{n}$ and therefore often denoted as $v_{rel,n}$.

Having found a contact *k*, C_k and $\dot{C}_k$ are examined to classify the contact:

- $C_k < 0$: This is a *penetrating contact*.
- $C_k = 0$: A *touching contact* without penetration.
- $C_k > 0$: This isn't a contact and will not be reported by the collision detection!

1 The penetration depth is given by $-C_k$.

- $\dot{C}_k < 0$: The bodies are moving towards each other. This is called *colliding contact* or simply *collision*.
- $\dot{C}_k = 0$: Bodies are touching and neither separating nor colliding. This is called *resting contact*.
- $\dot{C}_k > 0$: A *separating contact*, the bodies touch but are separating.

In practice, threshold limits are used to distinguish the contact type. For example, resting contacts are all contacts with $|\dot{C}_k| < RestingVelocityThreshold$. Similarly the collision detection will report all contacts within a small limit $C_k < CollisionEnvelope$.

4.4.1 Collisions

Collisions are a major issue for interaction in 3d applications. The difference to resting contacts is that bodies bounce off from each other and this has to be computed – but how?

What happens during a collision? Lets first imagine a simple example. One ball is floating motionless in space, and another ball is moving towards it. They start touching and both bodies start to deform. While they are deforming the first ball is slowing down and the second ball is gaining speed. This goes on until they have reached the same speed, and deformation is maximal. This is the end of the process in a perfectly plastic collision. In an elastic collision the game goes on. The deformation goes back, the first ball is still getting slower, and the second ball is still gaining speed. A perfectly elastic collision continues until the deformation is rebuilt. Then the first body's velocity was reduced, and the second body moves with higher velocity – they have exchanged momentum. The momentum of each body has changed, but the total momentum is conserved.

This all takes place in a small time interval. In the simulation this cannot be modeled since all the bodies are rigid and deformation is not possible. To treat collisions in rigid body simulations they are usually assumed to happen at one instant of time. The collision is detected, and the simulation changes the velocities to the valid post-collision velocities. This is a discrete change in velocity in contrast to the continuous change in reality.

To find the correct post-collision velocities following considerations are important: The total momentum is equal before and after the collision. For two colliding bodies *i* and *j*

$$\begin{aligned} \vec{p}_{i-} + \vec{p}_{j-} &= \vec{p}_{i+} + \vec{p}_{j+} \\ \vec{L}_{i-} + \vec{L}_{j-} &= \vec{L}_{i+} + \vec{L}_{j+} \end{aligned} . \tag{4.29}$$

Subscript '–' indicates a value before and subscript '+' a value after the collision. In terms of masses and velocities, this is

$$\begin{aligned} m_i \vec{v}_{i-} + m_j \vec{v}_{j-} &= m_i \vec{v}_{i+} + m_j \vec{v}_{j+} \\ \boldsymbol{I}_i \vec{\omega}_{i-} + \boldsymbol{I}_j \vec{\omega}_{j-} &= \boldsymbol{I}_i \vec{\omega}_{i+} + \boldsymbol{I}_j \vec{\omega}_{j+} \end{aligned} . \tag{4.30}$$

Each body experiences a change in momentum. In other words: Body *i* experiences an impulse and body *j* an opposite impulse of the same size:

$$\begin{aligned} \vec{J} &= m_i(\vec{v}_{i+} - \vec{v}_{i-}) = -m_j(\vec{v}_{j+} - \vec{v}_{j-}) \\ \Delta L &= \boldsymbol{I}_i(\vec{\omega}_{i+} - \vec{\omega}_{i-}) = -\boldsymbol{I}_j(\vec{\omega}_{j+} - \vec{\omega}_{j-}) \end{aligned} \tag{4.31}$$

The geometry of the colliding objects is important, because the velocity will only be changed along the line of action of the impact, which is given by the contact normal $\vec{n}$.

Collisions can be elastic. In a perfectly elastic collision the whole kinetic energy is conserved. The velocities along the line of action will be reflected. In an inelastic (or plastic) collision part of the kinetic energy is consumed by the deformation of the colliding bodies. In a perfectly plastic collision the two bodies will have the same velocity along the line of action after the collision.

Real collisions are between perfectly elastic and perfectly inelastic collisions. A quantity, which describes how elastic a collision is, is the *coefficient of restitution e*. It is the quotient of the separation velocity of the contact points after the collision and the approaching velocity of the contact points before the collision:

$$e = -\frac{(\vec{v}_{Pj+} - \vec{v}_{Pi+})\vec{n}}{(\vec{v}_{Pj-} - \vec{v}_{Pi-})\vec{n}} = -\frac{\vec{v}_{rel,+}\vec{n}}{\vec{v}_{rel,-}\vec{n}} = -\frac{v_{rel,n,+}}{v_{rel,n,-}} \tag{4.32}$$

This is known as *Newton's impact law* (also called *Newton's hypothesis* or *generalized Newton's rule*). e is 1 for perfectly elastic collisions because all the velocity is reflected along the line of action. e is 0 for perfectly plastic collisions because the separation velocity along the line of action is zero after the collision. In reality e lies between 0 and 1.

A constraint equation can be found for a collision as well. Since velocities are involved, the constraint is formulated directly with $\dot{C}_k$. The constraint function has to be defined on the post-collision velocities:

$$\dot{C}_k = (\vec{v}_{Pj+} - \vec{v}_{Pi+})\vec{n} \tag{4.33}$$

$\dot{C}_k$ is equal to $v_{rel,n,+}$. Formulating the constraint equation as

$$\dot{C}_k = (\vec{v}_{Pj+} - \vec{v}_{Pi+})\vec{n} \geq 0 \tag{4.34}$$

is equal to the resting contact constraint and describes a purely plastic collision, where the relative contact velocity after the collision is zero. In an elastic collision the relative post-collision velocity is defined by the coefficient of restitution. A collision constraint for plastic and elastic collisions can be written as

$$\dot{C}_k = v_{rel,n,+} + e\, v_{rel,n,-} \geq 0 \,. \tag{4.35}$$

The inequality is necessary to express that the relative normal velocity after collision can even be bigger than specified by the coefficient of restitution. This is because in a global view we have to take into account that there might be a third body that is colliding with one or both of these bodies, and this can add more velocity.

This velocity constraint has the new form $\dot{C}_k = \boldsymbol{J}_k \vec{u} + d_k = \boldsymbol{J}_{ki} \vec{u}_i + \boldsymbol{J}_{kj} \vec{u}_j + d_k \geq 0$:

$$\begin{aligned} \dot{C}_k &= (\vec{v}_{Pj+} - \vec{v}_{Pi+})\vec{n} + e\, v_{rel,n,-} = (\vec{v}_{j+} + \vec{\omega}_{j+} \times \vec{r}_j - \vec{v}_{i+} - \vec{\omega}_{i+} \times \vec{r}_i)\vec{n} + e\, v_{rel,n,-} = \\ &= \begin{pmatrix} -\vec{n}^T & -(\vec{r}_i \times \vec{n})^T \end{pmatrix} \begin{pmatrix} \vec{v}_{i+} \\ \vec{\omega}_{i+} \end{pmatrix} + \begin{pmatrix} \vec{n}^T & (\vec{r}_j \times \vec{n})^T \end{pmatrix} \begin{pmatrix} \vec{v}_{j+} \\ \vec{\omega}_{j+} \end{pmatrix} + e\, v_{rel,n,-} \geq 0 \end{aligned} \tag{4.36}$$

Collision Laws

In addition to Newton's impact law, we assume that Coloumb's friction is acting during a collision. In Coloumb's assumptions about friction, the friction force is always directly opposed to the

relative tangent velocity. This assumption leads to visually satisfying results in the simulation, but a microscopic scale observation of the contact can show that the direction of the friction force can change direction during the impact (see Mirtich [Mirt96b]). To model collisions more realistically, other collision laws exist (see Chatterjee and Ruina [ChRu98]).

Collisions and Resting Contacts in the Simulation

Although collisions and resting contacts can be treated similarly to all other constraints, they deserve special attention for several reasons: Contacts are detected by the collision detection and not set by the user. The active contact constraints change from time step to time step and additionally involve friction. Some difficult situations (like stacks of rigid bodies) should be handled very carefully.

4.5 Sequential Methods

Two bodies can collide and have multiple simultaneous collision points. It is also possible that multiple bodies collide at the same time. Various constraints may be active at the same time too. In general, several simultaneous constraints have to be handled in each time step. Sequential methods treat one constraint after the other using only local information.

4.5.1 Sequential Force-Based Methods

A relatively simple method that handles constraints locally, one after the other, and uses forces is the *penalty method*. This method is used in several works, for example: Witkin et a. [WiFB87], Platt and Barr [PlBa88], [TPBF87], or Kačić-Alesić et al. [KaNB03].

The basic idea is, if a point is in an invalid position, then a force is acting that tries to push the point back to the valid position. This correcting force can be imagined as a rubber band or a spring that tries to move the body back to the valid positions.

The constraint function is used as a potential to define an energy function. The energy can be formulated like this for example:

$$E = \frac{k_S}{2} \vec{C}\,\vec{C}\,. \tag{4.37}$$

The energy is zero if the constraint is met; the energy is growing if the deviation from the valid position is growing. This is similar to the elastic energy of a spring (with rest length of zero):

$$E_{elastic} = \frac{k_S}{2} l^2\,, \tag{4.38}$$

where l is the current length of the spring.

A penalty force is trying to minimize the energy and is acting in the direction of the maximum descent of E which is given by the gradient of E:

$$\vec{F_{Penalty}} = -\nabla E = -\begin{pmatrix} \frac{\partial E}{\partial x} \\ \frac{\partial E}{\partial y} \\ \frac{\partial E}{\partial z} \end{pmatrix} \tag{4.39}$$

This results in a penalty force that is like a spring force. Damping can be added to minimize oscillation. Thus, this penalty force is equal to a spring-damper element that is placed between the constraint anchor points as illustrated in Figure 4.3.

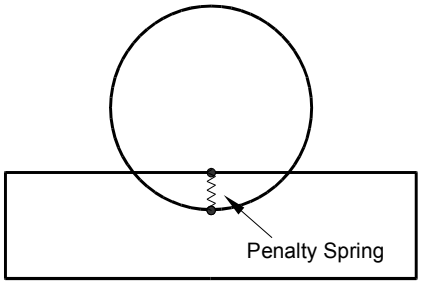

Figure 4.3: A penetrating contact with a penalty spring.

Adam [Adam03] uses springs to connect the limbs of a ragdoll. Resting contact can be resolved with penalty springs like this: At the beginning of the contact and penetration a spring is inserted between the collision points of each body. The spring is removed if the bodies are separating.

Colliding contact can be approximated similarly. This way the spring models the elastic deformation of the body and the force due to the elastic deformation. Moore and Wilhelms [MoWi88] use two different penalty springs to model a collision: During the contact, one penalty spring is active while the bodies are approaching and has the spring constant $k_{approach}$. Another penalty spring is active while the bodies are separating and has the spring constant $k_{separate} = e\,k_{approach}$ (where e is the coefficient of restitution). Consequently, in an inelastic contact no spring force is acting during the separation phase. Whereas a spring force is pushing the bodies apart in the separation phase of an elastic collision.

The advantage of this method is its simplicity. It automatically corrects deviations from valid positions and thus can be used for error correction too. The disadvantage is that an appropriate spring constant has to be chosen. This constant depends on the properties of the current object in the scene (for example coefficient of restitution and mass). So this parameter has to be adapted for each simulated scenario. And to ensure constraints a big spring constant is needed which results in stiff systems, which are difficult to integrate. These arguments also imply that the result of penalty methods may be visually believable but not physically exact.

4.5.2 Sequential Impulse-Based Methods

Sequential impulse-based methods use impulses to compute an instantaneous change in velocity. All constraints are treated one after the other.

Collisions without Friction

Baraff [Bara97b] shows how to deal with a single collision between two bodies without friction: For every colliding contact a linear impulse in direction of the contact normal $\vec{n}$ is computed:

$$\vec{J} = J\,\vec{n} \tag{4.40}$$

The impulse is then applied at the contact points. It will create a linear and angular impulse on the body and will change the velocities. Since $\vec{n}$ points from body *i* to body *j*, we will apply a negative impulse on body *i* and a positive impulse on body *j*:

$$\begin{aligned}
\vec{v}_{i+} &= \vec{v}_{i-} + \frac{-J\,\vec{n}}{m_i} \\
\vec{v}_{j+} &= \vec{v}_{j-} + \frac{J\,\vec{n}}{m_j} \\
\vec{\omega}_{i+} &= \vec{\omega}_{i-} + I_i^{-1}(\vec{r}_i \times (-J\,\vec{n})) \\
\vec{\omega}_{j+} &= \vec{\omega}_{j-} + I_j^{-1}(\vec{r}_j \times (J\,\vec{n}))
\end{aligned} \tag{4.41}$$

This impulse has to satisfy the collision constraint

$$\dot{C}_k = (\vec{v}_{j+} + \vec{\omega}_{j+} \times \vec{r}_j - \vec{v}_{i+} - \vec{\omega}_{i+} \times \vec{r}_i)\,\vec{n} + e\,\vec{v_{rel}}_{,n,-} = 0\,. \tag{4.42}$$

Here a local view is used that only involves the bodies *i* and *j*, hence the inequality is replaced by an equality. The post-collision velocities and the impulse magnitude *J* are the unknowns. Substituting equations (4.41) into (4.42) and rearranging yields Barraf's formula for the impulse magnitude *J*:

$$J = -\frac{\vec{v_{rel}}_{,n,-}\,(e+1)}{\frac{1}{m_i} + \frac{1}{m_j} + \vec{n} \cdot (I_i^{-1}(\vec{r}_i \times \vec{n})) \times \vec{r}_i + \vec{n} \cdot (I_j^{-1}(\vec{r}_j \times \vec{n})) \times \vec{r}_j} \tag{4.43}$$

After *J* was computed the velocities are changed to the post-collision velocities with equations (4.41).

Collisions with Friction

In the presence of friction forces the impulse need not be in the direction of $\vec{n}$. It can also have a tangential component that represents the friction impulse. The impulse formula (4.43) is only applicable for frictionless collisions. To compute friction we first derive another impulse formula (used in Mirtich [MiCa94]) that allows computing an impulse $\vec{J}$ for a desired relative velocity $\vec{v}_{rel,+}$ after the collision:

$$\vec{v}_{rel,+} = \vec{v}_{rel,-} + \boldsymbol{K}\,\vec{J} \quad \rightarrow \quad \vec{J} = \boldsymbol{K}^{-1}(\vec{v}_{rel,+} - \vec{v}_{rel,-}) \tag{4.44}$$

The matrix $\boldsymbol{K}$ can be derived like this:

$$\begin{aligned} v_{rel,+} - v_{rel,-} &= (\vec{v}_{j+} + \vec{\omega}_{j+} \times \vec{r}_j - \vec{v}_{i+} - \vec{\omega}_{i+} \times \vec{r}_i) - (\vec{v}_{j-} + \vec{\omega}_{j-} \times \vec{r}_j - \vec{v}_{i-} - \vec{\omega}_{i-} \times \vec{r}_i) \\ &= (\vec{v}_{j+} - \vec{v}_{j-} + (\vec{\omega}_{j+} - \vec{\omega}_{j-}) \times \vec{r}_j) - (\vec{v}_{i+} - \vec{v}_{i-} + (\vec{\omega}_{i+} - \vec{\omega}_{i-}) \times \vec{r}_i) \\ &= (\frac{\vec{J}}{m_j} + (\boldsymbol{I}_j^{-1}(\vec{r}_j \times \vec{J})) \times \vec{r}_j) - (\frac{-\vec{J}}{m_i} + (\boldsymbol{I}_i^{-1}(\vec{r}_i \times (-\vec{J}))) \times \vec{r}_i) \\ &= \frac{1}{m_j}\vec{J} - (skew(\vec{r}_j)\,\boldsymbol{I}_j^{-1}\,skew(\vec{r}_j))\vec{J} + \frac{1}{m_i}\vec{J} - (skew(\vec{r}_i)\,\boldsymbol{I}_i^{-1}\,skew(\vec{r}_i))\vec{J} \\ &= ((\frac{1}{m_j}\mathbf{1}_{3\times 3} - skew(\vec{r}_j)\,\boldsymbol{I}_j^{-1}\,skew(\vec{r}_j)) + (\frac{1}{m_i}\mathbf{1}_{3\times 3} - skew(\vec{r}_i)\,\boldsymbol{I}_i^{-1}\,skew(\vec{r}_i)))\vec{J} \\ &= (\boldsymbol{K}_j + \boldsymbol{K}_i)\vec{J} = \boldsymbol{K}\vec{J} \end{aligned} \tag{4.45}$$

So, matrix $\boldsymbol{K}$ can be computed in every time step with

$$\boldsymbol{K} = \boldsymbol{K}_j + \boldsymbol{K}_i = (\frac{1}{m_j}\mathbf{1}_{3\times 3} - skew(\vec{r}_j)\,\boldsymbol{I}_j^{-1}\,skew(\vec{r}_j)) + (\frac{1}{m_i}\mathbf{1}_{3\times 3} - skew(\vec{r}_i)\,\boldsymbol{I}_i^{-1}\,skew(\vec{r}_i)) \,. \tag{4.46}$$

Hahn [Hahn88] was the first to use impulses to compute collision response. Here is how he dealt with friction: He first assumes that friction in the collision is static, which means that no tangential movement occurs after the impact:

$$\vec{v}_{rel,+} = -e\, v_{rel,n,-}\, \vec{n}. \tag{4.47}$$

The static friction adds a tangential component to the impulse, so $\vec{J}$ is not in the direction of $\vec{n}$ any more. $\vec{J}$ can be computed with

$$\vec{J} = \boldsymbol{K}^{-1}(\vec{v}_{rel,+} - \vec{v}_{rel,-}) = \boldsymbol{K}^{-1}(-e\, v_{rel,n,-}\, \vec{n} - \vec{v}_{rel,-}) \tag{4.48}$$

This vector has a component in the direction of $\vec{n}$ and a component in a tangential direction to the surface:

$$\vec{J} = \vec{J}_n + \vec{J}_t \text{, with} \tag{4.49}$$

$$\vec{J}_n = (\vec{J}\,\vec{n})\,\vec{n} \text{ and} \tag{4.50}$$

$$\vec{J}_t = \vec{J} - \vec{J}_n \,. \tag{4.51}$$

The static friction assumption in (4.47) is only valid if

$$|\vec{J}_t| \le \mu |\vec{J}_n| \,. \tag{4.52}$$

Otherwise the impulse $\vec{J}$ does not lie in the friction cone and is not valid. In this case dynamic friction is used. Hahn simply clamps the tangential component to the maximal allowed friction impulse in dynamic friction and uses this impulse to compute the post-collision velocities:

$$\vec{J} = \vec{J}_n + \mu |\vec{J}_n| \frac{\vec{J}_t}{|\vec{J}_t|} \tag{4.53}$$

Guendelman et al. [GuBF03] use the same method as Hahn, but if the static friction assumption proves wrong, the collision impulse is recomputed. They assume that the bodies will start to slip in the direction

$$\vec{t} = \frac{\vec{v}_{rel,-} - v_{rel,n,-}\,\vec{n}}{|\vec{v}_{rel,-} - v_{rel,n,-}\,\vec{n}|} \,. \tag{4.54}$$

The impulse will be

$$\vec{J} = j\,\vec{n} - \mu\, j\,\vec{t} \tag{4.55}$$

and j (the magnitude of the normal component of $\vec{J}$) has to be found. Multiplying equation (4.44) by $\vec{n}^T$ leads to

$$v_{rel,n,+} = v_{rel,n,-} + \vec{n}^T \boldsymbol{K}\, \vec{J}\,. \tag{4.56}$$

Using the definition of the coefficient of restitution $v_{rel,n,+} = -e\, v_{rel,n,-}$, substituting $\vec{J}$ by equation (4.55), and rearranging yields

$$j = \frac{-(e+1)\, v_{rel,n,-}}{\vec{n}^T \boldsymbol{K}\,(\vec{n} - \vec{\mu}\,\vec{t})}\,. \tag{4.57}$$

If the static friction assumption was wrong, this impulse is used to compute the post-collision velocities.

Resting Contact

Hahn treats resting contacts in the same way as collisions: An object lying on the floor will be accelerated into the floor because of gravity. The object will gain velocity. An impulse with the procedure above is applied that reverses the velocity. In this method resting contact is a continuous series of collisions that is handled with a series of impulses. A body is not really "resting" on the floor, instead it is experiencing many tiny collisions.

The problem with this method is that resting contacts are treated as elastic collisions. This can lead to the phenomenon that resting bodies are vibrating, or bodies on an inclined plane are sliding down even if the static friction should stop the movement.

Mirtich and Canny [MiCa94], [MiCa95] built on Hahn's work. They use an impulse based formulation for simulation, but colliding contact and resting contact are treated differently. The direction of the relative tangent velocity in a contact is not assumed to be constant. So the reality is modeled more accurately than in the classical collision law. But the computation of the post-collision velocities involves solving an integral by numeric integration. Resting contacts are treated with so called *microcollisions*.

The method of Mirtich and Canny is known as *impulse-based simulation* and will not be explained in detail here since Guendelman et al. [GuBF03] have shown that colliding and resting contact can be treated uniformly. Uniform treatment is possible with the collision-contact-separated time step method discussed in Section 3.8.1, "Fixed Time Step Methods". In this time step method first all non-separating contacts are treated as collisions. After this step all colliding collisions should be resolved. Then the velocities are updated by numeric integration. External forces (like gravity) might cause velocities that violate the resting contact constraint. So after the velocity update all remaining non-separating contacts are treated as inelastic collisions (e = 0) to set the relative velocities to zero. Resolving an inelastic collision is the same as resolving a resting contact because the collision constraint equation equals the resting contact constraint equation if e = 0.

General Constraints

Research on sequential impulse-based methods has focused on handling collisions and resting contacts. The collision handling procedure can be generalized to deal with general constraints too.

For each constraint these steps are performed:

1. Compute the relative velocity $v_{rel,-}$ of the constraint anchor points.
2. Specify the desired relative velocity $v_{rel,+}$.
3. Compute a correction impulse:
$$\vec{J} = \boldsymbol{K}^{-1}(v_{rel,+} - v_{rel,-}) \tag{4.58}$$
4. Apply $-\vec{J}$ at body *i* and $\vec{J}$ at body *j* at the constraint anchor points.

The desired velocity in step 2 depends on the type of constraint. Constraints that only constrain the position and not the orientation can be realized easily with this procedure, for example: ball-and-socket joints and linear limits, where the desired velocity of step 2 is simply $v_{rel,+} = \vec{0}$. Several other constraints, like other joints, angular limits, linear and angular motors, can be implemented using sequential impulses and a bit of creativity. For example, a hinge joint can be created by connecting the bodies with two ball-and-socket joints along the hinge axis.

This method is simple and quick to implement, although it is not as efficient as the simultaneous methods described below.

Multiple Simultaneous Constraints

Multiple simultaneous constraints are resolved one after the other. But applying an impulse to two bodies causes new velocities, which might violate other constraints. Therefore sequential methods process the constraints over several iterations until either all constraints are satisfied or an iteration limit is reached.

Impulses can be applied directly, or an aggregated impulse for each body can be computed from the single constraint impulses.

It can take several iterations until all constraints are satisfied. Handling the constraints in the right order can accelerate the convergence of the process. For example, contacts with the highest penetration depth, lowest penetration depth, highest contact velocity, lowest contact velocity, etc. could be handled first, or another order could be found.

In Hahn [Hahn88] contacts are processed like this: A contact graph as described in Section 3.8.1, "Fixed Time Step Methods", is built. Then the graph is traversed in breadth first search order, starting at an immovable node (for example the ground floor). If there are no immovable nodes, the traversal can start at any node. All the contacts are processed in this order. All the impulses are summed up for a single body and then applied.

In Guendelman et al. [GuBF03] all contacts are processed in a loop. In each loop the non-separating contact with highest penetration depth is found and resolved. This is done until no colliding contact remains.

Stacks of rigid bodies may need a lot of iterations until a satisfying result is achieved because the impulses are propagated up and down during the iterations. Bodies in upper parts of a stack will keep pushing lower objects down into the immovable ground, which will cause interpenetrations. *Shock propagation* improves the convergence of sequential methods for stacks because the contacts are handled from the bottom to the top, and bodies in higher levels will not push bodies in lower levels down.

4.6 Simultaneous Methods

Simultaneous methods are more difficult to understand and implement than sequential methods, but very powerful. They allow finding the forces or impulses that guarantee constraints in one step for all simultaneous collisions, contacts, and general constraints. For equality constraints this involves solving a set of linear equations. With inequality constraints this implies solving an optimization problem. This optimization problem can be stated as a linear complementarity problem, which was discussed in Section 2.7, "The Linear Complementarity Problem".

4.6.1 Constraint Forces

An important term from classical mechanics is *constraint force*. All simulation methods that try to find constraint forces are called *analytical methods* or *constraint-based methods*.

The motion of a body is determined by the external forces (gravity, drag, springs, etc.) that act on it:

$$\boldsymbol{M}\,\dot{\vec{u}}(t) = \vec{f}_{ext}(t) \tag{4.59}$$

In the presence of constraints the equation of motion has to consider the constraint forces, which are an expression of the constraints and limit the motion:

$$\boldsymbol{M}\,\dot{\vec{u}}(t) = \vec{f}_{c}(t) + \vec{f}_{ext}(t) \tag{4.60}$$

$\vec{f}_{c}(t)$ is the *generalized constraint force vector* consisting of constraint force and constraint torques acting on the bodies at the centers of mass. If the constraint forces are applied in addition to the external forces, then the acceleration constraint equation is satisfied. In other words: The constraint forces are those forces that make $\ddot{\vec{C}} = \vec{0}$. If we assume that the position constraint equations and the velocity constraint equations are satisfied, the constraints will be satisfied as long as the right constraint forces are applied.

Next we have to think about the valid directions of the constraint forces. Each constraint needs one constraint force and determines its direction. The constraint force should be a passive, lossless force that stops "illegal" movement, but it should never add energy to the system nor should it remove energy from the system. In general this condition on the constraint force is known as the *principle of virtual work* (see Witkin et al. [WiGW90]). This principle implies that the constraint force is opposed to the direction in which the system is forbidden to move.

The constraint force will be applied at the constraint anchor points and result in a constraint torque and constraint force on the center of mass of the involved bodies. Since each constraint involves two bodies, each constraint force will be applied to two bodies (acting on one body positively and negatively on the other body).

From computational dynamics [Shab01] it is known that the valid constraint forces $\vec{f}_{c}$ are

$$\vec{f}_{c} = \boldsymbol{J}^{T}\vec{\lambda}\,, \tag{4.61}$$

where $\vec{\lambda}$ is an s-dimensional vector of scalar values. This scalar values determine the magnitudes of the constraint forces and are called *Lagrange multipliers*. The valid directions are determined by $\boldsymbol{J}^{T}$. Multiplying $\boldsymbol{J}^{T}$ by the Lagrange multipliers gives the correct forces and torques on the center of

mass of the bodies. The vector of scalars $\vec{\lambda}$ is the only unknown. – This sounds like black magic!

Let's investigate a few examples to illustrate this: In the linear limit constraint of Section 4.3.3, "Linear Limits", the constraint force on the center of mass may only act in the direction from one center of mass two the other, and it can not result in a torque. For the contact constraint the constraint force may only act in the direction of the normal vector, and it may only cause a torque that rotates the body around an axis normal to $\vec{r}$ and $\vec{n}$. In both cases this fact is correctly mirrored in the appropriate $\boldsymbol{J}$ matrices.

To further understand this fact, it helps to see what the Matrix $\boldsymbol{J}$ actually means. In Section 4.1.3, "Systems of Constraints", $\boldsymbol{J}$ was computed by differentiating $\vec{C}$ with respect to time:

$$\dot{\vec{C}} = \frac{\partial \vec{C}}{\partial \vec{s}} \dot{\vec{s}} \quad \text{and} \tag{4.62}$$

$$\frac{\partial \vec{C}}{\partial \vec{s}} = \begin{pmatrix} \frac{\partial C_1}{\partial s_1} & \frac{\partial C_1}{\partial s_2} & \cdots & \frac{\partial C_1}{\partial s_n} \\ \frac{\partial C_2}{\partial s_1} & \frac{\partial C_2}{\partial s_2} & \cdots & \frac{\partial C_2}{\partial s_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial C_n}{\partial s_1} & \frac{\partial C_n}{\partial s_2} & \cdots & \frac{\partial C_n}{\partial s_n} \end{pmatrix} \tag{4.63}$$

Each row vector of this matrix is the gradient of one of the scalar constraint functions C_k. These gradients are perpendicular to the allowed positions and show the direction in which the system is not permitted to move. Hence, the row-vectors determine the valid directions of the constraint forces. More on this thought is given in Witkin [Witk97b].

The equations of motion can now be written as

$$\boldsymbol{M}\,\dot{\vec{u}} = \boldsymbol{J}^T \vec{\lambda} + \vec{f}_{ext}\,. \tag{4.64}$$

The Lagrange multipliers $\vec{\lambda}$ are unknown and have to be found. Then the resulting acceleration $\dot{\vec{u}}$ can be computed and this acceleration will obey the constraints. Using this acceleration in the velocity and position update leads to a correct motion that satisfies the constraints.

4.6.2 Simultaneous Force-based Methods

This section will show how constraints formulated as acceleration constraints in the form

$$\ddot{\vec{C}} = \boldsymbol{J}\,\dot{\vec{u}} + \vec{c} = \vec{0} \quad \text{or} \quad \ddot{\vec{C}} = \boldsymbol{J}\,\dot{\vec{u}} + \vec{c} \geq \vec{0} \tag{4.65}$$

can be solved simultaneously.

Equality constraints

Finding the constraint forces by computing $\vec{\lambda}$ for equality constraints is known as the *Lagrange multiplier method*. It is described in Witkin et al. [Witk97b], [WiGW90], and Baraff [Bara96].

The Lagrange multipliers and the accelerations can be found with the equation of motion and the acceleration constraint equations:

$$\boldsymbol{M}\,\dot{\vec{u}} = \boldsymbol{J}^T\vec{\lambda} + \vec{f}_{ext} \tag{4.66}$$

$$\ddot{\vec{C}} = \boldsymbol{J}\,\dot{\vec{u}} + \vec{c} = \vec{0} \tag{4.67}$$

These two equations are often combined:

$$\begin{pmatrix} \boldsymbol{M} & -\boldsymbol{J}^T \\ \boldsymbol{J} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \dot{\vec{u}} \\ \vec{\lambda} \end{pmatrix} = \begin{pmatrix} \vec{f}_{ext} \\ -\vec{c} \end{pmatrix} \tag{4.68}$$

This representation is common in robotics. By solving for $\vec{\lambda}$ first, a smaller system of equations is obtained. Substituting $\dot{\vec{u}}$ of equation (4.66) in equation (4.67) gives

$$\dot{\vec{u}} = \boldsymbol{M}^{-1}\boldsymbol{J}^T\lambda + \boldsymbol{M}^{-1}\vec{f}_{ext} \quad \rightarrow \tag{4.69}$$

$$\ddot{\vec{C}} = \boldsymbol{J}\,(\boldsymbol{M}^{-1}\boldsymbol{J}^T\lambda + \boldsymbol{M}^{-1}\vec{f}_{ext}) + \vec{c} = \vec{0} \tag{4.70}$$

Rearranging leads to

$$\boldsymbol{J}\,\boldsymbol{M}^{-1}\boldsymbol{J}^T\lambda + \boldsymbol{J}\,\boldsymbol{M}^{-1}\vec{f}_{ext} + \vec{c} = \vec{0} \tag{4.71}$$

With $\boldsymbol{A} = \boldsymbol{J}\,\boldsymbol{M}^{-1}\boldsymbol{J}^T$ and $\vec{b} = \boldsymbol{J}\,M^{-1}\vec{f}_{ext} + \vec{c}$ this simplifies to

$$\boldsymbol{A}\vec{\lambda} + \vec{b} = \vec{0} \tag{4.72}$$

Equation (4.68) or equation (4.72) can be solved to find the unknown magnitudes of the constraint forces. The involved matrices, the mass matrix $\boldsymbol{M}$ and the Jacobian $\boldsymbol{J}$, are sparse. This should be considered when deciding how to solve this equations. Witkin [Witk97b] uses a conjugate gradient method (see Press et al. [PTVF03]). Baraff [Bara96] shows how this equation can be solved even more efficiently.

Resting Contact without Friction

In the last section only equality constraints were handled. The resting contact constraint is an inequality constraint:

$$C_k = (\vec{x}_{Pj} - \vec{x}_{Pi})\,\vec{n} \geq 0 \tag{4.73}$$

If only inequality constraints should be handled, the system

$$\ddot{\vec{C}} = \boldsymbol{A}\vec{\lambda} + \vec{b} \geq \vec{0} \tag{4.74}$$

has to be solved instead of (4.72). This problem can be formulated as a LCP. Löstedt [Löts82] was the first who used a LCP to solve this kind of problems, and LCPs were made popular by the works of Baraff [Bara89], [Bara90], [Bara91], [Bara93], [Bara94], [Bara95], [Bara97b]. The conditions that lead to the LCP formulation are:

- The relative contact acceleration in the direction of the normal vector is given by $\ddot{\vec{C}} = \boldsymbol{A}\vec{\lambda} + \vec{b}$.
- The bodies are not allowed to accelerate into each other, therefore $\ddot{C}_k \geq 0$.

- The constraint forces act to push the bodies apart, but they are not allowed to "glue" the bodies together: $\lambda_k \geq 0$.
- The constraint force is only acting if the bodies are not separating. If the bodies are separating (which means that the constraint is satisfied), the constraint force has to disappear: $\ddot{C}_k \lambda_k = 0$. This is the complementarity condition.

This leads to a standard LCP that allows solving for $\vec{\lambda}$:

$$\ddot{\vec{C}} = \boldsymbol{A}\vec{\lambda} + \vec{b}\,,\ \ddot{C}_k \geq 0\,,\ \lambda_k \geq 0\,, \text{ and } \ddot{C}_k \lambda_k = 0\,. \tag{4.75}$$

This LCP formulation is known as the *acceleration-based formulation*. Baraff [Bara97b] shows in detail how $\ddot{C}_k$ can be computed, which is too lengthy to repeat here.

Combining Equality and Inequality Constraints

A MLCP can treat equality constraints and inequality constraints together. This is done by combining the rows of equality constraints and inequality constraints in one Jabocian matrix $\boldsymbol{J}$ and setting appropriate bounds lo_k and hi_k. Each equality constraint equation contributes one row in $\boldsymbol{J}$ and the bounds are set to $lo_k = -\infty$ and $hi_k = \infty$. Each inequality constraint contributes one row and the bounds are $lo_k = 0$ and $hi_k = \infty$. This way, only one MLCP has to be solved:

$$\begin{pmatrix} \vec{0} \\ \ddot{\vec{C}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{M} & -\boldsymbol{J}^T \\ \boldsymbol{J} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \dot{\vec{u}} \\ \vec{\lambda} \end{pmatrix} + \begin{pmatrix} -\vec{f}_{ext} \\ \vec{c} \end{pmatrix}, \tag{4.76}$$

$$\begin{pmatrix} \vec{0} \\ \ddot{\vec{C}} \end{pmatrix} \geq \vec{0}\,,\ \begin{pmatrix} -\vec{\infty} \\ \vec{lo} \end{pmatrix} \leq \begin{pmatrix} \dot{\vec{u}} \\ \vec{\lambda} \end{pmatrix} \leq \begin{pmatrix} \vec{\infty} \\ \vec{hi} \end{pmatrix}, \text{ and } \begin{pmatrix} \vec{0} \\ \ddot{\vec{C}} \end{pmatrix}^T \begin{pmatrix} \dot{\vec{u}} \\ \vec{\lambda} \end{pmatrix} = \vec{0} \tag{4.77}$$

The conditions (4.77) can be simplified since the upper parts of the vectors contain no restrictions:

$$\ddot{\vec{C}} \geq \vec{0}\,,\ \vec{lo} \leq \vec{\lambda} \leq \vec{hi}\,, \text{ and } \ddot{\vec{C}}^T \vec{\lambda} = \vec{0} \tag{4.78}$$

We will encounter a MLCP in a similar form several times throughout this thesis. Such a MLCP can always be reduced to the following MLCP by substituting $\dot{\vec{u}}$ and solving for $\vec{\lambda}$ first:

$$\ddot{\vec{C}} = \boldsymbol{A}\vec{\lambda} + \vec{b}\,,\ \ddot{\vec{C}} \geq \vec{0}\,,\ \vec{lo} \leq \vec{\lambda} \leq \vec{hi}\,, \text{ and } \ddot{\vec{C}}^T \vec{\lambda} = \vec{0}\,. \tag{4.79}$$

Resting Contact with Friction

The acceleration-based LCP without friction always has a solution, whereas in the presence of friction the LCP might have no feasible solution – even if there is only one contact involved. This means that there might be situations in which there is no valid set of contact forces – no contact forces can stop the bodies from penetrating. Some interpenetrations can only be avoided with the use of impulses [Bara91]. Baraff [Bara94] describes Dantzig's algorithm for solving LCPs and how it can be modified and used to find the static and dynamic friction forces and the impulses to solve inconsistent configurations. Baraff [Bara91] states that in practice he has not encountered such an inconsistent configuration in simulations with $\mu < 1$. The absence of a valid solution indicates that Coloumb's model of friction is not sufficient to describe all situations in the real world.

4.6.3 Simultaneous Impulse-based Methods

Simultaneous force-based methods are normally not used to handle collisions, since collisions are better treated with impulses. Resting contacts and general constraints can also be solved using constraint impulses instead of constraint forces.

Simultaneous impulse-based methods seem to have several advantages to force-based methods: They use only impulses and avoid a mix of constraint forces and impulses. Anitescu and Potra [AnPo97] showed that a velocity-based LCP formulation always has a solution in contrast to the acceleration-based formulation.

Inequality and Equality Constraints

Stewart and Trinkle [StTr96] showed that the acceleration-based formulation with forces can be turned into a formulation that uses impulses. Their original formulation was position-based (it used the position constraint equation). Anitescu and Potra [AnPo97] showed that using a velocity-based formulation yields a LCP that will always have a solution for any number of contacts, even if friction is present.

A constraint force that is acting over a time step results in an impulse since $\vec{J} = \vec{F}\,\Delta t$. For this method the accelerations are approximated by an explicit Euler step:

$$\dot{\vec{u}} \approx \frac{\vec{u}_+ - \vec{u}_-}{\Delta t}. \tag{4.80}$$

Subscript '+' denotes the velocity at the end of the time step, subscript '–' denotes the velocity at the beginning of the time step. Inserting this into the equation of motion leads to

$$\boldsymbol{M}\,(\vec{u}_+ - \vec{u}_-) = \Delta t\,\boldsymbol{J}^T \vec{\lambda}_f + \Delta t\,\vec{f}_{ext}. \tag{4.81}$$

$\vec{\lambda}_f$ are the unknown constraint force magnitudes. Here we directly search for the impulses, so our new unknowns $\vec{\lambda}$ are the constraint impulse magnitudes:

$$\vec{\lambda} = \Delta t\,\vec{\lambda}_f \tag{4.82}$$

This leads to

$$\boldsymbol{M}\,\vec{u}_+ - \boldsymbol{J}^T \vec{\lambda} = \boldsymbol{M}\,\vec{u}_- + \Delta t\,\vec{f}_{ext} \tag{4.83}$$

In combination with the velocity constraints this produces a new MLCP:

$$\begin{pmatrix} \vec{0} \\ \dot{\vec{C}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{M} & -\boldsymbol{J}^T \\ \boldsymbol{J} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \vec{u}_+ \\ \vec{\lambda} \end{pmatrix} + \begin{pmatrix} -\boldsymbol{M}\,\vec{u}_- - \Delta t\,\vec{f}_{ext} \\ \vec{0} \end{pmatrix}, \tag{4.84}$$

$$\dot{\vec{C}} \geq \vec{0}\,,\ \vec{lo} \leq \vec{\lambda} \leq \vec{hi}\,, \text{ and } \dot{\vec{C}}^T \vec{\lambda} = \vec{0} \tag{4.85}$$

This MLCP allows finding the constraint impulses $\boldsymbol{J}^T \vec{\lambda}$ and the new velocities $\vec{u}_+$ at the end of the time step. The method is also called a *time-stepping method* because explicit Euler integration of velocities is part of this approach.

Explicit Euler integration in equation (4.80) is not suitable for stiff systems. Using implicit Euler integration leads to stable results. To apply implicit Euler integration $\boldsymbol{M}(t)$ and $\vec{f}_{ext}(t)$ in equation

(4.81) have to be substituted by $\boldsymbol{M}(t+\Delta t)$[1] and $\vec{f}_{ext}(t+\Delta t)$. Since these values are unknown, they have to be approximated as shown in Anitescu and Potra [AnPo02].

Collisions

Baraff [Bara89] and Eberly [Eber04] show that the collision impulses for multiple simultaneous collisions can be found by using the collision constraint equation

$$\dot{C}_k = v_{rel,n,+} + e\, v_{rel,n,-} \geq 0 \tag{4.86}$$

to formulate a LCP. Giang et al. [GiBO03] show in detail how the LCP can be computed. Kawachi et al. [KaSK97] describe how friction can be calculated in this LCP.

These collision constraints can be integrated easily in MLCP (4.84-4.85) by allowing velocity constraints functions of the form $\dot{\vec{C}} = \boldsymbol{J}\,\vec{u} + \vec{d}$:

$$\begin{pmatrix} \vec{0} \\ \dot{\vec{C}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{M} & -\boldsymbol{J}^T \\ \boldsymbol{J} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \vec{u}_+ \\ \vec{\lambda} \end{pmatrix} + \begin{pmatrix} -\boldsymbol{M}\,\vec{u}_- - \Delta t\, \vec{f}_{ext} \\ \vec{d} \end{pmatrix}, \tag{4.87}$$

$$\dot{\vec{C}} \geq \vec{0}\,,\ \vec{lo} \leq \vec{\lambda} \leq \vec{hi} \text{ and } \dot{\vec{C}}^T \vec{\lambda} = \vec{0}\,. \tag{4.88}$$

The elements d_k of $\vec{d}$ are zero for normal constraints and set to $e\, v_{rel,n,-}$ for collision constraints. Nonzero d_k values have the effect that velocities are enforced, which can also be used to model motors.

Friction

Yet friction is not handled in the MLCP. The friction in the tangential plane is given by

$$f_x^2 + f_y^2 \leq \mu^2 f_n^2\,. \tag{4.89}$$

This condition describes the friction cone, but it is not linear and thus cannot be used in a MLCP. Therefore the friction cone of each contact k can be approximated by a 4-sided pyramid as it is suggested in Trinkle et al. [TPSL97], see Figure 4.4. For the two tangential directions given by $\vec{t}_x$ and $\vec{t}_y$ a constraint is created that forbids movement in this direction:

$$\begin{aligned} \dot{C}_{k,t_x} &= (\vec{v}_j + \vec{\omega}_j \times \vec{r}_j - \vec{v}_i - \vec{\omega}_i \times \vec{r}_i)\,\vec{t}_x = \\ &= \begin{pmatrix} -\vec{t}_x^{\,T} & -(\vec{r}_i \times \vec{t}_x)^T \end{pmatrix} \begin{pmatrix} \vec{v}_i \\ \vec{\omega}_i \end{pmatrix} + \begin{pmatrix} \vec{t}_x^{\,T} & (\vec{r}_j \times \vec{t}_x)^T \end{pmatrix} \begin{pmatrix} \vec{v}_j \\ \vec{\omega}_j \end{pmatrix} = 0 \end{aligned} \tag{4.90}$$

$$\begin{aligned} \dot{C}_{k,t_y} &= (\vec{v}_j + \vec{\omega}_j \times \vec{r}_j - \vec{v}_i - \vec{\omega}_i \times \vec{r}_i)\,\vec{t}_y = \\ &= \begin{pmatrix} -\vec{t}_y^{\,T} & -(\vec{r}_i \times \vec{t}_y)^T \end{pmatrix} \begin{pmatrix} \vec{v}_i \\ \vec{\omega}_i \end{pmatrix} + \begin{pmatrix} \vec{t}_y^{\,T} & (\vec{r}_j \times \vec{t}_y)^T \end{pmatrix} \begin{pmatrix} \vec{v}_j \\ \vec{\omega}_j \end{pmatrix} = 0 \end{aligned} \tag{4.91}$$

Coloumb's law claims that the friction impulse has to be limited by the normal impulse in this contact k. The normal impulse is given by λ_k. Let's call the Lagrange multipliers for the two friction constraint rows for contact k λ_{k,t_x} and λ_{k,t_y}. The bounds for these unknowns are set to

1 $\boldsymbol{M}$ depends on time because it contains the time-dependent inertia tensors in world coordinates.

$$\begin{aligned} -\mu\lambda_k \le \lambda_{k,t_x} \le \mu\lambda_k \quad &\rightarrow \quad lo_{k,t_x} = -\mu\lambda_k, hi_{k,t_x} = \mu\lambda_k \\ -\mu\lambda_k \le \lambda_{k,t_y} \le \mu\lambda_k \quad &\rightarrow \quad lo_{k,t_y} = -\mu\lambda_k, hi_{k,t_y} = \mu\lambda_k \end{aligned} . \tag{4.92}$$

This way, for each contact two additional friction constraints are created and appended to the other constraints. This creates dependent limits as discussed in Section 2.7.3, "Dependent Limits".

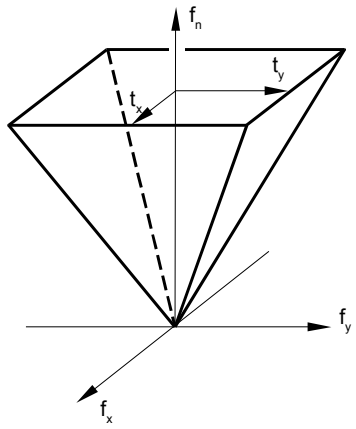

Figure 4.4: 4-sided friction pyramid.

In practice it is often sufficient to use only one friction constraint per contact in direction $\vec{t}_x$ instead of one constraint for $\vec{t}_x$ and one constraint for $\vec{t}_y$. Therefore the direction of $\vec{t}_x$ has to be chosen carefully. In the dynamic friction case the friction impulse opposes the relative tangent velocity, so $\vec{t}_x$ is chosen to point in the direction of the relative tangent velocity. In the static friction case the relative tangent velocity is zero, and $\vec{t}_x$ is set to the last valid tangent direction or to an arbitrary tangential direction. For new contacts an arbitrary direction for $\vec{t}_x$ is used as well. This has led to satisfying results in the tested scenarios.

Some solvers cannot deal with dependent limits. Dependent limits can be avoided by using constant bounds instead of bounds that depend on the normal impulse, see Karma [Karm02] or Catto [Catt05]:

$$-f_{max} \le \lambda_{k,t_x} \le f_{max} \tag{4.93}$$

f_{max} should be set to a value proportional to the coefficient of friction, the involved masses, and the gravity acceleration. This is called *box friction* since the friction pyramid is substituted by a box. This simplification is sufficient for many scenarios. The drawback of this method is especially visible in stacks of boxes: With box friction the same friction bound is applied to all boxes. But Coloumb's law states that the highest box slides more easily than the lowest box since the normal force on the lowest box is bigger.

Stewart and Trinkle [StTr96] and Anitescu and Potra [AnPo97] show how the friction cone can be approximated by an n-sided pyramid with more than 4 sides. Additionally dependent limits can be avoided by adding auxiliary variables and additional constraints to the LCP formulation.

Over-constrained Systems

The matrix $\boldsymbol{A}$ of the LCPs is not guaranteed to be non-singular. Redundant constraints will make matrix $\boldsymbol{A}$ singular. For example, a box on the floor with more than three contact points is overdetermined and will lead to redundant constraints, as discussed in Mirtich [Mirt98a]. In this

case the solution is not unique. Several solutions will lead to the correct physical behavior, see Figure 4.5. General constraints can be contradicting too, for example if one body is attached to two distant points in space. These cases should be handled robustly in the simulation.

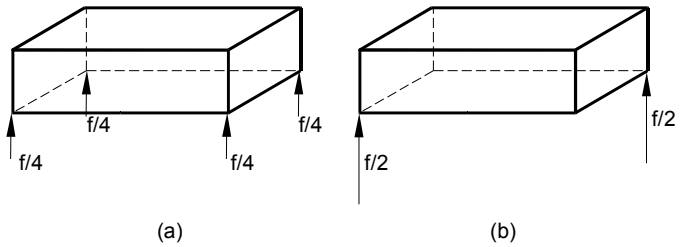

Figure 4.5: Two sets of forces on a box. The same total force is applied in both cases.

4.7 Conclusion

Which of the described methods is the best? No method is truly superior. Penalty methods are easy to understand and easy to implement at first, but many authors claim that penalty methods lead to stiff equations and thus unstable systems. Nevertheless, Kačić-Alesić et al. [KaNB03] show that these methods can be used successfully in practical simulation systems.

Sequential impulse-based methods are also easy to understand and easier to implement than LCP-based methods. Lacoursière [Laco03] claims that all sequential pairwise methods resemble Gauss-Seidel processes and thus are similar to solving an equivalent LCP with an iterative Gauss-Seidel solver.

Velocity-based LCP formulations seem to replace force-based LCP formulations as they do not suffer from non-existence of solutions and can easily combine collisions and resting contacts. Many available dynamics frameworks use velocity-based LCP methods, for example the Open Dynamics Engine [Smit05]. This LCP method has many advantages, but it is more difficult to understand and it needs a robust and fast LCP solver.

5 Error Correction

"In a few minutes a computer can make a mistake so great that it would have taken many men many months to equal it."

- Author unknown

During the simulation errors will occur. Here, errors are violations of constraints. Normally constraints are defined on positions and orientations of bodies, so an error is a wrong position or orientation. This error is given by C_k in equality constraints. The discussed constraint solvers (except the penalty method) try to prevent errors from growing – but they don't remove existing errors. A robust simulation has to be able to correct errors. This could be done by setting a correction force, impulse, or position-change which will move the bodies to the right position.

Error correction can even be used for other effects. For example: The rigid body parts of a human body are unconnected and lie on the floor. Then the joint constraints of the human are activated, and the body parts start to move, so that the body will slowly "assemble". Or: A box shall be moved to another place. A joint can be setup that connects the box and the distant location. Error correction moves the box until the joint constraint is satisfied. See Section 8.4, "Example: Error Correction".

Constraint stabilization is another name for error correction. Usually it is integrated in the methods discussed in the last chapter. This is typically done with *Baumgarte stabilization*. An exception is the penalty method that automatically corrects errors because it computes a force that tries to keep $C_k=0$. One could claim that what penalty methods do is pure error correction.

This chapter introduces position-based methods. A position-based constraint solver can be called in an extra step to separate error correction from the normal constraint solving.

5.1 Error Reduction Parameter

It is not always desirable to reduce the whole error in one time step. Therefore in ODE [Smit05] a factor *ERP* controls how much of the error should be corrected per time step. *ERP* is a value between 0 and 1. $ERP=0$ means no error correction, $ERP=1$ means the entire error is fixed in

each time step. Setting *ERP* to 1 is often not recommended. This could lead to undesirable effects because of internal approximations and inaccuracies in the error correction procedure.

5.2 Baumgarte's Constraint Stabilization Method

Simultaneous force-based methods assume that the position constraint equation and the velocity constraint equation are satisfied. In terms of equality constrains: $\vec{C}=\vec{0}$ and $\dot{\vec{C}}=\vec{0}$. A constraint force is computed that keeps $\ddot{\vec{C}}=\vec{0}$. If due to errors $\vec{C}$ or $\dot{\vec{C}}$ deviate from zero, nothing is done to correct this. Using Baumgarte stabilization [Shab01] the constraint forces are calculated in order to keep

$$\ddot{C}_k+2\alpha\dot{C}_k+\beta^2 C_k=0\,, \tag{5.1}$$

where $\alpha>0$ and $\beta\neq 0$.

The force that is calculated in this way is a combination of a constraint force and an error correction force. This is used for example in Barzel and Barr [BaBa88]. The disadvantage of this method is that there is no known reliable method to select the coefficients α and β. Incorrect coefficients can produce visually incorrect behavior. For example, penetrating bodies gain a lot of speed and shoot out of each other. Karasawa and Sogabe [KaSo04] state that the constants α and β are usually taken as equal values between 1 and 20.

Baumgarte stabilization can be used to add error correction to the sequential impulse-based method, the simultaneous force-based method and the simultaneous impulse-based method.

5.2.1 Sequential Impulse-based Methods

Impulse-based methods use velocity constraints. Baumgarte stabilization is therefore simplified to

$$\dot{C}_k+\beta^2 C_k=0\,. \tag{5.2}$$

The coefficient can be chosen to be, for example, $\beta^2=\dfrac{ERP}{\Delta t}$. Then equation (5.2) is easier to understand:

$$\dot{C}_k=-ERP\,\frac{C_k}{\Delta t} \tag{5.3}$$

This equation states that the constraint velocity should be set to a velocity that removes the position deviation. And the amount of error reduction is controlled by the factor *ERP*.

The steps for this method are:

1. Compute the relative velocity $v_{rel,-}$ of the constraint anchor points.
2. Specify the desired relative correction velocity $v_{rel,+}$ that will remove the position error.
3. Compute a correction impulse:

$$\vec{J}=\boldsymbol{K}^{-1}(v_{rel,+}-v_{rel,-}) \tag{5.4}$$

4. Apply $-\vec{j}$ at body *i* and $\vec{j}$ at body *j* at the constraint anchor points.

This method can be used as separate step after the constraint handling, or it can be combined with the sequential impulse-based constraint handling to find a combined constraint and error correction impulse.

5.2.2 Simultaneous Force-based Methods

Adding error correction to simultaneous force-based methods can be done by adding Baumgarte stabilization to the MCLP (4.76-4.77). This gives a new MLCP that computes a combined constraint and error correction force:

$$\begin{pmatrix} \vec{0} \\ \ddot{\vec{C}} + 2\alpha\dot{\vec{C}} + \beta^2\vec{C} \end{pmatrix} = \begin{pmatrix} \boldsymbol{M} & -\boldsymbol{J}^T \\ \boldsymbol{J} & 0 \end{pmatrix} \begin{pmatrix} \dot{\vec{u}} \\ \vec{\lambda} \end{pmatrix} + \begin{pmatrix} -\vec{f}_{ext} \\ \vec{\tau} + 2\alpha\dot{\vec{C}} + \beta^2\vec{C} \end{pmatrix}, \tag{5.5}$$

$$\ddot{\vec{C}} + 2\alpha\dot{\vec{C}} + \beta^2\vec{C} \geq \vec{0}\ ,\ \vec{lo} \leq \vec{\lambda} \leq \vec{hi} \text{ and } (\ddot{\vec{C}} + 2\alpha\dot{\vec{C}} + \beta^2\vec{C})^T\vec{\lambda} = \vec{0}\ . \tag{5.6}$$

5.2.3 Simultaneous Impulse-based Methods

The simplified Baumgarte stabilization given in equation (5.3) can be added to the MLCP (4.84-4.85) like this

$$\begin{pmatrix} \vec{0} \\ \dot{\vec{C}} + \frac{ERP}{\Delta t}\vec{C} \end{pmatrix} = \begin{pmatrix} \boldsymbol{M} & -\boldsymbol{J}^T \\ \boldsymbol{J} & 0 \end{pmatrix} \begin{pmatrix} \vec{u}_+ \\ \vec{\lambda} \end{pmatrix} + \begin{pmatrix} -\boldsymbol{M}\,\vec{u}_- - \Delta t\,\vec{f}_{ext} \\ \frac{ERP}{\Delta t}\vec{C} \end{pmatrix} \tag{5.7}$$

$$\dot{\vec{C}} + \frac{ERP}{\Delta t}\vec{C} \geq \vec{0}\ ,\ \vec{lo} \leq \vec{\lambda} \leq \vec{hi} \text{ and } (\dot{\vec{C}} + \frac{ERP}{\Delta t}\vec{C})^T\vec{\lambda} = \vec{0}\ . \tag{5.8}$$

5.3 Position-based Methods

Position-based methods are also called *projection-based methods* [Jako01] or *separation*. The idea is very intuitive: If a position constraint is violated, then we change the positions of the participating bodies until the constraint is satisfied again. This method manipulates the position and orientation of bodies directly and leaves velocities untouched. Thus, it is excellent for correcting errors of position constraints. But it is not sufficient to handle collisions because collisions require velocity changes. This method is also not adequate to handle resting contacts and general constraints: Imagine a sphere sitting on a plane. Position correction will manage to avoid penetrations, but due to the gravity force the velocity will grow to infinity and cause numerical problems.

This method for error correction is usually done as *post-stabilization*, which means, it is called after the position update to remove any errors in the new positions.

In simple implementations of this method only the position of the center of mass is changed. This is used in several works, for example: Egan [Egan03] and Kavan [Kava03]. If a constraint is only

defined on the center of mass or only on the orientation, then simply moving the center of mass or rotating the body can bring the body into a valid state. But if the constraint anchor point is different from the center of mass, computing the position and orientation change is not trivial. A new corrected position can result from several different combinations of rotations around the center of mass and translations of the center of mass. The right combination of rotation and translation has to be found, and the mass and inertia matrix should be considered.

Baltman and Radeztsky [BaRa04] use the collision-impulse equation (4.43) to compute the position and orientation change, but they do not explain why this works. One way to explain this is: The relative velocity of the constraint anchor points is temporarily set to a velocity that would move the anchor points to the desired relative places within one time step Δt. This velocity can be set by applying an impulse at the anchor point. After applying the impulse, the body's position and orientation is integrated with an explicit Euler step using the new temporary velocities. This moves the bodies to the desired places[1]. The temporary velocities are only used for this position change; the original velocities of the bodies stay untouched.

5.3.1 Sequential Position-based Methods

In sequential position-based method, all constraint errors are corrected sequentially [Jako01], [BaRa04]. Correcting one error could violate other constraints. Therefore, this method has to iterate over all constraint errors several times – like the sequential impulse-based methods.

First, the bodies are assumed to be motionless since only the spatial variables are considered at the moment. For each constraint an impulse is computed with $\vec{J} = \boldsymbol{K}^{-1}(v_{\vec{rel},+} - v_{\vec{rel},-})$, where $v_{\vec{rel},+}$ is the desired temporary correction velocity and $v_{\vec{rel},-} = \vec{0}$. This impulse is applied to set a temporary velocity, which is used in an explicit Euler step to integrate positions.

A difficulty in sequentially handling position corrections this way is that after each update the collision detection has to be called for the updated bodies. Due do the position change new errors could have occurred or other errors could have been resolved.

5.3.2 Simultaneous Position-based Methods

The same principle can be applied in a simultaneous method [Faur97], [Clin02], [ClPa03], [Erle04]. The MLCP (4.84-4.85) can be used to compute the necessary impulses and temporary velocities. In the correction step the current velocities and external forces are ignored ($\vec{u}_{-} = \vec{0}$ and $\vec{f}_{ext} = \vec{0}$), so we obtain a much simpler MLCP[2]:

$$\begin{pmatrix} \vec{0} \\ \dot{\vec{C}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{M} & -\boldsymbol{J}^T \\ \boldsymbol{J} & 0 \end{pmatrix} \begin{pmatrix} \vec{u}_{temp} \\ \vec{\lambda} \end{pmatrix} + \begin{pmatrix} \vec{0} \\ \dfrac{ERP}{\Delta t} \vec{C} \end{pmatrix}, \tag{5.9}$$

1 Interestingly, this procedure can be interpreted as applying a *first order world force* over the time Δt. The real world is a *second order world*: a force applied over time changes the velocity of a particle. A *first order world* is a world in which $F = mv$ instead of $F = ma$. In a first order world, a force applied over time directly changes the position of a particle. See Erleben [Erle04] for a detailed discussion.

2 In Cline and Pai [Clin02], [ClPa03] this LCP is multiplied by Δt. This way, the MLCP is formulated on the position level instead of the velocity level.

$$\dot{\vec{C}} \geq \vec{0}\,,\ \vec{lo} \leq \vec{\lambda} \leq \vec{hi} \text{ and } \dot{\vec{C}}^T \vec{\lambda} = \vec{0}\,. \tag{5.10}$$

An explicit Euler integration step with $\vec{u}_{temp}$ moves bodies to the corrected positions.

5.4 Conclusion

Error correction removes penetrations and other constraint violations. In addition it can be used for other effects like assembling articulated bodies. It can be added to the constraint solvers with Baumgarte stabilization.

Baumgarte stabilization has the drawback that a velocity is set to correct the error. This velocity remains even after the error has disappeared.

Position-based methods are a new kind of constraint solvers, which don't suffer from this problem. A disadvantage of these methods is that they usually require an extra step.

6 Other Simulation Methods

"Advice is what we ask for when we already know the answer but wish we didn't."

- Erica Jong

So far, the most prominent simulation methods have been discussed. This chapter shows several other ideas, which cannot be discussed in this thesis in detail.

6.1 Velocity-less Formulations

Usually, a rigid body's state is described by its spatial coordinates and velocities. In terms of linear motion: $\vec{x}(t)$ and $\vec{v}(t)$. This is not the only choice. Another idea is to store the current position $\vec{x}(t)$ and the last position change $\Delta\vec{x}(t)$ [Faur97], or the current position $\vec{x}(t)$ and the last position $\vec{x}(t-\Delta t)$ [Jako01]. In this state representation the velocity is described implicitly. Jakobsen [Jako01] uses a Verlet integration scheme to update the rigid body state:

$$\begin{aligned}\vec{x}(t+\Delta t) &= 2\vec{x}(t)-\vec{x}(t-\Delta t)+\vec{a}(t)\Delta t^2,\\ \vec{x}(t-\Delta t) &= \vec{x}(t)\end{aligned}, \tag{6.1}$$

This state representation is suitable for certain applications, for example, when tracking data is used, where the sensors only provide position data and no velocity data [Faur97]. Another advantage is that position correction of $\vec{x}(t)$ implicitly changes and corrects the velocities. Using this scheme the position-based method alone can be used to handle general constraints and resting contacts.

A drawback is that whenever a velocity value is needed, it has to be approximated by the position change and the time step Δt.

Jakobsen adds another idea and uses only particles in his simulation. Rigid bodies, like boxes, are described by several particles (one particle for each corner) that are connected by a fixed-distance constraint. This reduces rigid body simulation to the simpler problem of simulating particles with constraints, but even for only one single body several particles and constraints are necessary.

These disadvantages make this concept less appropriate for a general-purpose rigid body simulator.

6.2 Generalized Coordinates and Lagrangian Dynamics

Holonomic constraints reduce the degrees of freedom of a system and create dependencies between the coordinates of the particles and bodies. A system of n bodies has $6n$ independent coordinates or *degrees of freedom*. k independent holonomic constraint equations in the form of equation (4.2) reduce the degrees of freedom by k. Only $6n-k$ independent coordinates are left.

Using positions and orientations to describe rigid bodies is intuitive. These are called *maximal coordinates*. But sometimes other coordinates are more useful to solve specific problems. For example, the motion of planets is known to be easier to solve in *polar coordinates* than in *Cartesian coordinates*. Another example: A rigid body is on one end connected to a fixed point in the world with a hinge joint. The other end is connected to another body with a second hinge joint. Both bodies together form a *double pendulum*. Each hinge joint consists of five holonomic constraints (five rows in the constraint Jacobian). The only two variables, needed to describe the system, are the two independent hinge angles.

In general, if we have a system with $6n$ maximal coordinates and k independent holonomic constraints, then we can introduce $6n-k$ new independent variables to describe the system. These new variables are called *generalized coordinates* or *reduced coordinates*. Candidates for generalized coordinates are not only orthogonal position coordinates or angles. Also other quantities, such as energy or angular momentum, can be used.

Solving for constraints in the previous chapter meant that the constraint forces had to be found, which were additional unknowns in the equations of motion. This problem can be solved by another form of equation of motion: *Lagrange's equation of motion*. This formulation is based on generalized coordinate and does not contain constraint forces. Lagrange's equation of motion leads to the extensive field of *Lagrangian dynamics*[1] (in contrast to *Newtonian dynamics* which we have discussed before), which is thoroughly discussed in Goldstein et al. [GoPS02] and Kuypers [Kuyp03].

Methods that use generalized coordinates are not discussed here for a number of reasons. To use them, a parameterization of the system in generalized coordinates has to be found a priori. This depends heavily on the current problem situation. There are no rules how to find the right parameterization. And generalized coordinates only allow handling holonomic constraints but not nonholonomic constraints, such as joint limits or velocity-dependent constraints. Maximal coordinate formulations have the advantage that they allow a modular design in which constraints can be added or removed at will. This is vital for an extensible, general-purpose simulation system. A comparison of formulations with generalized coordinates and formulations with maximal coordinates is given in Baraff [Bara96].

Nevertheless, generalized coordinate methods can be a valuable extension for a rigid body dynamics framework. They allow the efficient computation of loop-free articulated rigid bodies where the constraints are permanent. For such cases a parameterization can trivially be found and the generalized coordinate accelerations can be computed in linear time with *Featherstone's*

1 The terms Lagrangian dynamics and Lagrange's equations of motion should not be confused with Lagrange multiplier method of Section 4.6.2, “Simultaneous Force-based Methods”.

Articulated Body Method (ABM) [Feat87]. This method has the advantage that a drifting error between connected rigid bodies does not appear. Kokkevis [Kokk05] describes Featherstone's method and how it can be combined with LCP-based methods to incorporate collisions, resting contacts, and joint limits.

6.3 Other Methods

Redon et al. [ReKC02] have proposed an approach based on *Gauss' principle of least constraints* that is mathematically equivalent to traditional LCP formulations, but is computationally more efficient. This approach only deals with frictionless systems.

Mirtich [Mirt00] applied Jefferson's *Timewarp algorithm* [Jeff85], a technique from parallel discrete event simulation. This method can help to improve performance with large numbers of bodies, even on single-processor machines, and it paves the way for a parallel rigid body simulation. Mirtich introduced *contact groups*[1]. A contact group is a set of bodies in contact. Typically not all bodies are in one contact group at a specific time of the simulation. Instead they form different contact groups, i. e. different clusters of bodies that are spatially separated. Each contact group can be seen as an independent problem. Several smaller problems can be solved in place of one big problem. Of course a performance gain is only achieved if bodies form multiple contact groups and not if all bodies form one big contact group.

A method which is able to handle "crowded" arrangements with one big contact group, like many bodies settling into stacks, was introduced in Milenkovic and Schmidl [MiSc01]. Milenkovic [Mile96] is a predecessor of this method. This predecessor method does not handle friction, collisions, or rotations of bodies. It is a position-based method that tries to change the positions to non-penetrating positions by minimizing potential energy using linear programming. It can effectively simulate piles of spheres. But apart from that the realism is limited. Milenkovic and Schmidl's new method uses an optimization algorithm to move bodies from their unconstrained target position to a position in which no overlap occurs. This solution is attractive for scenes with many bodies where physical accuracy is not crucial.

Kaufman et al. [KaEP05] have recently presented a new approach that uses formulations in the so-called *configuration space* to solve rigid body dynamics including friction.

1 Contact groups are known under different names, for example *islands* in ODE [Smit05].

7 A Unified Framework

"When I am working on a problem I never think about beauty. I only think about how to solve the problem. But when I have finished, if the solution is not beautiful, I know it is wrong."

- Buckminster Fuller

Most publications on rigid body dynamics focus on single detailed problems (like resting contact) or on a single method (such as simultaneous impulse-based simulation). Fitting the parts together is a non-trivial challenge.

This chapter demonstrates how to build a rigid body simulation framework that is independent of the underlying simulation method. This framework shows what the different simulation methods have in common and which parts of the simulation are exchangeable. It can be used to test, combine and compare different methods. The proposed framework is suitable for use in an interactive 3d application – but because of its universal design it better serves as a starting point for application specific tailoring and optimization.

7.1 Requirements

The functional requirements of this implementation are:

- Describe rigid bodies with different mass, material, and shape.
- Compute the motion of the rigid bodies under the influence of different forces.
- Handle collisions, resting contact, general constraints, and error correction.
- Enable visualization of debugging information. This includes drawing the bodies, visualizing velocities, forces, contacts, and more. (Normally, drawing the rigid bodies is not job of the dynamics engine. Drawing methods are only included to facilitate debugging.)

7.2 Used Technologies

The framework was implemented using Microsoft's **.NET** technology and the programming language **C#**. A thorough introduction to these topics is given in Beer et al. [BBMW03]. **Managed DirectX 9** was used as 3d graphics API. Miller [Mill04] is a good guide for Managed DirectX 9. These technologies were chosen because they allow a rapid, object-oriented implementation, and the code is readable and easy to use – which is important because physics alone is difficult enough.

NDoc [NDoc05] was used to create source code documentation. All code and written documentation was managed using **Subversion** [Subv05] as version control system.

7.3 Design Issues

A design pattern that is extensively used in the framework is the *strategy* pattern: An abstract base class defines the interface of an algorithm. Different algorithms can be implemented in the subclasses and can be used interchangeably. More on design patterns and this specific pattern can be found in Gamma et al. [GHJV95].

Only important design issues and interface details are presented in the class diagrams below to keep them readable. The type MatrixNM is used to denote a matrix of size $N \times M$, for example: Matrix33. Matrix describes a matrix without a predefined number of rows and columns. Vector3 describes a 3-dimensional vector and Vector a vector without predefined length.

The following sections show the structure of the implemented framework. The classes are presented bottom up. First the objects that can be simulated (like rigid bodies, forces, and constraints) are introduced. Then the classes that implement the strategies for integration, collision detection, constraint solving, etc. are shown. Finally the class Simulation is presented, which combines everything.

7.4 Simulation Objects

Simulation objects describe what is being simulated. This includes rigid bodies, forces, and constraints. The simulated objects have several properties in common that are summarized in the base class SimulationObject (Figure 7.1).

To uniquely identify each object, it automatically gets a unique id when it is created. The user may additionally set a name for the object, which simplifies debugging. IsActive can be set to false to ignore objects in the simulation.

Each object stores a reference to the simulation to which it belongs. Each simulation object belongs to exactly one simulation. Multiple simultaneous simulations cannot share simulation objects.

The method Render is used to draw debug information. A force effect can visualize the force vector and magnitude. A rigid body can visualize its mass properties, velocities, if it is sleeping or not, etc. A Geo object presents the shape of a rigid body and can render a representation for the

shape (for example a box or a sphere). A constraint can draw its local coordinate system and its current constraint deviation. These are just examples for reasonable debug information. This information is only drawn if the flag DebugIsActive is set to true.

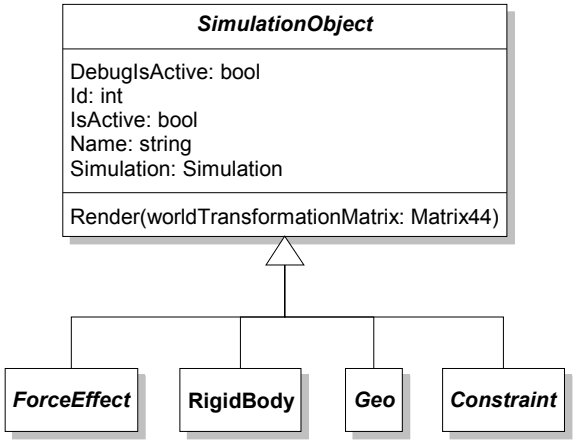

Figure 7.1: Class SimulationObject.

7.4.1 Rigid Bodies

The rigid bodies are the leading actors in the simulation. The class RigidBody is shown in Figure 7.2. It contains properties to describe motion, shape, material, mass, and other attributes.

Motion Properties

The interface IMoveable contains the properties of an object that can move in the simulated world: position $\vec{x}(t)$, orientation $\vec{q}(t)$, linear velocities $\vec{v}(t)$, and angular velocities $\vec{\omega}(t)$. Position and orientation define the body coordinate system relative to the world coordinate system. The interface IMoveable further contains several methods to convert positions and vectors between body space and world space. Additionally the rigid body contains properties for the linear momentum $\vec{p}(t)$ and angular momentum $\vec{L}(t)$.

The method GetFirstOrderDerivatives is very important for the numerical integration of position and velocity. It returns the first order time derivative of the motion state. The equations of motion show how the first order time derivatives can be computed. The derivatives are (using momentums instead of velocities):

$$\frac{d}{dt}\begin{pmatrix}\vec{x}\\ \vec{q}\\ \vec{p}\\ \vec{L}\end{pmatrix}=\begin{pmatrix}\dot{\vec{x}}\\ \dot{\vec{q}}\\ \dot{\vec{p}}\\ \dot{\vec{L}}\end{pmatrix}=\begin{pmatrix}\vec{v}\\ \frac{1}{2}\vec{\omega}*\vec{q}\\ \vec{F}\\ \vec{\tau}\end{pmatrix} \tag{7.1}$$

The total forces and torques on the rigid body are accumulated in this method. The user can explicitly set a force that should act on the rigid body in UserForce. Force effects (like gravity) are discussed in Section 2.6, "Forces". The total force is the sum of all constraint forces (of force-

based simulation methods) plus all forces set by the user plus all forces set with force effects. The computations for torques are similar.

The flag IsFixed can be set to make *static bodies*[1] (also called *fixed*, *scripted,* or *procedural bodies*). An example of a fixed body is the immovable ground plane. "Fixed" does not mean that they do not move, it means that the velocity is not changed by the simulation. The user can set any desired velocity, and the body will move with this velocity and push other bodies out of its way. In collisions these objects are handled as if they have infinite mass.

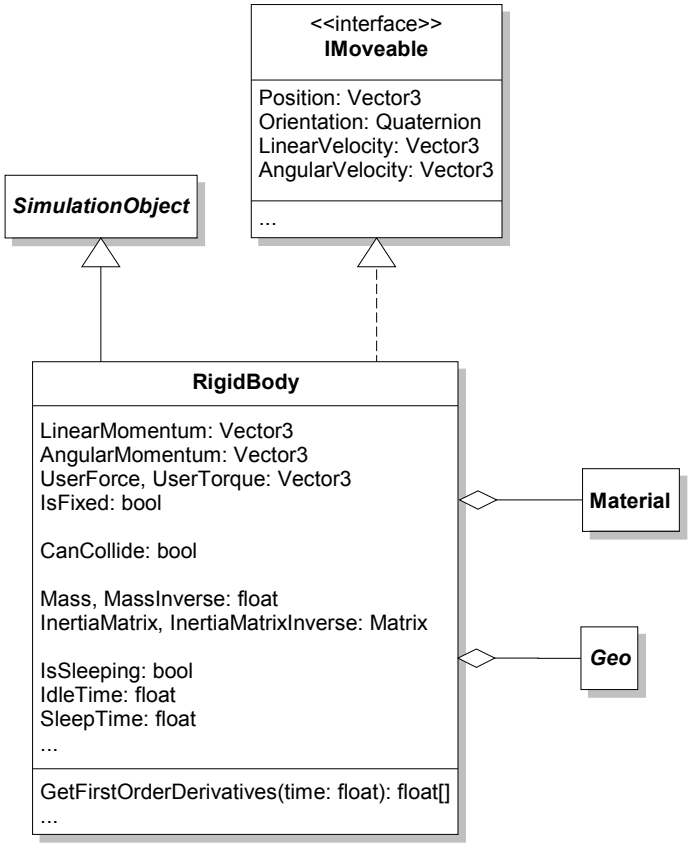

Figure 7.2: Class RigidBody.

Shapes

Each rigid body has a shape. To speed up and simplify calculations the shape of the rigid bodies are assumed to be simple shapes or combinations of simple shapes. Common shapes are: box, sphere, cylinder, capsule, convex hull, and combinations of these shapes. The shape is needed to calculate the mass properties and to detect collisions. Therefore there can be two distinct shapes: A detailed one to calculate mass properties and another one optimized for collision detection.

Also unusual shapes can proof useful for rigid body simulation: Planes can be used to describe limits of the simulation world (for example a ground plane). Rectangles can be used to describe thin objects (for example a postcard).

The center of the shape does not need to be identical with the center of mass of a body. This can be required for

1 The opposite are *dynamic bodies,* which are controlled by the simulation.

- bodies with non-uniform mass distribution, for example: a block with an upper half of wood and a lower half of iron.
- complex bodies that are approximated by a simple shape, for example: a bottle that is approximated by a cylinder shape.
- other reasons: A developer could set the center of mass to a lower position to avoid that this body falls over to easily.

The class Geo (Figure 7.3) contains information that is necessary for the collision detection. Geo has a reference to the rigid body associated with this Geo. HasMoved returns true if the body has changed its position or orientation since the last run of the collision detection. And BoundingSphereRadius returns the radius of a bounding sphere.

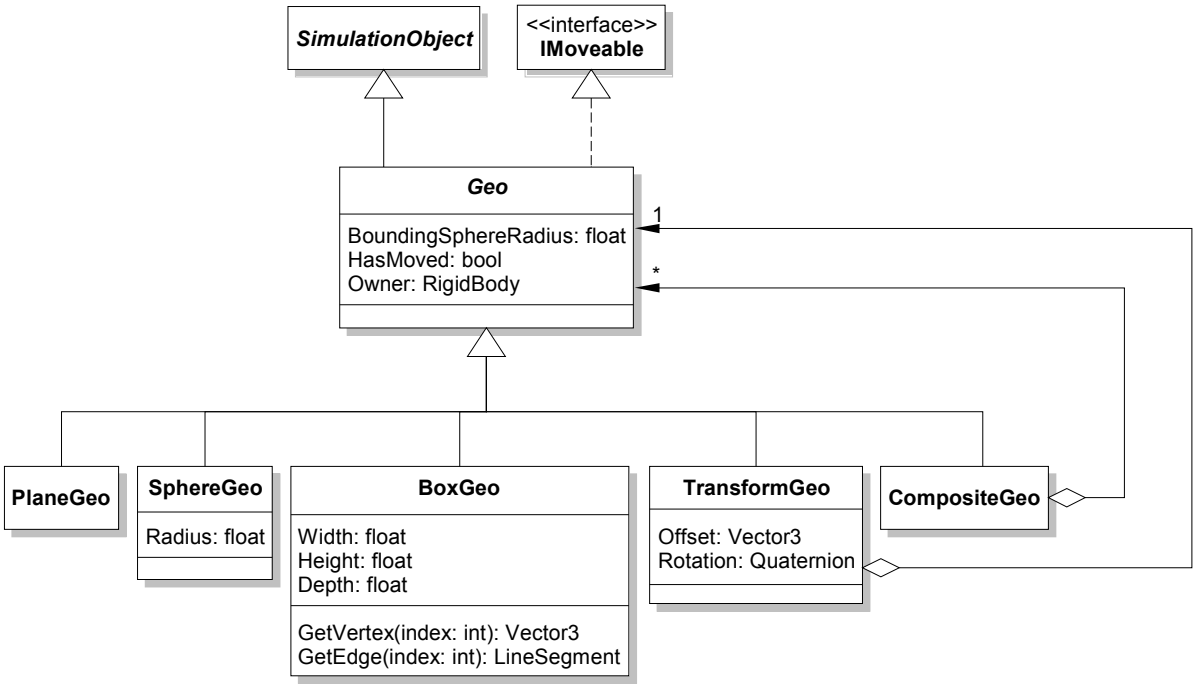

Figure 7.3: Geo classes

Only simple shapes are implemented at the moment. PlaneGeo describes a plane, where the volume on one side of the plane is considered to be solid. This is used to describe the ground plane and walls. SphereGeo represents a sphere and BoxGeo a box.

The center of the shape is normally identical with the center of mass of the rigid body, and the local coordinate system of the shape is aligned with the local coordinate system of the body. An instance of the class TransformGeo can be used to change this. This class implements the *decorator* design pattern. TransformGeo has a reference to a Geo and moves and rotates the local coordinate system of the shape relative to the rigid body.

CompositeGeo can be used to describe a shape that consists of several other shapes (*composite* design pattern). One example is the table in Figure 7.4. This table is one CompositeGeo that is positioned at the center of mass of the rigid body. It consists of five TransformGeos. Each TransformGeo owns a BoxGeo, which is moved to the relative position in the composite shape. The Rotation property is not needed in this example; it would be needed if the table legs should be inclined.

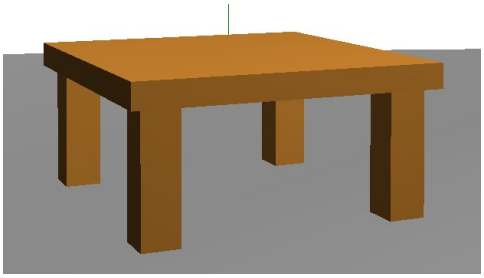

Figure 7.4: A composite shape.

The flag CanCollide of the class RigidBody can be set to false to ignore the body in the collision detection.

Material properties

The material properties are stored in a Material object (Figure 7.5).

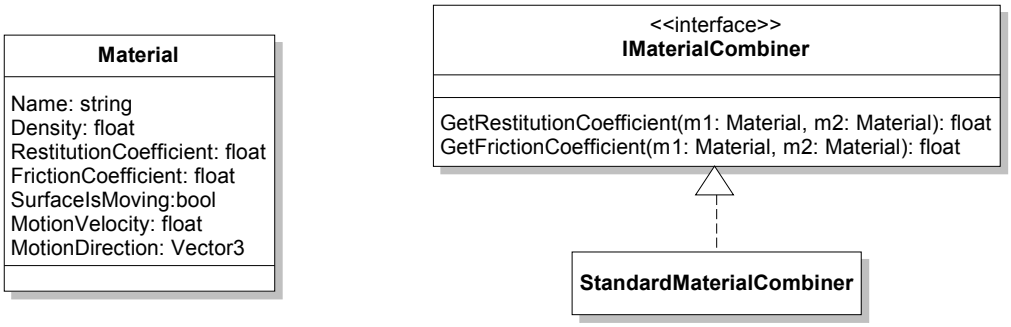

Figure 7.5: Material classes.

Each material has a name (for example: "iron", "glass"). Density contains the uniform density of the body. A restitution coefficient and a friction coefficient are needed for each contact between two materials. For a given contact a class that implements the interface IMaterialCombiner is used to retrieve these coefficients. The methods of this class take both materials and find the coefficients for the material combination in a predefined table – since there is no way to compute the exact coefficients.

A simpler method is to assign a coefficient to each Material object (properties RestitutionCoefficient and FrictionCoefficient). The minimum of the restitution coefficients and the maximum of the friction coefficients are used in a contact. This is done in StandardMaterialCombiner. This class can also be configured to return the average or product of the coefficients.

SurfaceIsMoving can be set to create a rigid body with a moving surface. This is a trick to create objects like conveyor belts. When collisions or resting contacts are computed, it is assumed that all points on the rigid body move into the direction MotionDirection with a speed of MotionVelocity.

Mass Properties

The mass and the inertia matrix in body space are computed when the body is created. It is recommendable to additionally store the inverses of these quantities because they are needed frequently.

The mass m of a body can be calculated by equation (2.39). In most cases we can assume that the rigid body has uniform mass. Then the equation reduces to: $mass = density \cdot volume$.

Some rigid bodies are more complex and maybe consist of different materials. These bodies can often be divided into a set of simple shapes. For example, the table in Figure 7.4, consists of five simple boxes. The mass and center of mass can be calculated for each of these shapes. The total mass is the sum of these mass points, and the center of mass of the table can be calculated.

The same accounts for the mass moment of inertia. The inertia tensor can be computed for each sub-body of a complex body. The corresponding inertia tensors for the center of mass of the composite body can be found using equations (2.54) and (2.56). Finally all inertia tensors are summed up.

Mirtich [Mirt96a] and Eberly [Eber04] show how to compute mass and mass moment of inertia for polyhedra with uniform density.

The principal axis theorem says that there is a set of local body axes for which the inertia matrix is a diagonal matrix. In a preprocessing step this orientation of the local body space axes and the corresponding diagonal inertia matrix can be calculated.

Many times it is necessary to deal with fixed bodies. A body can be made immovable by setting the mass and inertia matrix to infinity and the inverses of the mass and inertia matrix to zero.

Simulation Properties

IsSleeping, IdleTime, and SleepTime are managed by the sleeping method. These properties tell whether a body is sleeping and how long it has been idle or sleeping.

Default Rigid Bodies

Two bodies are always added to the simulation: an abstract world object and the ground plane. The world is represented by an immovable rigid body that cannot collide with other bodies. This body is called World. Representing the empty space as a rigid body has the advantage that all constraints can be expressed as two-body constraints. For example, a body can be fixed to a point in space with a ball-and-socket joint, or a body can be hung on a spring in the air. This helps to treat all constraints uniformly.

The ground plane is the second default rigid body. It is an immovable rigid body with a PlaneGeo shape.

7.4.2 Forces

Each external force is described by a class derived from ForceEffect.

Some forces affect all bodies in the scene, for example: gravity, fluid dynamic drag, and an explosion. Other forces (like springs) act on a pair of bodies. Therefore a force effect has a reference to a collection that contains all affected bodies. This collection can be a .NET ArrayList.

Two additional collections have been implemented: BodyPair contains exactly two rigid bodies. This class determines the body pair of spring. ExclusionBodySet is similar to a normal ArrayList. But in contrast to a normal ArrayList, it contains bodies that are ignored by the force effect. All

bodies of the simulation are affected by default – as it is the case for gravity or drag. If a body should be ignored, it can be added to ExclusionBodySet.

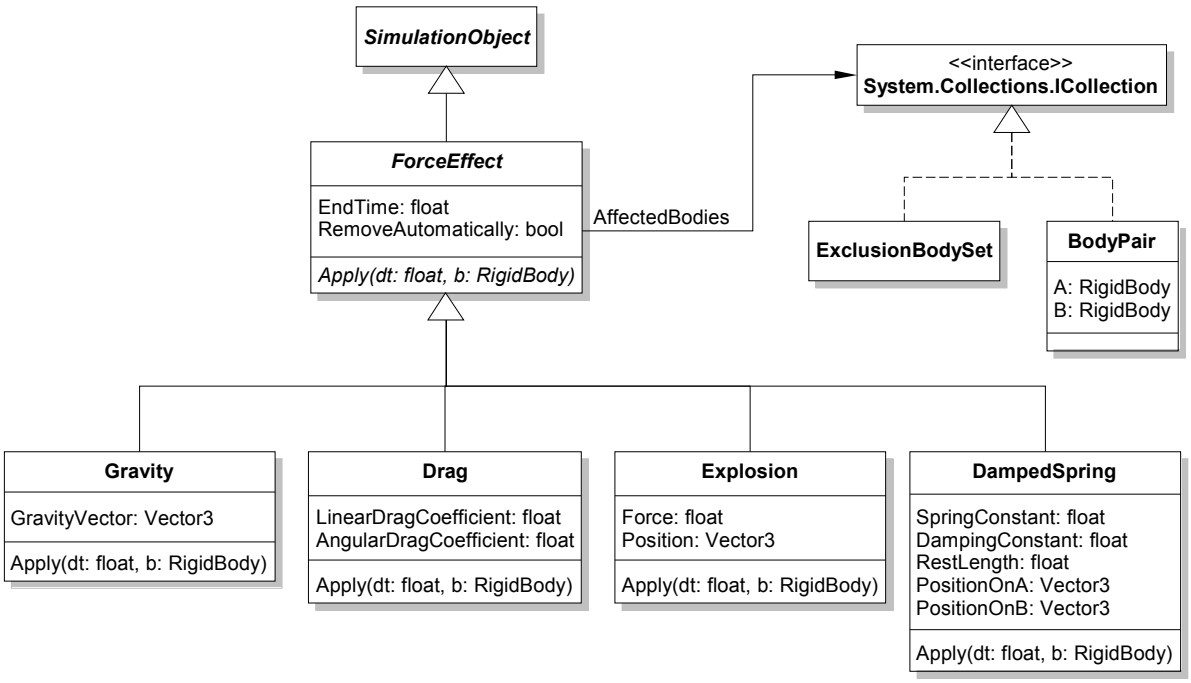

Figure 7.6: ForceEffect classes.

The method Apply applies a force to the bodies in the AffectedBodies collection. The force is computed, it is split into a force and torque regarding the center of mass, and added to the current internal force and torque value of a rigid body.

A force effect can end after a certain time. This time can be set in EndTime. Finished force effects are automatically removed from the simulation if RemoveAutomatically is set to true.

Four force effects have been implemented: Gravity, Fluid dynamic drag, explosions, and damped springs.

Explosion applies a force that pushes bodies away from the center of explosion. The force magnitude is given by Force, and the center of explosion is given by the vector Position. An explosion force effect typically ends after a short time.

DampedSpring can be used to apply a spring element, a damper element, or the combination of both: a damped spring. A spring is fixed between two bodies. The properties PositionOnA and PositionOnB are the body space positions where the spring is attached.

Two force effect objects are added automatically when the simulation is reset: One instance of Gravity and one instance of Drag.

7.4.3 Constraints

Class Constraint

Constraint is the base class of all constraints (see Figure 7.7). Each constraint is defined on two

rigid bodies A and B. Contacts between A and B are ignored if BodiesCanCollide is true. The constraint anchor points are AnchorA and AnchorB. The constraint coordinate system on A is defined by the vectors XAxisA, YAxisA, and ZAxisA. XAxisB, YAxisB and ZAxisB define the coordinate system on B. ERP is the error reduction parameter.

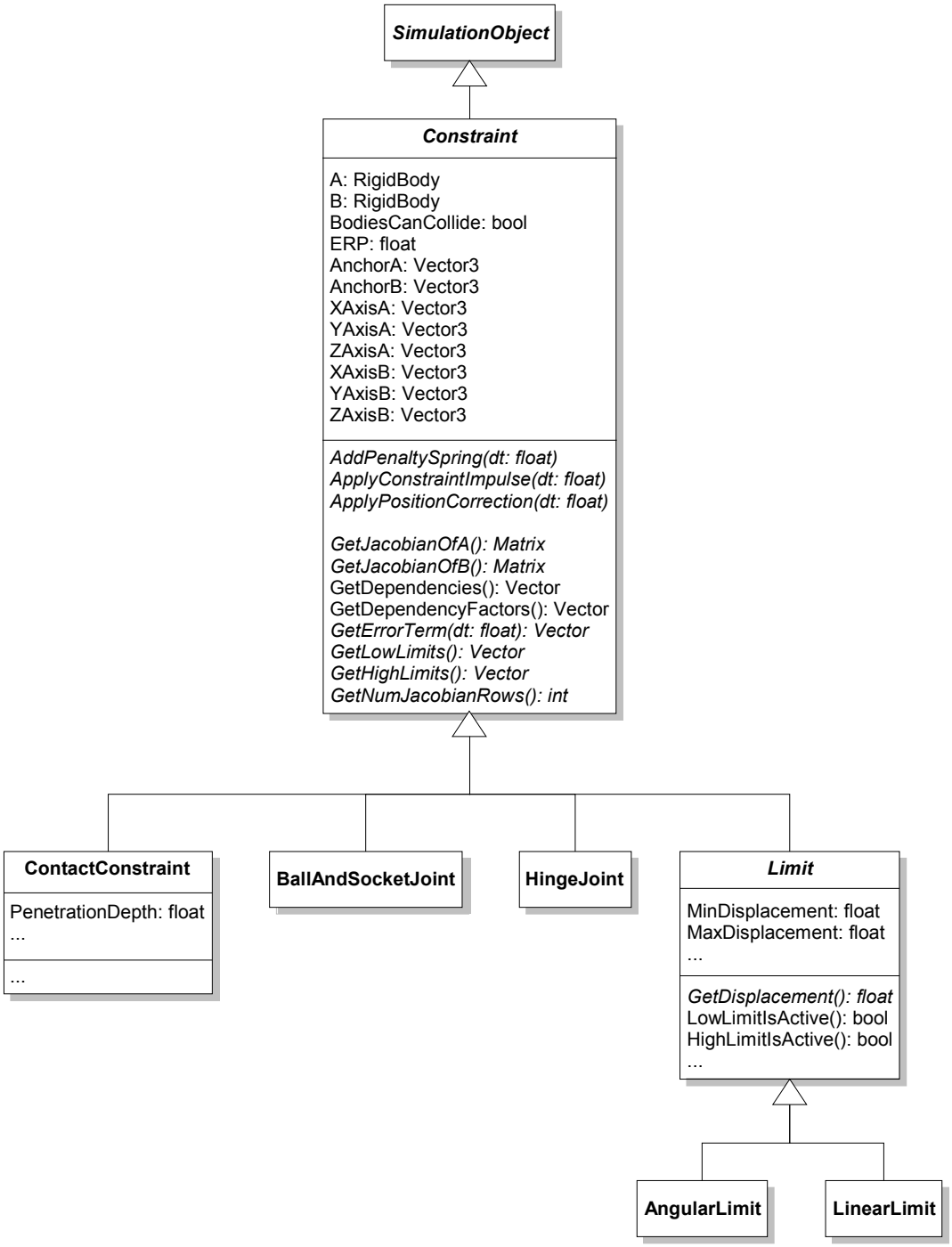

Figure 7.7: Constraint classes.

Constraint further contains all ingredients that are necessary for the different simulation methods. The class contains one abstract method for each sequential constraint solver method: AddPenaltySpring adds penalty springs for a penalty method. A penalty spring is an instance of

DampedSpring that is added to the simulation for one time step. ApplyConstraintImpulse applies a constraint impulse to the bodies. And ApplyErrorCorrection makes a position correction of both bodies.

The other methods implement the ingredients for the LCPs of simultaneous methods. GetJacobianOfA and GetJacobianOfB return the current values for the constraint Jacobian matrix. These matrices are $\boldsymbol{J}_{ki}$ and $\boldsymbol{J}_{kj}$ of equation (4.15).

GetDependencies and GetDependencyFactors return vectors to specify *dependent limits*. They are only relevant for friction in contact constraints, see equation (4.92). Let's assume that dependencies is the vector returned by GetDependencies and dependencyFactors is the vector returned by GetDependencyFactors. These vectors contain one element for each row of the constraint Jacobian. dependencies[i] contains the number of the row that determines the limits of row *i*. If the constraint equation in row *i* does not have dependent limits, dependencies[i] contains a constant that indicates "no dependency". dependencyFactors contains the factors for dependent limit. The LCP solver will use the following limits for the constraint force of row *i*:

$$lo_i = - \text{dependencyFactors[i]} \cdot \lambda_{\text{dependencies[i]}}$$
$$hi_i = + \text{dependencyFactors[i]} \cdot \lambda_{\text{dependencies[i]}}$$

GetErrorTerm returns $\vec{d}$ of the MLCP (4.87-4.88). The name "error term" reflects the fact that $\vec{d}$ normally contains a value for Baumgarte stabilization; only in collisions it contains the desired post-collision velocities. GetLowLimits and GetHighLimits give $\vec{lo}$ and $\vec{hi}$. The number of constraint rows that this constraint adds to the Jacobian of the system is given by GetNumJacobianRows.

Implementing Constraints

Deriving from Constraint allows creating new constraints. Several constraints have been implemented in this work.

ContactConstraint is used to describe colliding and resting contacts. These constraints are automatically added by the collision detection and not by the user.

Two joint types are implemented: BallAndSocketJoint and HingeJoint.

Limit is the base class linear and angular limits. A limit is not always active, similar to a contact constraint. Limits measure displacements with GetDisplacement, which has to be implemented in subclasses: An AngularLimit measures the angle between two bodies, a LinearLimit measures the distance between two constraint anchor points. LowLimitIsActive and HighLimitIsActive return true when MinDisplacement or MaxDisplacement is reached. The constraint solver has to care about limit constraints only if bounds are reached.

Using Constraints

Constraints are simple to use. When the user wants to add a new constraint, she creates a new instance of BallAndSocketJoint, HingeJoint, AngularLimit, or LinearLimit. Then she specifies the properties A, B, BodiesCanCollide, ERP, AnchorA, AnchorB. The six axes XAxisA, XAxisB, etc. only have to be specified for constraints that constrain orientation, such as ContactConstraint, HingeJoint or AngularLimit: ZAxisA has to be set for ContactConstraints to set the contact normal vector. For HingeJoints XAxisA has to be set, which describes the hinge rotation axis. The other axes are computed automatically. Then the Constraint object is added to the simulation and will be handled by the constraint solvers.

7.5 Simulation Strategies

Simulation objects define the scenario that is simulated. The classes in this section will provide the tools to simulate this world. These classes are based on the *strategy* design pattern.

7.5.1 Integrators

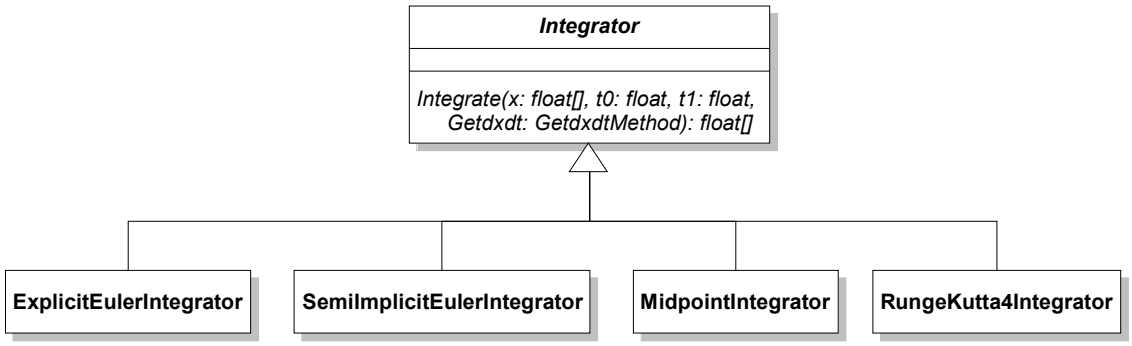

Figure 7.8: Integrator classes.

An integrator is an object that performs numerical integration. It contains the method Integrate. The argument x specfies the state at time t0. Integrate computes the state at time t1 by numerical integration and returns this new vector. The vector x contains several physical quantities, and the first order time derivatives are needed to compute the new state. Therefore a C# delegate Getdxdt is handed over to the method[1]. The exact type of the function that Getdxdt points to is:

```
float[] GetdxdtMethod(float[] x, float t)
```

This method takes x and assumes that this is the current state at time t. It computes the first order time derivatives for the quantities and returns the vector of time derivatives. This method has to be implemented by the object that wants to use the integrator.

ExplicitEulerIntegrator is the simplest implemented numerical integration method. Integrate of ExplicitEulerIntegrator returns (written in pseudo-code):

```
x + (t1 – t0) * Getdxdt(x, t0)
```

Other implemented numerical integration methods are the semi-implicit Euler method, the midpoint method, and the Runge-Kutta method of order 4. Getdxdt is very important for the latter two methods because these methods compute intermediate state vectors at times between t0 and t1. They need the first order time derivations for the intermediate states and times, thus they will call the method Getdxdt several times in one integration step.

Section 7.5.7, "Simulation", will explain in detail how an integrator is used to update the velocities and positions.

1 A C# delegate is similar to C++ function pointer.

7.5.2 Collision Detection

The role of the collision detection in the simulation is to find all contacts at the current simulation time. Therefore the shapes of the bodies are tested whether they touch or intersect. A ContactConstraint is reported to the simulation for each contact found.

A CollisionDetectionStrategy object (see Figure 7.9) is responsible to find all contacts and add them to the simulation. This is done in the method CheckForCollisions. This method calls GetContactSet for all Geos of rigid bodies that can collide. GetContactSet returns a PairContactSet, which is a list of ContactConstraint objects for contacts between two bodies.

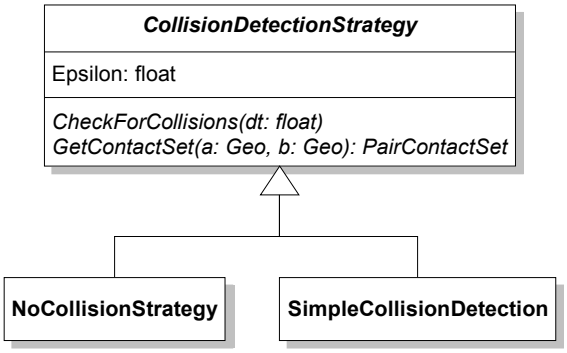

Figure 7.9: Collision detection classes.

NoCollisionStrategy simply ignores all collisions and can be used for debugging. SimpleCollisionDetection implements a simple collision detection method that should be improved in the future. In the broad phase a bounding sphere check is made to see which bodies might be in contact. In the narrow phase the contacts for bodies are computed.

The property Epsilon is a distance threshold. If a distance between two bodies is below Epsilon, a contact is reported. Epsilon is usually chosen to be three to four orders of magnitude smaller than the dimensions of the bodies in the simulated scene [MiCa94].

7.5.3 Constraint Solvers

A constraint solver takes a set of constraints from the simulation and handles these constraints. How this is done depends on the constraint solver method, which is implemented in a subclass of ConstraintSolverStrategy. All solvers can handle collisions, resting contacts, general constraints, and error correction – except the position-based solvers, which can only be used for error correction.

NoSolver is the simplest implementation – it ignores all constraints.

PenaltyMethodSolver uses the penalty method. It simply calls the AddPenaltySpring method of each constraint. At the end of method Solve several DampedSpring force effects are added to the simulation. These springs represent the penalty springs for the current time step. All penalty springs are automatically removed at the end of the time step.

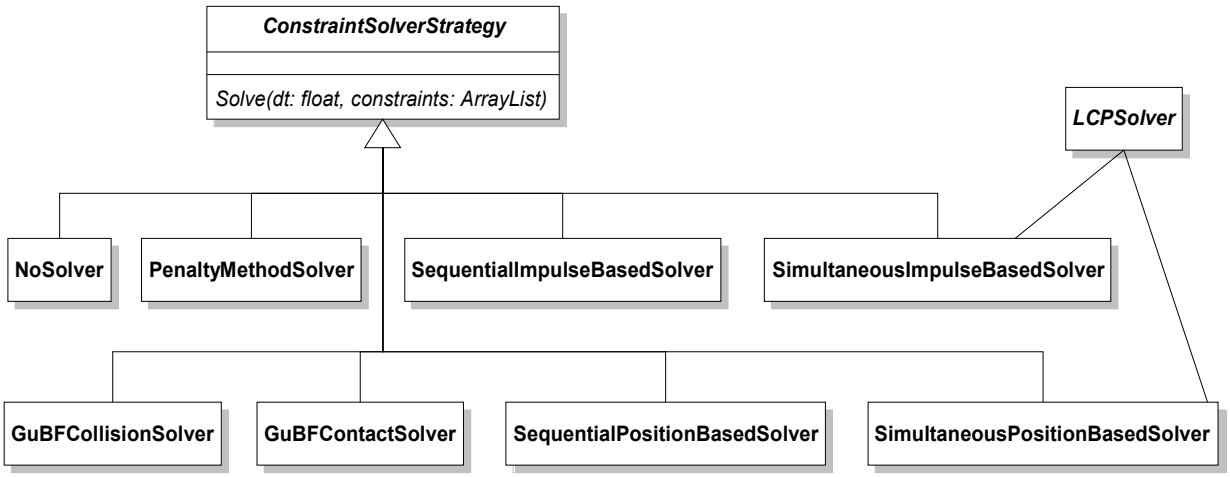

Figure 7.10: Constraint solver classes.

SequentialImpulseBasedSolver, GuBFCollisionSolver, and GuBFContactSolver are three sequential impulse-based methods. All of them call the method ApplyConstraintImpulse of the constraint – but the order in which the constraints are processed is different: SequentialImpulseBasedSolver calls ApplyConstraintImpulse for all general constraints in an arbitrary order. ContactConstraints are first sorted by a criterion. Different criteria are used in the literature, and to test different methods, the contacts can be ordered ascending or descending by penetration depth, relative normal velocity, by a random order, or they can be left unsorted. GuBFCollisionSolver handles constraints in arbitrary order and contact constraints in the order that is suggested in Guendelman et al. [GuBF03]. GuBFContactSolver uses the method for resting contacts suggested in the same paper. GuBFCollisionSolver is faster but not as accurate as GuBFContactSolver.

SequentialPositionBasedSolver calls ApplyPositionCorrection of the constraints to perform a position correction.

All sequential solvers have to perform several iterations over all constraints, in contrast to simultaneous solvers, which compute the result in one step. SimultaneousImpulseBasedSolver builds an impulse-based MLCP with Baumgarte stabilization. SimultaneousPositionBasedSolver builds the MLCP (5.9-5.10) for position correction.

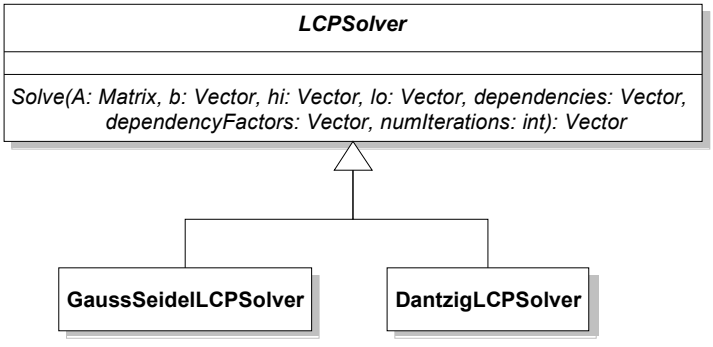

Figure 7.11: LCP solver classes.

The class LCPSolver solves a MLCP. Two different solvers have been implemented: GaussSeidelLCPSolver uses a *projected Gauss Seidel method* [Erle04], [Catt05]. This is an iterative method. It is fast and easier to implement. DantzigLCPSolver implements a LCP solver that is based on *Dantzig's method* [Bara94], [Smit05]. *Dantzig's method* is not as fast as an iterative method,

but it returns an exact solution instead of an approximation. It is more difficult to implement, especially if the involved matrices can be singular. For interactive applications an iterative solver is recommended.

7.5.4 Time Step Strategies

The time step strategies of Section 3.8, "Time Step Methods", are implemented as subclasses of the class TimeStepStrategy. This class has a method Step that performs one time step. Section 7.5.7, "Simulation", goes into detail on these classes.

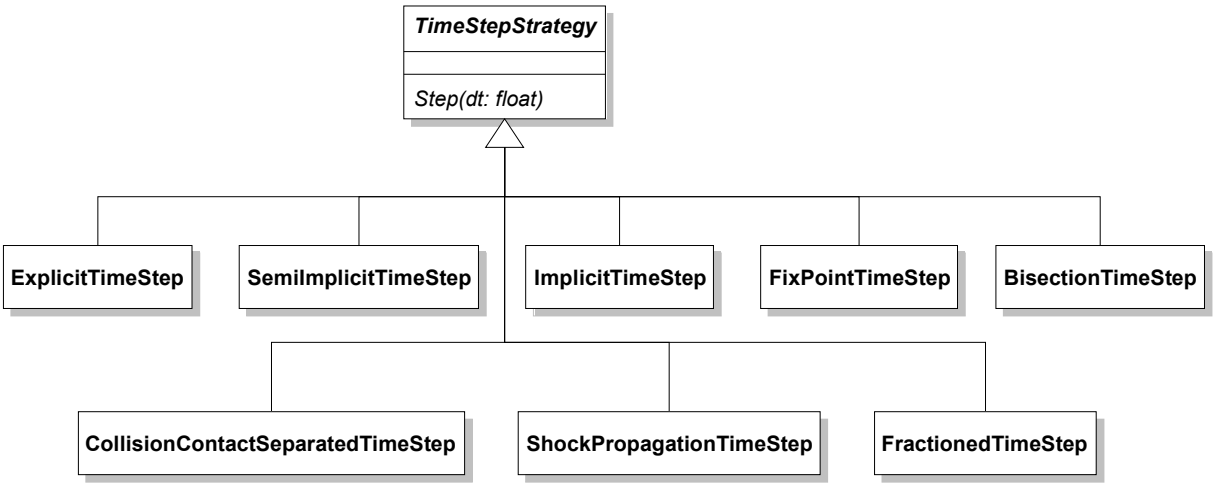

Figure 7.12: Time step method classes.

7.5.5 Sleeping

A SleepStrategy (Figure 7.13) takes a list of rigid bodies and sets the IsSleeping property of the rigid bodies. The method HandleSleep has to be implemented in the subclasses. It returns true if all bodies in the given list are sleeping.

NoSleepStrategy is the simplest implementation: Bodies are never sleeping. Sleeping is simply ignored.

ThresholdBasedSleep compares some body properties against a given threshold. The method IsIdle returns true if the body is not moving. This method is implemented in the subclasses of ThresholdBasedSleep. VelocityBasedSleep tests the linear and angular velocity to see if the body has come to rest. EnergyBasedSleep tests the kinetic energy of the body.

If IsIdle returns true, the body is not moving. This condition is not sufficient to set the body to sleep. For example, a ball thrown in the air will be motionless at the top of its trajectory, but it will gain velocity in the next time steps. Therefore the motion of a body has to be observed for a defined time span.

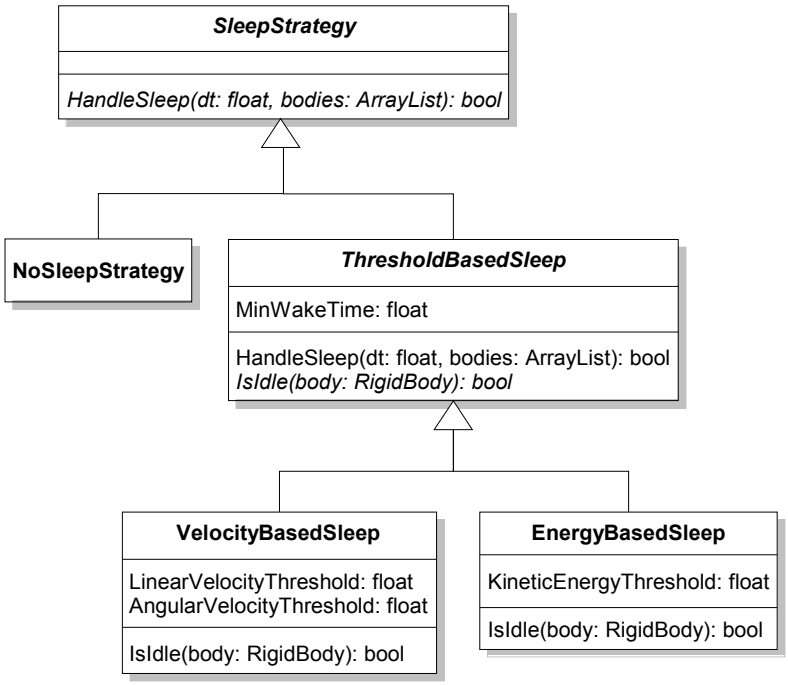

Figure 7.13: SleepStrategy classes.

Schmidl [Schm02] measures the body's kinetic energy for a certain time. If the kinetic energy is below a threshold for a given duration, the body is set to sleep. The threshold for the kinetic energy is determined by the kinetic energy the body would gain in one time step in the gravity force field. Erleben [Erle04] suggests adding a relative test: The body's kinetic energy must be below a threshold and must be non-increasing.

Using velocity or kinetic energy as a measure has different effects. Velocity refers to the visual movement of a body: If the user does not see any movement, the body can be deactivated. Kinetic energy refers to the physical result: If all kinetic energy is consumed, the body stops moving and is set to sleep. To see the difference, let's imagine a rolling cylinder: If it moves very slowly, it could be set to sleep in a velocity-based sleeping strategy. In a kinetic energy-based strategy the mass of the body is indirectly regarded. Even if the cylinder moves slowly, it could still have a lot of kinetic energy and thus the potential to push other bodies. So it would be a bad idea to disable it too early.

Another advantage of energy-based methods is that linear and angular motion are combined; only one threshold parameter is needed. Erleben [Erle04] suggest to measure kinetic energy divided by the mass of the body.

Implementing sleeping functionality is not easy. Errors in the simulation might prevent bodies from coming to rest – some bodies always experience a small movement. The efficiency of a sleeping method also depends on the current simulated scenario. If the simulation scenario is known beforehand, a sleep strategy and the used parameters can be optimized to set the bodies to sleep more quickly.

Used Sleeping Strategy

Here, the following strategy was implemented and has led to good results in the tested scenarios:

A body is set to sleep if `IsIdle` returns `true` for `MinWakeTime` seconds. The limit `MinWakeTime` is

used if the body has at least 3 contacts. Otherwise a greater limit is used (for example $10 \cdot$ MinWakeTime).

Therefore the number of contacts of a body have to be counted. If a body has more than 2 contacts over several time steps, it is more likely that this body has come to rest. For example, a box that touches the ground with only one corner or one edge (two contacts) will tip over in the next time steps. But if it has three contacts, it is probably in a stable position. This strategy assumes that the collision detection provides reasonable contact information. A ball joint is counted as one additional contact, and a hinge joint as two additional contacts. This heuristic method allows using a small time limit and determining quickly if a body is resting.

Forces and constraints can wake bodies up:

- The currently acting force on the body is compared to the force that was acting in the last time step. If the difference exceeds a given threshold, a body wakes up.
- If body *A* was in contact with body *B,* and *B* moved away, then body *A* wakes up. This is necessary because body *A* may remain in an unstable position without *B*.
- If body *A* is connected to body *B* with a constraint and *B* is significantly moving, then *A* wakes up.
- A body has to wake up too if a new constraint for this body is activated. For example, if another body collides with this body, or a new general constraint, like a hinge joint, is added to the simulation.

7.5.6 Validation Rules

The simulation has a list of Validator objects. At the end of each time step the method Validate of each Validator object is called. Validators are often used to make sure that the simulation stays in a valid state. Two examples are drawn in Figure 7.4.

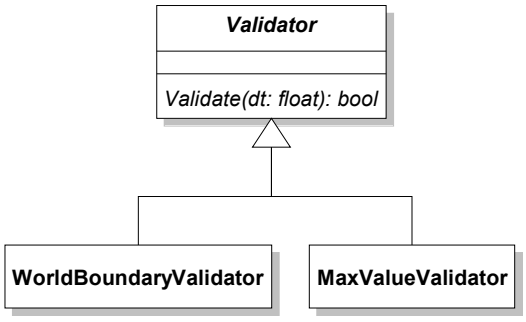

Figure 7.14: Validator classes.

WorldBoundaryValidator checks if all objects are within the borders of the world – the world borders are given by the extent of the Geo of the RigidBody object World. World can, for example, have a SphereGeo with a radius of infinity. But in most applications the focus lies on a limited visible area. Bodies that leave this area do not have to be simulated any more. Therefore World can have BoxGeo as its shape with finite extent. WorldBoundaryValidator checks if all bodies are in this volume and removes bodies that have left it.

MaxValueValidator checks if the linear and angular velocities are within a given limit. If a velocity is

too big, it is set to a maximal allowed velocity. This is important because sometimes instabilities cause high velocities. Clamping velocities to reasonable values improves the stability of the simulation.

7.5.7 Simulation

Now it is the time to put the pieces together.

Figure 7.15 shows the heart of the framework, the class Simulation. Simulation has references to several other classes. The classes on the left side of Figure 7.15 describe "how" the simulation is carried out. Their subclasses implement the algorithms that drive the simulation. The classes on the right side describe "what" will be simulated. Hence Simulation is a container for the used methods and the simulated objects.

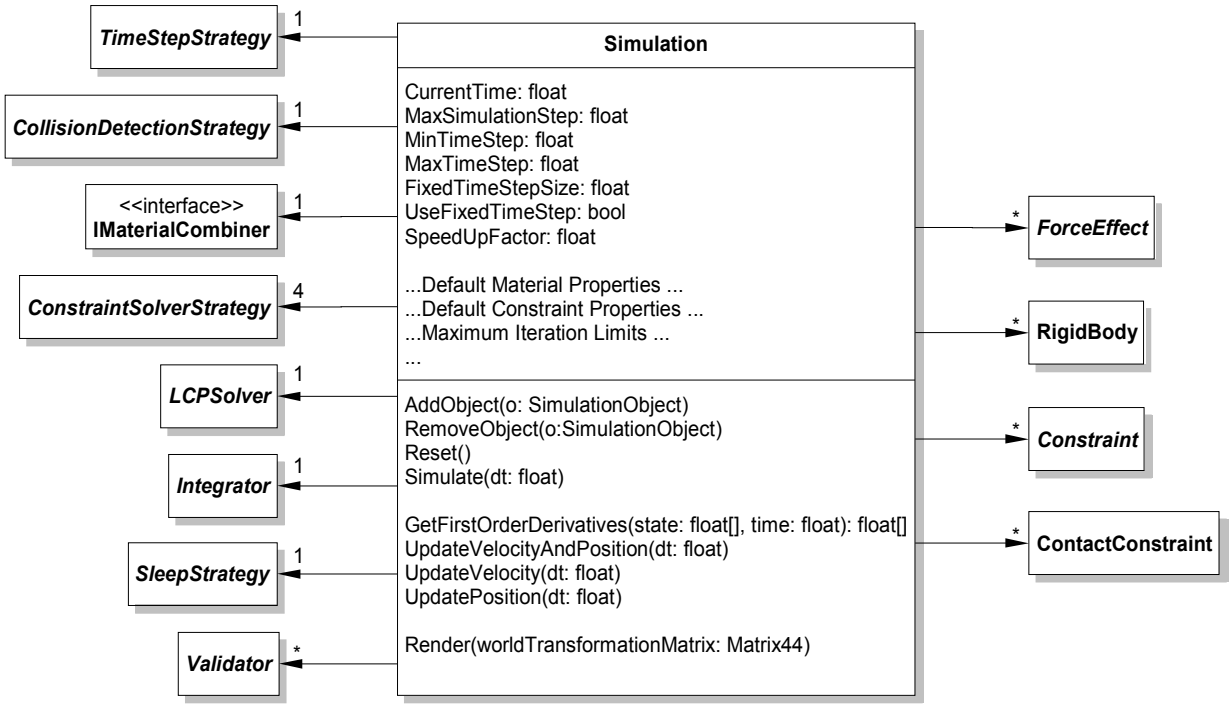

Figure 7.15: Class Simulation.

The simulation has a list of forces, a list of rigid bodies, a list of general constraints, and a list of contact constraints. The methods AddObject and RemoveObject allow adding and removing simulation objects. When a rigid body is removed, the constraints that depend on this body have to be removed as well. Only the list of ContactConstraints is special because this list will be filled by the collision detection.

CurrentTime contains the current simulation time. The method Reset will set CurrentTime to zero and will remove all simulation objects. Reset also sets default forces (gravity and drag) and default bodies (the world and the ground plane).

The simulation is further a container for default settings, like material properties and iterations limits.

The method Render draws debug information to the screen. This method calls the Render method

of the contained objects. Each object can draw its representation and debug information.

Usually, an application has only one instance of Simulation. But sometimes it makes sense to have several instances, for example,

- to divide the simulated world into smaller parts. Each room with objects can be simulated in its own Simulation instance.
- to run simulations in parallel and compare different simulation methods or settings (see Section 8.3, "Example: Parallel Simulation").

Method Simulate()

Simulate is the most important method. This method will be called regularly by the application. It advances the simulation state and time by Δt. Δt is normally the elapsed time since the last call of Simulate. Here, we call this Δt a *simulation step*. This value can vary from call to call. The simulation could use this value to make a single time step. But time steps should be in the bounds of MinTimeStep and MaxTimeStep. Too small time steps waste processing time. Too big time steps may cause instabilities. Therefore, too big simulation steps are automatically divided into smaller time steps.

Time steps can have a variable or a fixed size. The advantage of a variable time step is that the simulation makes big time steps if the application is busy with other tasks. And the simulation makes small time steps if the application has enough time to do so. The disadvantage of a variable time step is that each execution of the application creates different simulation results because the time steps depend on the current CPU utilization. A fixed time step has to be used to get the same results in each simulation run. When UseFixedTimeStep is true, the simulation step is automatically broken down into fixed time steps. The size of the fixed time steps is FixedTimeStepSize.

MaxSimulationStep is the maximum limit for a single simulation step. It has to be limited too if the PC is too slow for real-time simulation; because if the speed of execution is slow, the simulation step is big and many time steps are made. This takes some time and the next simulation step has to be bigger to keep up with the real time. The simulation steps will be growing and the frame-rate will continually decrease. Therefore the simulation steps have to be limited. This will produce a simulation that is running in slow motion – which is still preferable to no simulation.

Sometimes slow motion is actually desired, or the simulation should be executed faster than real-time – because sometimes reality is not exciting enough. This can be done with the property SpeedUpFactor.

In Simulate the current SleepStrategy instance is called to set bodies to sleep or wake bodies up. Then the active TimeStepStrategy instance is invoked to perform a single time step. Afterwards the Validator objects are executed. At the end of each time step the list of current contacts is deleted and objects are prepared for the next time step in a clean up phase.

The Simulate method in pseudocode looks like this:

```
Simulate(dt)
{
    dt = dt * SpeedUpFactor
    IF dt > MaxSimulationStep THEN dt = MaxSimulationStep
    TargetTime = TargetTime + dt
    IF Simulation does not contain non-sleeping bodies THEN CurrentTime = TargetTime
    WHILE CurrentTime < TargetTime
```

```
    {
        dt = TargetTime – CurrentTime
        IF dt > MaxTimeStep THEN dt = MaxTimeStep
        IF dt < MinTimeStep THEN dt = MinTimeStep
        IF UseFixedTimeStep is true THEN dt = FixedTimeStepSize

        IF CurrentTime + dt > TargetTime THEN exit while loop

        Call sleeping strategy
        IF Simulation contains non-sleeping bodies THEN
            call time step strategy to make a time step of size dt
        Call validation rules
        Clean up
        CurrentTime = CurrentTime + dt
    }
}
```

TargetTime contains the time that the simulation should advance to. If the current time step would step beyond TargetTime, then Simulate stops. The remaining time from CurrentTime to TargetTime will be processed in the next call of Simulate.

Method TimeStepStrategy.Step(dt)

The current TimeStepStrategy instance has to perform a single time step. It calls the collision detection, the constraints solvers, and the velocity and position update, as described in Section 3.8, "Time Step Methods".

The simulation contains four ConstraintSolverStrategy objects: one for collisions, one for resting contact, one for general constraints, and one for error correction. This allows combining different constraint methods. For example, Baltman and Radeztsky [BaRa04] use sequential impulse-based methods for collisions and general constraints, the penalty method for resting contacts, and a sequential position-based method for error correction. This flexibility is further helpful to test and develop new simulation methods. Of course, in practice it is seldom necessary to use four different constraint solvers. A constraint solver, like SimultaneousImpulseBasedSolver, can handle all four tasks. It is sufficient to set one ConstraintSolverStrategy reference and ignore the others by setting them to an instance of NoSolver.

The time step strategy calls the method UpdateVelocityAndPosition[1] of Simulation to advance the velocities and positions. This method uses an Integrator object for numerical integration. The motion states (positions, orientations, and velocities/momentums) of all rigid bodies are concatenated to one vector State. The integrator needs a method that computes the first order time derivative of this state vector. This is done with the method GetFirstOrderDerivatives. It calls GetFirstOrderDerivatives of each rigid body and concatenates the results. The integrator call in UpdateVelocityAndPosition looks like

```
NewState = integrationStrategy.Integrate(State, 0, dt, GetFirstOrderDerivatives)
```

The vector NewState receives the new positions and velocities, which are then copied back into the RigidBody objects.

1 The methods UpdateVelocity and UpdatePosition can be used in time step methods where the velocity and position update is done separately.

7.6 Conclusion

The class Simulation is the centerpiece of the framework. It controls all objects that describe the simulated world: rigid bodies, force effects, and constraints. And it uses a set of exchangeable algorithms to perform the simulation. Developers are free to add new shapes, constraints, force effects, and algorithms to the framework.

The simulation objects can be added and removed freely by the owning application. The simulation controls the bodies, but the user can take over control by setting forces directly or by making static bodies (IsFixed property). Composite shapes, constraints, and different force effects allow the adaption to complex simulation scenarios.

Algorithms of different simulation methods have been dissected into independent and swappable parts: integrators, collision detection, constraint solvers, time step methods, sleeping methods, validators. These simulation strategies can be combined freely – even at run-time. This allows to optimize the simulation for different applications or different scenarios within the same application.

8 Evaluation

"Something unknown is doing we don't know what."

- Sir Arthur Eddington, comment on the Uncertainty Principle

This chapter presents applications of the implemented framework. Several scenarios demonstrate different aspects of rigid body simulation. This chapter concludes with some additional lessons learned during the development and testing of the framework.

Performance measures are not given in detail because the simulation methods have not been optimized so far. An important performance factor is the collision detection, which was outside the scope of this work.

The LCP solvers in the examples use 10 iterations. Sequential simulation methods use 4 to 10 iterations per time step. The time step size was set to 0.01 seconds.

8.1 Viewer

A viewer application has been implemented, which allows testing the rigid body framework and comparing different simulation methods and parameters.

This application is based on the DirectX Sample Framework [DXSF05]. The Sample Framework helps to build simple and moderately complex Direct3D applications. It simplifies the Windows and Direct3D APIs for typical usage. This saves time when building tutorials, samples, or prototypes. It also contains user interface support and allows adding control objects, like buttons and check boxes. Figure 8.1 shows a screenshot of the viewer application.

The viewer can execute several test scenarios – some of them will be presented here. Most settings of the rigid body framework can be adjusted: Time step size, time step strategy, friction coefficients, integrator type, constraint solver strategies, iteration limits, and more. The framework allows changing of parameters and simulation methods even while the simulation is running. The simulation runs in real-time or is executed time step by time step. Debugging information and contact sets can be visualized.

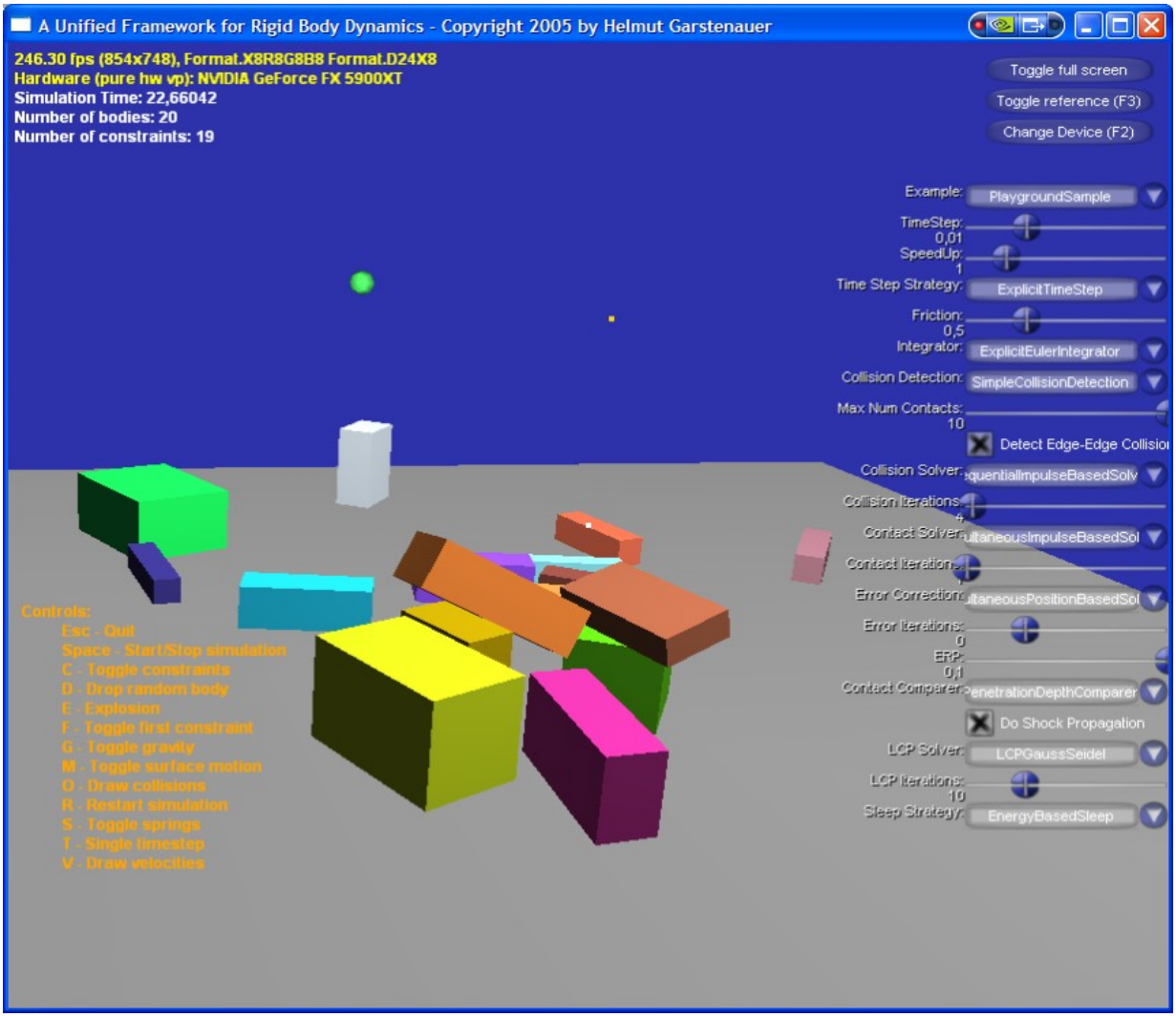

Figure 8.1: Screenshot of viewer application.

8.2 Example: Seesaw

Figure 8.2 shows a scene with colliding contact, resting contact, and general constraints: A board is fixed with a hinge joint on an immovable box to build a seesaw. A heavy and a light box are dropped onto the seesaw. The heavy box catapults the smaller one into a pyramid stack of boxes and the top box is knocked down.

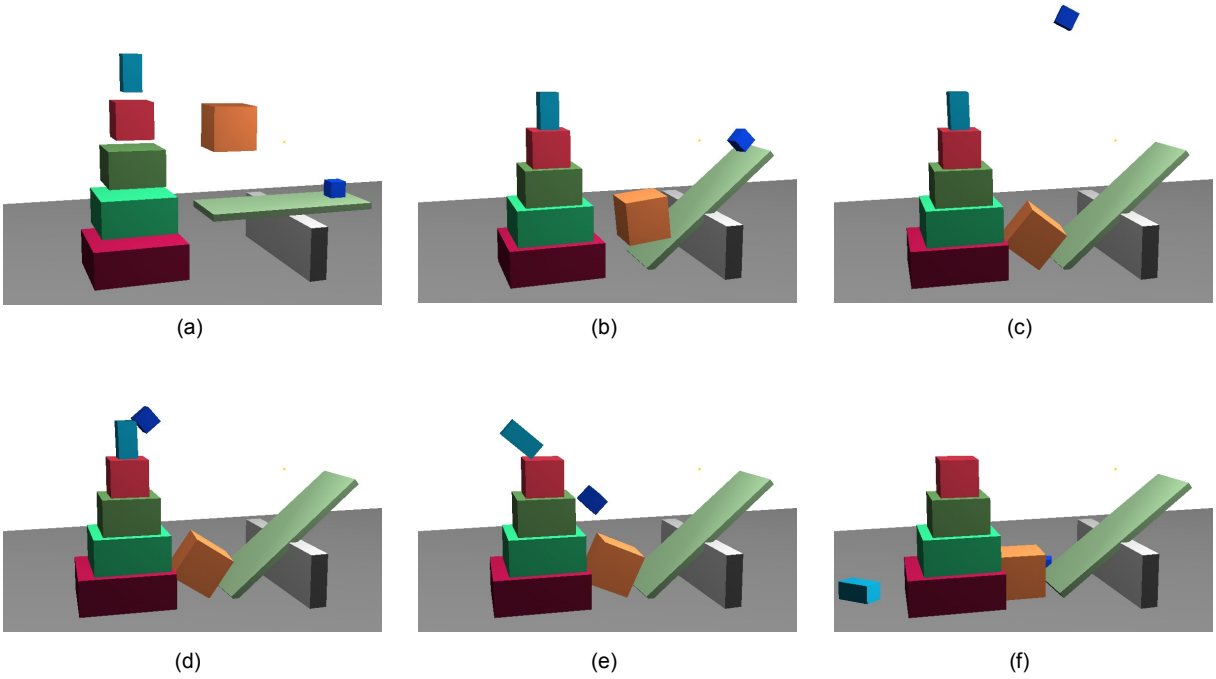

Figure 8.2: A big box catapults a small box, which hits the top box of a pyramid stack.

The code to create the seesaw catapult is simple:

```
Simulation sim = new Simulation();

// Create small box at position (3.5, 4, 0), with no initial rotation and with dimensions (1, 1, 1).
RigidBody smallBox = new RigidBody(sim, new Vector3(3.5f, 4f, 0f), Quaternion.Identity,
        new  BoxGeo(sim, 1f, 1f, 1f));
sim.AddObject(smallBox);

// Create big box.
RigidBody bigBox = new RigidBody(sim, new Vector3(-3.5f, 8f, 0f), Quaternion.Identity,
        new BoxGeo(sim, 2.5f, 2.5f, 2.5f));
sim.AddObject(bigBox);

// Create seesaw board.
RigidBody seeSaw = new RigidBody(sim, new Vector3(0.0f, 3.2f, 0.0f), Quaternion.Identity,
        new BoxGeo(sim, 10.0f, 0.4f, 4.0f));
sim.AddObject(seeSaw);

// Change friction of seesaw board, so that boxes do not simply slide down.
seeSaw.Material.FrictionCoefficient = 0.6f;

// Create immovable box under the seesaw.
RigidBody fixedBox = new RigidBody(sim, new Vector3(0f, 1.5f, 0f), Quaternion.Identity,
        new BoxGeo(sim, 1f, 3.0f, 10f));
fixedBox.IsFixed = true;
sim.AddObject(fixedBox);

// Create a hinge joint that connects the fixed box and the seesaw board
HingeJoint joint = new HingeJoint(simulation);
joint.A = fixedBox;
joint.B = seeSaw;
// Set local position of the joint in the fixed box.
joint.AnchorA = new Vector3(0f, 1.78f, 0f);
// Local position of the joint in the seesaw.
joint.AnchorB = new Vector3(0f, 0f, 0f);
```

```
// Set direction of the hinge axis for each body. (Hinges rotate around the joints x axis).
joint.XAxisA = joint.XAxisB = new Vector3(0f, 0f, 1f);
sim.AddObject(joint);
```

Prior to rendering a frame the application calls the simulation with the elapsed time since the last frame. The application does not have to worry about the fixed time step size, since the time given by the application is automatically broken down into appropriate fixed time steps:

```
// Advance simulation by elapsedTime:
sim.Simulate(elapsedTime);
```

8.3 Example: Parallel Simulation

The framework can run several simulations simultaneously to compare different methods and parameters. Figure 8.3 shows an application where two stacks are simulated, each in its own simulation. The left stack is simulated with a sequential impulse-based method: SequentialImpulseBasedSolver with 10 iterations per time step. The right stack is simulated with the simultaneous impulse-based method: SimultaneousImpulseBasedSolver where the iterative Gauss-Seidel LCP solver makes 10 iterations in each time step. The bodies of one stack cannot collide with the bodies in the other stack because they are simulated in separate Simulation instances.

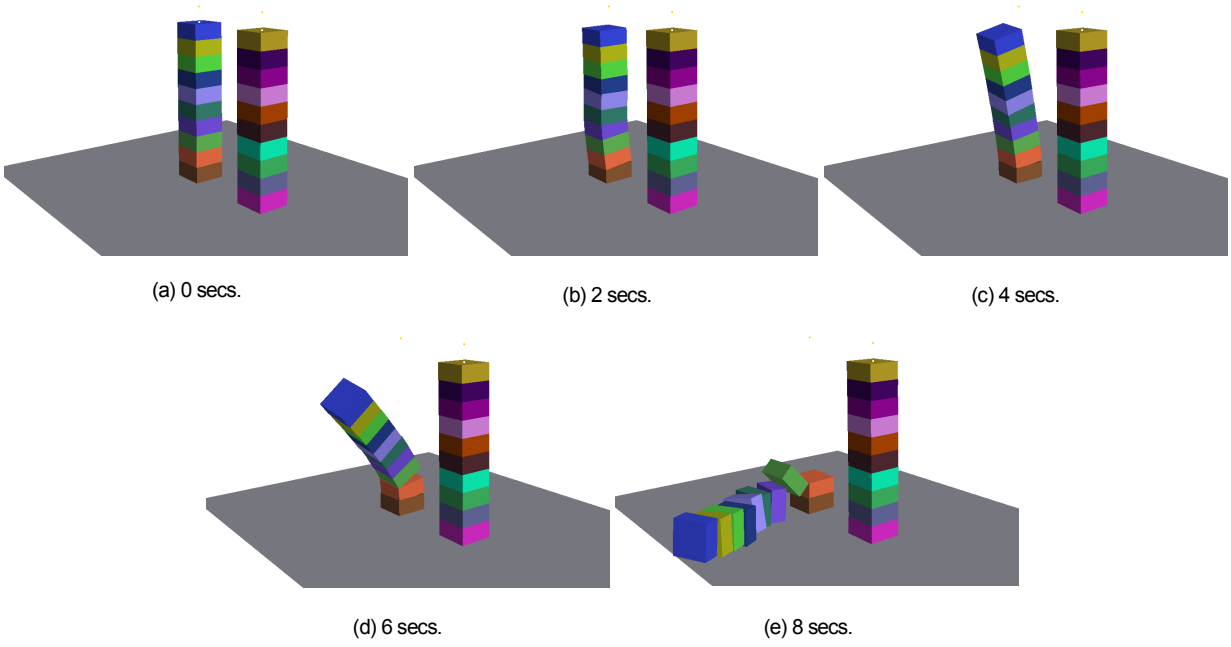

Figure 8.3: Parallel simulation of two stacks.

The left stack shows interpenetrations after 2 seconds of simulated time. After 4 seconds these penetrations cause the stack to tilt noticeably. And finally the stack collapses after 8 seconds - whereas the right stack is stable.

This shows that the sequential impulse-based method does not seem to be appropriate to handle this kind of scenarios – at least not with an explicit fixed time step method. Changing the time step

strategy from ExplicitTimeStep to ShockPropagationTimeStep leads to a stable stack with sequential impulse-based simulation. Further sleeping can be used to give stable results. With an activated sleeping method, the simulation detects the resting objects and deactivates them.

8.4 Example: Error Correction

Figure 8.4 (a) shows a scene in which boxes are incorrectly initialized: They are stuck in the floor. Error correction is used to bring all three boxes into valid positions.

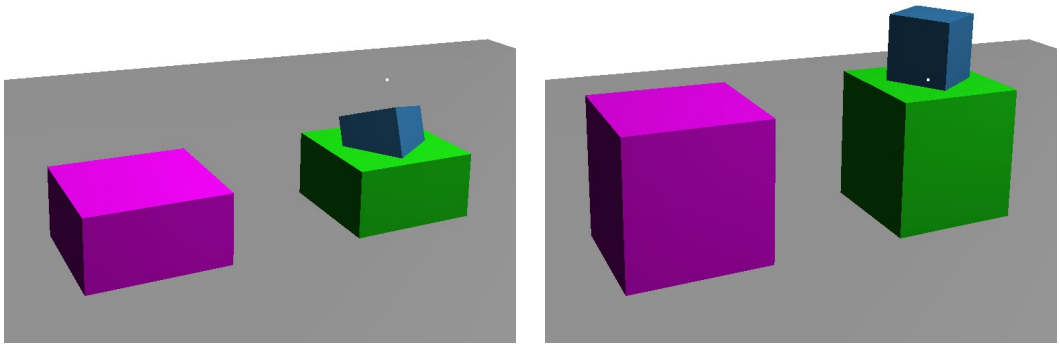

Figure 8.4: Interpenetrating boxes (a) and the corrected positions of the next time step (b).

Baumgarte stabilization can be used, but an inadequate *ERP* value will cause the error correction to remove the error very slowly or to add a lot of energy to the system – the bodies will shoot out of the floor into the air.

Position-based error correction corrects the scene in a single time step. Figure 8.4 (b) shows the result after of the next time step. The positions of the boxes and the orientation of the small box are corrected.

Error correction can also be used to visually assemble articulated rigid bodies. In Figure 8.5 several bodies are lying on the floor. Then new constraints are activated and cause the bodies to assemble as the logo of the Institute for Graphics and Parallel Processing (GUP). Finally the constraints are deactivated again, and the bodies fall to the ground. Since a "fixed joint" has not been implemented yet, three ball joints per body pair are used to glue bodies together and to hang the logo into the air.

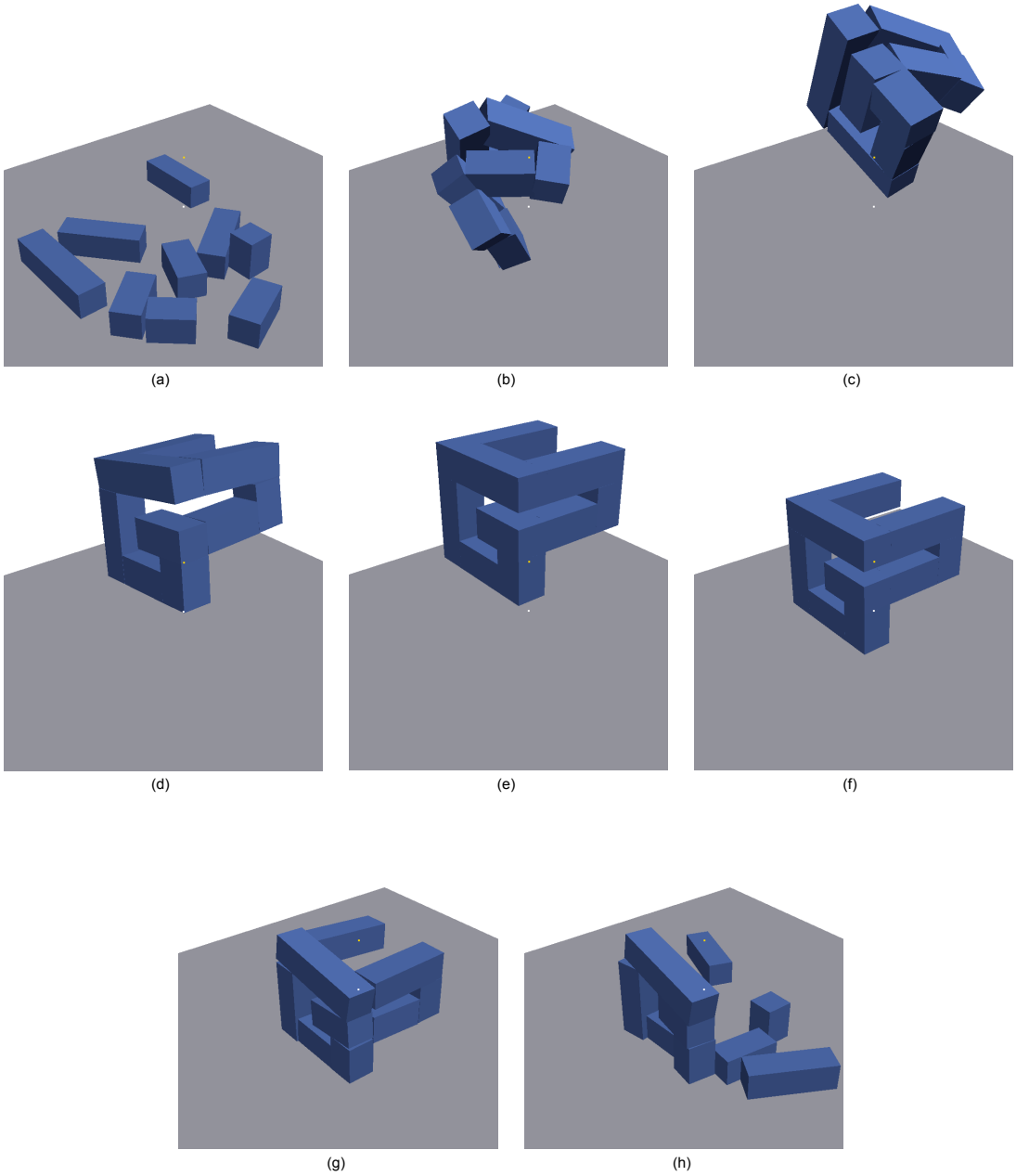

Figure 8.5: Boxes moving into a constraint position (a-e) and falling back to the ground (e-h).

8.5 Example: Ragdoll

A *ragdoll* is an articulated rigid body that resembles the body of a human or an animal. In this example the rigid body simulation is combined with an animation system. The human 3d model and the animation systems were developed by Martin Garstenauer [Gars06]. This model can be animated via skeletal animation, see [Gars06] for details on animation.

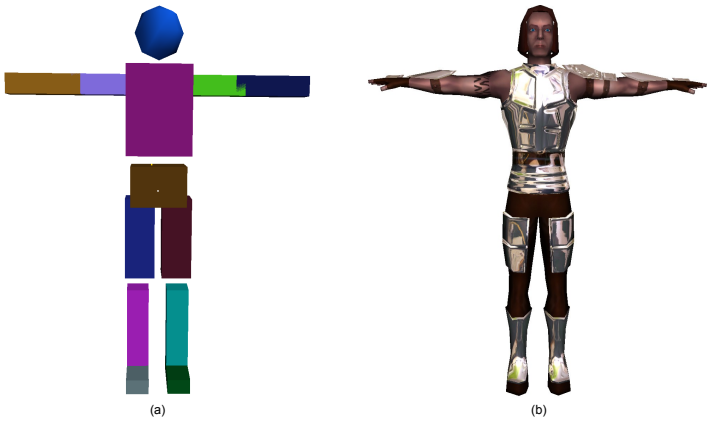

Figure 8.6: Rigid body model and detailed human 3d model.

Several rigid bodies are adjusted to fit the proportions of the human model. Hinge joints connect the rigid bodies at the positions of the human joints. Angular limits restrict the joint motion. Figure 8.6 compares the rigid body model used by the simulation and the human 3d model used by the animation system.

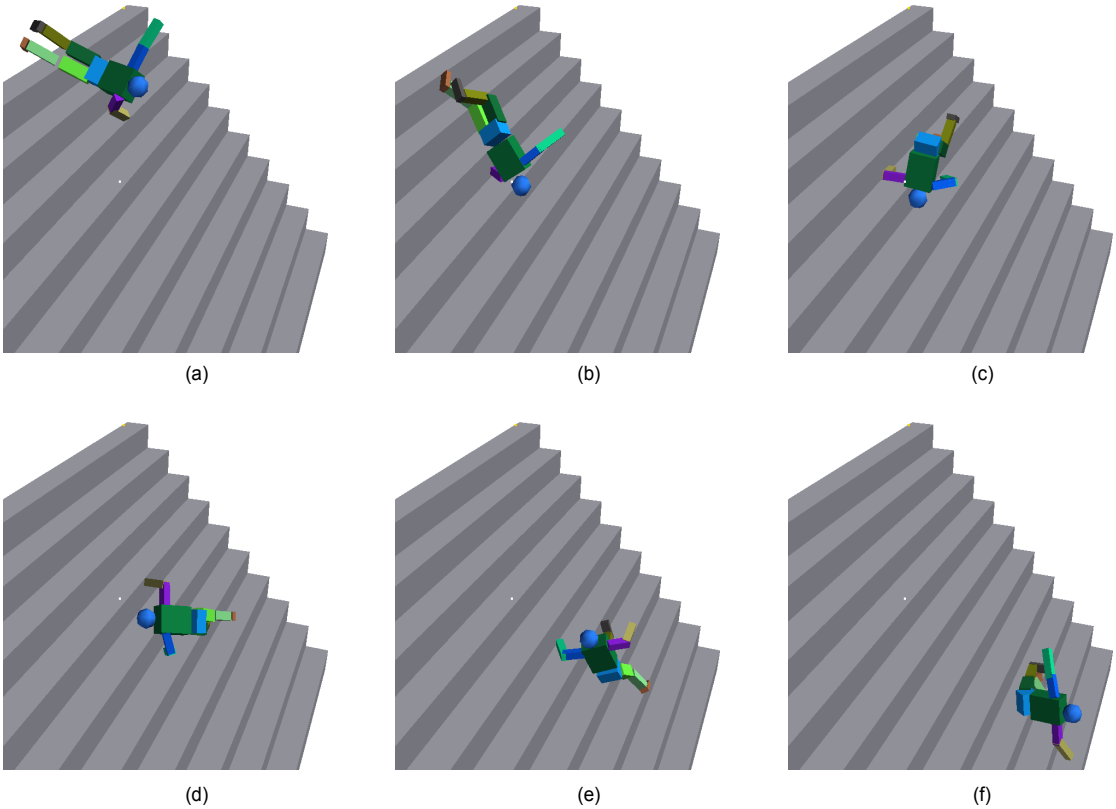

Figure 8.7: A ragdoll tumbling downstairs.

The rigid body model was used in several simulation scenarios. Figure 8.7 shows a scene in which the ragdoll is thrown down a stair. The data of the simulation is used in Figure 8.8 to control the animation of the model. The position of the box at the pelvis controls the position of the human

model. The rotations of the rigid bodies control the skeletal animation of the limbs.

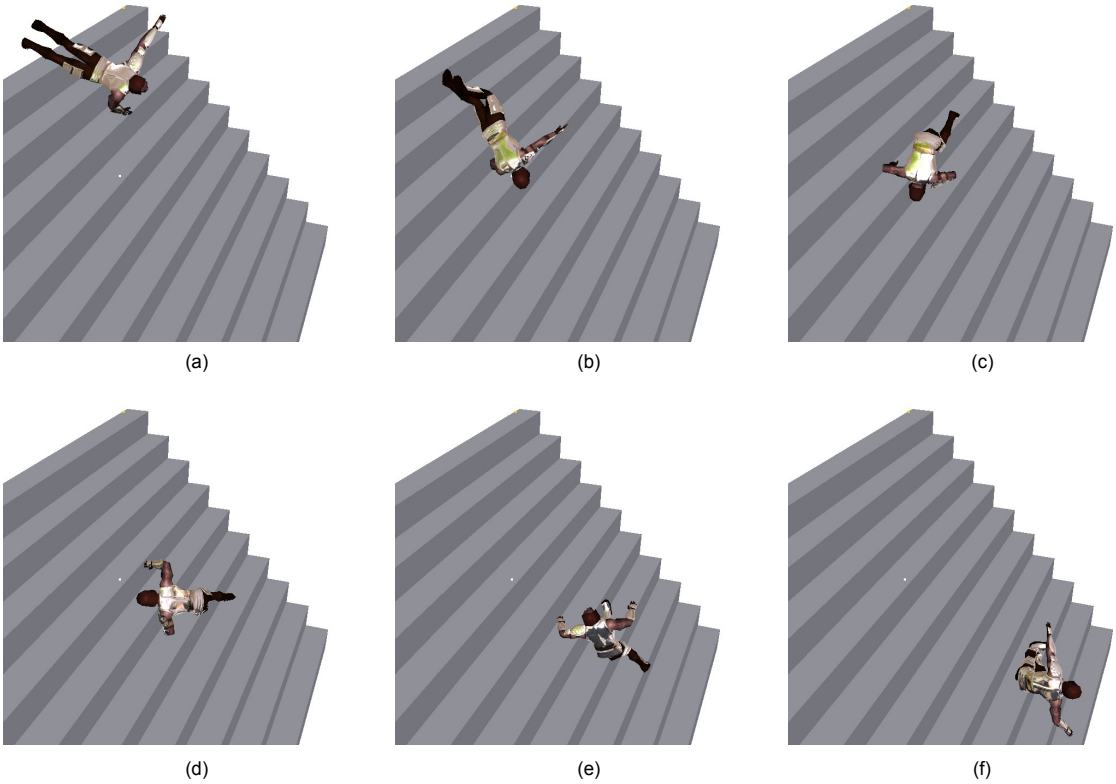

Figure 8.8: The simulation data is applied to a 3d model.

This example demonstrates how rigid body dynamics is normally used in interactive 3d applications: A real model and scene are described by several simplified rigid bodies and constraints. The scene is simulated. In each frame the application reads the positions and rotations of the rigid bodies. This information is used to animate detailed 3d models. The rigid body framework itself does not draw anything – except for debugging purposes.

8.6 Lessons Learned

Dynamics simulation is a difficult task. A variety of problem sources can cause inaccurate, instable, or unrealistic simulation results. These problem sources and selected problems are discussed here.

8.6.1 Problem Sources

- The c*ollision detection* can deliver improper collision information, for example: inappropriate collision normals or inappropriate contact sets.
- *Iterative solvers* return bad approximations if too few iterations are used.
- Numerical integration and limited precision of floating-point numbers are other sources for *numerical inaccuracy* .

- The applied *model of the real world* may simply be wrong, inaccurate, or incapable of describing the real world.

8.6.2 Repeatability

One problem is the lack of *repeatability*. Simulations often use a variable time step. A variable time step leads to different simulation results in each simulation run because the states and collisions are evaluated at different times. Even when a fixed time step is used, the simulation exhibits different behavior if slightly different simulation methods or settings are used.

8.6.3 Visible errors

Some visible errors are very common, for example: Boxes should rest on the floor, but instead they are vibrating. This is especially visible in unstable stacks. Another common problem is that simulation errors cause a box on an inclined plane to slide down, even if it should stop because of static friction.

Strategies to reduce visible errors are:

- Adding a *damping force* (like fluid dynamic drag) that reduces "uncontrolled" movement of bodies due to instabilities.
- *Limiting* velocities, forces, and impulses. Thus, instability does not lead to infinite values.
- *Parameter tuning*: Parameters (coefficients of friction, coefficients of restitution, initial velocities, etc.) have to be adapted carefully to get stable and good results. Mass ratios between different bodies should be less than an order of magnitude, otherwise solutions converge very slowly if a heavy body is on top of a lighter body [Catt05].
- Usage of *contact caching* [Catt05]: Current constraint information (like constraint forces) can be used as additional information in the next time step. In simulation methods that use iterative solvers, cached information can be used to improve the initial guess of the iterative solver. In a scene with slow moving or resting bodies the contact forces will not change much. The result of the iterative solver will be more accurate because the initial guess is near the result. But in scenes with fast moving objects, the initial guess can lead to worse results because contact information has changed a lot. Nevertheless, due to the speed of the bodies, the wrong results will be less visible. Contact caching can be used to get more accurate results or to get comparable results with fewer iterations.
- Usage of *sleeping* to deactivate "jiggling" bodies on the floor.

8.6.4 Bad performance

A very obvious problem is a lack of simulation speed. Tips to improve performance are:

- Use a *good collision detection* system.
- Use *caching* for often used results and a profiler to find bottlenecks.

- Use *contact reduction* and *contact caching*, see Moravánszky and Terdiman [MoTe04], and Catto [Catt05].
- Use *iterative algorithms* which can be aborted any time and still give usable intermediate results.
- Separate the system into *contact groups* (as mentioned in Section 6.3, "Other Methods") and process these contact groups individually
- Explore *hardware and parallelism*. Moravánszky [Mora04] shows how to use programmable graphics hardware to solve an LCP problem.

8.6.5 Believability

To achieve believable results it is important that a simulation is fast, stable, and represents the laws of physics to some extent. Many things can be done to improve believability that are beyond "pure physics simulation". The knowledge of animators and film makers is a good source, for example: Williams [Will01].

Ways to improve the "illusion":

- Add *sound effects*: Sound effects are part of collisions or friction (bodies scratching over a surface). Different sounds can be used to indicate different material.
- Use *shadows*: Shadows need to be used to indicate positions in a 3d scene.
- Use *special effects*: Dust clouds appear on impact, sparks spray if metal scratches metal, the screen can shake if a very heavy body lands on the floor, etc.

For believable simulation it is important to fulfill the expectations of the audience – "realism" is more than the laws of mechanics.

9 Conclusion and Future Work

"Prediction is very difficult, especially about the future."

- Niels Bohr (1885 - 1962)

9.1 Conclusion

Simulation of rigid body dynamics is a method to improve the interactivity and the believability of 3d applications. Several simulation methods exist and qualify for use in 3d applications, from simple physically-based effects to general purpose physics engines.

Chapter 2 introduced the necessary basics of physics and mechanics. A thorough understanding of these basics is inevitable for developing physically-based applications. Chapter 3 presented a modular overview of the simulation process. The process is dissected into several exchangeable modules – this overview was the result of studying state-of-the-art simulation methods. Handling collisions, contacts, and constraints are the major challenges of rigid body dynamics, and Chapter 4 showed possible ways to tackle this problems.

Rigid body dynamics is not a closed chapter. Recent works show that development is still going on. Chapter 6 took a glance at methods, which could not be handled in this thesis in detail.

The result of these studies is a framework that was presented in Chapter 7. The framework is independent of specific simulation methods: All examined simulation paradigms fit well into this design. This makes the framework ideal for the study and comparison of existing methods, and for the development and improvement of new methods. It allows splitting simulation methods into independent, exchangeable parts and to combine different ideas from different works.

Chapter 8 showed this framework in action and gave several important insights that were gained during the development and evaluation of the rigid body dynamics framework. Taking these lessons into account will, hopefully, lead to faster, more robust, and more appealing simulations.

9.2 Future Work

For specific use in interactive 3d application, the framework should be tailored and optimized. The main focus of future work will be the collision detection: The quality of the collision detection results, the speed of execution, as well as the number of usable shapes need to be improved.

In addition, the created simulation system can be extended with various features:

- An editor to edit rigid bodies, material properties, constraints, etc.
- File formats to load and save bodies, materials, scenes, etc.
- Automatic creation of rigid bodies from given 3d models. Automatic computation of mass properties, bounding boxes, etc.
- Callback mechanisms for important events. This will allow developers to react on events. For example, sounds can be played when bodies collide.

Probably, the future of rigid body dynamics will lead towards the exploration of hardware parallelism and dedicated physics hardware[1]. This topic has just started to get exciting...

1 Ageia Inc. [Agei05] announced the release of a dedicated physics processor for standard PCs.

A Notation

This section summarizes the notation used in this thesis.

Whenever vectors are used in this work, they are column vectors; so matrices are multiplied from the left. A standard right-handed coordinate system is assumed.

An n-dimensional vector will be denoted by an arrow symbol above the letter and the elements of the vector by subscript ranging from 1 to n. For example:

$$\vec{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}$$

The elements of a 3-dimensional vector are sometimes written with subscripts x, y, z for clarity:

$$\vec{\omega} = \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix}$$

An $m \times n$ matrix will normally be denoted by a bold letter and the elements of the matrix by subscript, for example:

$$\boldsymbol{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \cdots & A_{mn} \end{pmatrix}$$

Subscripts

This table contains the most common meanings of variable subscripts:

Subscript	***Meaning***
body	body space coordinates

Subscript	*Meaning*
C	constraint
CM	center of mass
ext	external
i	belongs to body *i*
j	belongs to body *j*
k	belongs to constraint *k*
n	component in direction of the normal vector
P	belongs to point *P*
rel	relative
rot	rotational (angular) motion only
t	part in direction of tangent vector
trans	translational (linear) motion only
world	world space coordinates
+	after constraint handling
-	before constraint handling

Symbols

The next table summarizes the most common meanings of variables for reference. If the dimension of a multi-dimensional quantity is not given here, it depends on the context in which it is used. The page column references the page on which this quantity was defined.

Variable	*Dimension*	*Meaning*	*Page*
$\vec{0}, \vec{0}_3$		Zero vector (Dimension is given as subscript or depends on the context.)	
$\mathbf{0}$		Zero matrix (all elements are zero)	
$\mathbf{1}_3$	3×3	Identity matrix	
$\boldsymbol{A}$		LCP coefficient matrix	29
$\vec{a}$	3	Linear acceleration	11
C	1	Constraint function	49
$\vec{C}$	s	Constraint function vector	52
E	1	Energy	14, 23
ERP	1	Error reduction parameter	73

Variable	***Dimension***	***Meaning***	***Page***
e	1	Coefficient of restitution	57
$\vec{F}$	*3*	Force	11
$\vec{f}$	$6n$	Generalized force vector	25
$\vec{g}$	*3*	Local acceleration of gravity	25
$\vec{hi}$	s	LCP high limits	30
$\boldsymbol{I}$	3×3	Mass moment of inertia	18
$\vec{J}$	*3*	Linear impulse	13
$\boldsymbol{J}$	$s \times 6n$	Constraint Jacobian of the system	52
$\boldsymbol{K}$	3×3	Matrix for collision impulse computation	61
$\vec{L}$	*3*	Angular momentum	13
$\vec{lo}$	s	LCP low limits	30
$\boldsymbol{M}$	$6n \times 6n$	Mass matrix	25
m	*1*	Total mass of a particle or rigid body	11
n	*1*	Total number of rigid bodies	
$\vec{n}$	*3*	Normal vector	28
$\vec{p}$	*3*	Linear momentum	13
$\boldsymbol{Q}$	4×3	Quaternion multiplication matrix	21
$\vec{q}$	*4*	Rotation quaternion	9
$\boldsymbol{R}$	3×3	Rotation matrix	8
$\vec{r}$	*3*	Radius vector from center of mass to a point	12, 16
$\boldsymbol{S}$	$7n \times 6n$	Velocity-to-position-derivation matrix	25
s	*1*	Total number of constraint rows	
$\vec{s}$	$7n$	Generalized position vector	25
skew	*3x3*	Skew symmetric matrix (Cross-product matrix)	21
t	*1*	Current simulation time	
$\vec{t}$	*3*	Tangent vector	
$\vec{u}$	$6n$	Generalized velocity vector.	25
$\vec{v}$	*3*	Linear velocity	11
V	*1*	Volume	15
W	*1*	Work	14
$\vec{x}$	*3*	Position of a particle or of the center of mass of a rigid body	11
$\vec{\alpha}$	*3*	Angular acceleration	12

Variable	*Dimension*	*Meaning*	*Page*
$\Delta \vec{L}$	*3*	Angular impulse	14
Δt	*1*	Size of current time step	
$\vec{\lambda}$	*s*	Lagrange multipliers	64
μ	1	Coefficient of friction	27
ρ	1	Density	15
$\vec{\tau}$	3	Torque	13
$\vec{\omega}$	3	Angular velocity	12

B Analogy Translation – Rotation

The linear and angular quantities and important equations are summarized here for reference and to improve understanding.

Linear Quantity	***Equations***	←→	***Angular Quantity***	***Equations***
Position $\vec{x}$	$\dot{\vec{x}}=\vec{v}$		Rotation $\boldsymbol{R}$ or $\vec{q}$	$\dot{\boldsymbol{R}}=skew(\vec{\omega})\,\boldsymbol{R}$ $\dot{\vec{q}}=\frac{1}{2}\vec{\omega}*\vec{q}$
Linear velocity $\vec{v}$	$\dot{\vec{v}}=\vec{a}$		Angular velocity $\vec{\omega}$	$\dot{\vec{\omega}}=\vec{\alpha}$
Linear acceleration $\vec{a}$			Angular acceleration $\vec{\alpha}$	
Force $\vec{F}$	$\vec{F}=m\vec{a}$		Torque $\vec{\tau}$	$\vec{\tau}=\vec{\omega}\times(\boldsymbol{I}\,\vec{\omega})+\boldsymbol{I}\,\vec{\alpha}$
Mass m			Mass moment of inertia $\boldsymbol{I}$	
Linear momentum $\vec{p}$	$\vec{p}=m\vec{v}$		Angular momentum $\vec{L}$	$\vec{L}=\boldsymbol{I}\,\vec{\omega}$
Linear impulse $\vec{J}$	$\vec{J}=\vec{F}\,\Delta t=$ $=m\,\Delta\vec{v}=\Delta\,p$		Angular impulse $\Delta\vec{L}$	$\vec{\tau}\,\Delta t=\boldsymbol{I}\,\Delta\vec{\omega}=\Delta\vec{L}$
Translational kinetic energy E_{trans}	$E_{trans}=\frac{m\lvert\vec{v}\rvert^2}{2}$		Rotational kinetic energy E_{rot}	$E_{rot}=\frac{\vec{\omega}^{\,T}\boldsymbol{I}\,\vec{\omega}}{2}$

Important relations between linear and angular quantities are

Quantity	***Relation***
Velocity	$\vec{v}=\vec{\omega}\times\vec{r}$
Acceleration	$\vec{a}=\vec{\alpha}\times\vec{r}$
Force and torque	$\vec{\tau}=\vec{r}\times\vec{F}$
Momentum	$\vec{L}=\vec{r}\times\vec{p}$
Impulse	$\Delta L=\vec{r}\times\vec{J}$

C Mass Moment of Inertia

The elements of the mass moment of inertia matrix may be calculated for rigid bodies with

$$
\begin{aligned}
I_{xx} &= \int_V \rho(x, y, z)(y_i^2 + z_i^2)\, dV, \\
I_{yy} &= \int_V \rho(x, y, z)(x_i^2 + z_i^2)\, dV, \\
I_{zz} &= \int_V \rho(x, y, z)(x_i^2 + y_i^2)\, dV, \\
I_{xy} &= \int_V \rho(x, y, z)\, x_i\, y_i\, dV, \\
I_{xz} &= \int_V \rho(x, y, z)\, x_i z_i\, dV, \\
I_{yz} &= \int_V \rho(x, y, z)\, y_i z_i\, dV.
\end{aligned}
$$

This section shows how to calculate these values for common shapes: boxes and spheres. The axes of the body space coordinate system are placed such that the bodies are symmetric to these axes. Thus the products of inertia are all zero. These formulas and formulas for other shapes are given in Bourg [Bour02] and Shabana [Shab01].

Box[1]

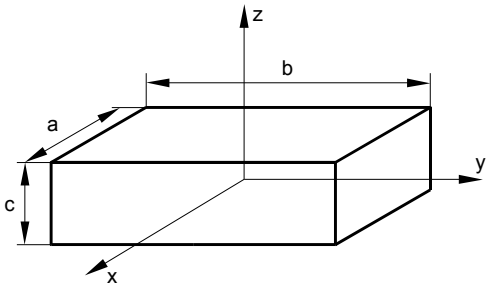

Figure 9.1: Box

$$I_{xx} = \frac{1}{12} m(b^2 + c^2); I_{yy} = \frac{1}{12} m(a^2 + c^2); I_{zz} = \frac{1}{12} m(a^2 + b^2)$$

1 Baraff [Bara97a] shows how to derive these formulas.

Sphere

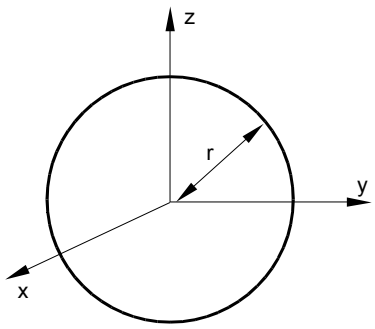

Figure 9.2: Sphere

$$I_{xx} = I_{yy} = I_{zz} = \frac{2}{5} m r^2$$

List of Figures

Figure 2.1: Term overview..... 6
Figure 2.2: Angular motion of a particle..... 12
Figure 2.3: A rigid body with local body coordinate system in world space..... 16
Figure 2.4: A rigid body with linear and angular velocity..... 17
Figure 2.5: A frictional contact between two bodies in 2d..... 27
Figure 2.6: A contact between two bodies in 3d..... 28
Figure 2.7: The friction cone..... 29
Figure 2.8: Illustration of the i'th variable of the MLCP..... 30
Figure 3.1: Modules of a rigid body simulation..... 34
Figure 3.2: Reduced contact set with 5 contacts of a box-box contact..... 39
Figure 3.3: Explicit Time Step Method..... 44
Figure 4.1: The constraint anchor points and local constraint axes..... 52
Figure 4.2: A touching contact (a) and a penetrating contact (b)..... 55
Figure 4.3: A penetrating contact with a penalty spring..... 59
Figure 4.4: 4-sided friction pyramid..... 70
Figure 4.5: Two sets of forces on a box. The same total force is applied in both cases..... 71
Figure 7.1: Class SimulationObject..... 85
Figure 7.2: Class RigidBody..... 86
Figure 7.3: Geo classes..... 87
Figure 7.4: A composite shape..... 88
Figure 7.5: Material classes..... 88
Figure 7.6: ForceEffect classes..... 90
Figure 7.7: Constraint classes..... 91
Figure 7.8: Integrator classes..... 93
Figure 7.9: Collision detection classes..... 94
Figure 7.10: Constraint solver classes..... 95
Figure 7.11: LCP solver classes..... 95
Figure 7.12: Time step method classes..... 96
Figure 7.13: SleepStrategy classes..... 97
Figure 7.14: Validator classes..... 98
Figure 7.15: Class Simulation..... 99
Figure 8.1: Screenshot of viewer application..... 104
Figure 8.2: A big box catapults a small box, which hits the top box of a pyramid stack..... 105

Figure 8.3: Parallel simulation of two stacks. 106
Figure 8.4: Interpenetrating boxes (a) and the corrected positions of the next time step (b)..... 107
Figure 8.5: Boxes moving into a constraint position (a-e) and falling back to the ground (e-h).... 108
Figure 8.6: Rigid body model and detailed human 3d model..... 109
Figure 8.7: A ragdoll tumbling downstairs..... 109
Figure 8.8: The simulation data is applied to a 3d model..... 110
Figure 9.1: Box..... 121
Figure 9.2: Sphere..... 122

Bibliography

[Adam03] Jim Adams: *Advanced Animation with DirectX*, First Edition, Premier Press, 2003.

[Agei05] Ageia Inc.: *Ageia*, URL: http://www.ageia.com, [2005-11-10].

[AnPo02] Mihai Anitescu and Florian A. Potra: A Time-Stepping Method for Stiff Multibody Dynamics with Contact and Friction, *International Journal for Numerical Methods in Engineering*, Vol. 55, No. 7, 2002.

[AnPo97] Mihai Anitescu and Florian A. Potra: Formulating Dynamic Multi-Rigid-Body Contact Problems with Friction as Solvable Linear Complementarity Problems, *Nonlinear Dynamics*, Vol. 14, 1997.

[BaBa88] Ronen Barzel and Alan H. Barr: A Modeling System Based On Dynamic Constraints, *Computer Graphics*, Vol. 22, No. 4, 1988.

[BaRa04] Rick Baltman and Ron Radeztsky Jr: Verlet Integration and Constraints in a Six Degree of Freedom Rigid Body Physics Simulation, *Game Developer Conference 2004 Proceedings*, CMP Media, 2004.

[Bara89] David Baraff: Analytical Methods for Dynamic Simulation of Non-penetrating Rigid Bodies, *Computer Graphics*, Vol. 23, No. 3, 1989.

[Bara90] David Baraff: Curved Surfaces and Coherence for Non-Penetrating Rigid Body Simulation, *Computer Graphics*, Vol. 24, No. 4, 1990.

[Bara91] David Baraff: Coping with Friction for Non-penetrating Rigid Body Simulation, *Computer Graphics*, Vol. 25, No. 4, 1991.

[Bara92] David Baraff: *Dynamic Simulation of Non-Penetrating Rigid Bodies*, Ph.D. Thesis, Cornell University, 1992.

[Bara93] David Baraff: Non-Penetrating Rigid Body Simulation, *Eurographics 1993: State of the Art Reports*, 1993.

[Bara94] David Baraff: Fast Contact Force Computation for Nonpenetrating Rigid Bodies, *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, ACM Press, 1994.

[Bara95] David Baraff: Interactive Simulation of Solid Rigid Bodies, *IEEE Computer Graphics and Applications*, Vol. 15, No. 3, 1995.

[Bara96] David Baraff: Linear-Time Dynamics using Lagrange Multipliers, *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM Press, 1996.

[Bara97a] David Baraff: *An Introduction to Physically Based Modeling: Rigid Body Simulation I - Unconstrained Rigid Body Dynamics*, URL: http://www.cs.cmu.edu/~baraff/pbm/pbm.html, [2005-10-01].

[Bara97b] David Baraff: *An Introduction to Physically Based Modeling: Rigid Body Simulation II: Nonpenetration Constraints*, URL: http://www.cs.cmu.edu/~baraff/pbm/pbm.html, [2005-10-01].

[Bara97c] David Baraff: *Physically Based Modeling: Principles and Practice - Implicit Methods for Differential Equations*, URL: http://www.cs.cmu.edu/~baraff/pbm/pbm.html, [2005-10-01].

[Bart99] Hans-Jochen Bartsch: *Taschenbuch mathematischer Formeln*, 18th Edition, Fachbuchverlag Leipzig, 1999.

[BBMW03] Wolfgang Beer et al.: *Die .NET-Technologie, Grundlagen und Anwendungsprogrammierung*, First Edition, dpunkt.verlag, 2003.

[Bear05] Roy Beardmore: *Coefficients of Friction*, URL: http://www.roymech.co.uk/Useful_Tables/Tribology/co_of_frict.htm, [2005-10-07].

[Berg04] Gino van den Bergen: *Collision Detection in Interactive 3D Environments*, First Edition, Morgan Kaufmann, 2004.

[Bour02] David M. Bourg: *Physics for Game Developers*, First Edition, O'Reilly & Associates, 2002.

[BSMM01] I.N. Bronstein et al.: *Taschenbuch der Mathematik*, Fifth Edition, Verlag Harri Deutsch, 2001.

[Came97] Stephen Cameron: Enhancing GJK: Computing Minimum Penetration Distances between Convex Polyhedra, *Proceedings of the International Conference on Robotics and Automation*, IEEE, 1997.

[Catt05] Erin Catto: *Iterative Dynamics with Temporal Coherence*, URL: http://www.gphysics.com/?page_id=5, [2005-11-02].

[ChRu98] A. Chatterjee and A. Ruina: A New Algebraic Rigid Body Collision Law based on Impulse Space Considerations, *Journal of Applied Mechanics*, Vol. 65, No. 4, 1998.

[Clin02] Michael Bradley Cline: *Rigid Body Simulation with Contact and Constraints*, M.Sc. Thesis, University of British Columbia, 2002.

[ClPa03] Michael B. Cline and Dinesh K. Pai: Post-Stabilization for Rigid Body Simulation with Contact and Constraints, *Proceedings of the International Conference on Robotics and Automation*, IEEE, 2003.

[Cout01] Murilo G. Coutinho: *Dynamic Simulations of Multibody Systems*, First Edition, Springer-Verlag, 2001.

[DXSF05] Microsoft Corporation: *The Sample Framework*, URL: http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/directx9_c_Dec_2004/directx/graphics/programmingguide/sampleframework/sampleframework.asp, [2005-11-22].

[Eber02] David Eberly: *Quaternions Algebra and Calculus*, URL: http://www.geometrictools.com/Documentation/Quaternions.pdf, [2005-10-06].

[Eber04] David H. Eberly: *Game Physics*, First Edition, Morgan Kaufmann, 2004.

[Egan03] Kevin Egan: *Techniques for Real-Time Rigid Body Simulation*, Undergraduate Honors Thesis, Brown University, 2003.

[Enca04] Microsoft Corporation: *Encarta Enzyklopädie Professional*, 2004.

[Erle04] Kenny Erleben: *Stable, Robust, and Versatile Multibody Dynamics Animation*, Ph.D. Thesis, University of Copenhagen, 2004.

[Faur97] Francois Faure: Interactive Solid Animation using Linearized Displacement Constraints, *Proceedings of the 9th Eurographics Workshop on Computer Animation and Simulation*, 1997.

[FDFH05] James Foley et al.: *Computer Graphics: Principles and Practice*, Second Edition, Addison-Wesley, 2005.

[Feat87] R. Featherstone: *Robot Dynamics Algorithms*, Kluwer, 1987.

[FrDi05] Farlex, Inc.: *The Free Dictionary*, URL: www.thefreedictionary.com, [2005-10-04].

[Gars06] Martin Garstenauer: *Character Animation in Real-Time*, Master Thesis, Johannes Kepler University Linz, 2006.

[GHJV95] Erich Gamma et al.: *Design Patterns - Elements of Reusable Object-Oriented Software*, First Edition, Addison-Wesley, 1995.

[GiBO03] T. Giang, G. Bradshaw, and C. O'Sullivan: Complementarity Based Multiple Point Collision Resolution, *Proceedngs of the Fourth Irish Workshop on Computer Graphics*, 2003.

[GiJK88] Elmer G. Gilbert, Daniel W. Johnson, and S. Sathiya Keerthi: A Fast Procedure for Computing the Distance between Complex Objects in Three-Dimensional Space, *IEEE Journal of Robotics and Automation*, Vol. 4, No. 2, 1988.

[Gome02] Miguel Gomez: Coping with Friction in Dynamic Simulations, In *Game Programming Gems 3*, First Edition, Charles River Media Inc., 2002.

[GoPS02] Herbert Goldstein, Charles Poole, and John Safko: *Classical Mechanics*, Third Edition, Pearson Education International, 2002.

[GuBF03] Eran Guendelman, Robert Bridson, and Ronald Fedkiw: Nonconvex Rigid Bodies with Stacking, *ACM Transactions on Graphics*, Vol. 22, No. 3, 2003.

[Hahn88] James K. Hahn: Realistic Animation of Rigid Bodies, *Computer Graphics*, Vol. 22, No. 4, 1988.

[Jako01] Thomas Jakobsen: Advanced Character Physics, *Game Developers Conference 2001 Proceedings*, CMP Media, 2001.

[Jeff85] David R. Jefferson: Virtual Time, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, 1985.

[KaEP05] Danny M. Kaufman, Timothy Edmunds, and Dinesh K. Pai: Fast Frictional Dynamics for Rigid Bodies, *ACM Transactions on Graphics*, Vol. 24, No. 3, 2005.

[KaNB03] Zoran Kačić-Alesić, Marcus Nordenstamm, and David Bullock: A Practical Dynamics System, *Eurographics/SIGGRAPH Symposium on Computer Animation*, 2003.

[Karm02] MathEngine: *Karma, Physical Simulation Software*, 2002.

[KaSK97] Katsuaki Kawachi, Hiromasa Suzuki, and Fumihiko Kimura: Simulation of Rigid Body Motion with Impulsive Friction Force, *Proceedings of IEEE International Symposium on Assembly and Task Planning*, 1997.

[KaSo04] Shinya Karasawa and Kiyoshi Sogabe: Evaluation of Multibody Dynamics Analysis Methods, *Proceedings of The Second Asian Conference on Multibody Dynamics*, 2004.

[Kava03] Ladislav Kavan: Rigid Body Collision Response, *Proceedings of the 7th Central European Seminar on Computer Graphics*, 2003.

[Kokk05] Vangelis Kokkevis: Analytical Constraints for Articulated Body Dynamics, *Lecture notes for the course "Introduction to Articulated Rigid Body Dynamics", ACM SIGGRAPH*, 2005.

[Kuyp03] Friedhelm Kuypers: *Klassische Mechanik*, 6th Edition, Wiley-VCH, 2003.

[Laco03] Claude Lacoursière: Splitting Methods for Dry Frictional Contact Problems in Rigid Multibody Systems: Preliminary Performance Results, *Proceedings of the Annual SIGRAD Conference*, 2003.

[Löts82] P. Lötsdtedt: Mechanical Systems of Rigid Bodies subject to Unilateral Constraints, *SIAM Journal on Applied Mathematics*, Vol. 42, No. 2, 1982.

[MiCa94] Brian Mirtich and John Canny: Impulse-Based Dynamic Simulation, *Proceedings of Workshop on Algorithmic Foundations of Robotics*, 1994.

[MiCa95] Brian Mirtich and John Canny: Impulse-Based Simulation of Rigid Bodies, *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, 1995.

[Mile96] Victor J. Milenkovic: Position-Based Physics: Simulating the Motion of Many Highly Interacting Spheres and Polyhedra, *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM Press, 1996.

[Mill04] Tom Miller: *Managed DirectX 9, Graphics and Game Programming*, First Edition, Sams Publishing, 2004.

[Mirt00] Brian Mirtich: Timewarp Rigid Body Simulation, *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press, 2000.

[Mirt96a] Brian Mirtich: Fast and Accurate Computation of Polyhedral Mass Properties, *Journal of Graphics Tools*, Vol. 1, No. 2, 1996.

[Mirt96b] Brian Mirtich: *Impulse-based Dynamic Simulation of Rigid Body Systems*, Ph.D. Thesis, University of California, 1996.

[Mirt98a] Brian Mirtich : *Rigid Body Contact: Collision Detection to Force Computation*, URL: http://www.merl.com/publications/TR1998-001/, [2005-11-22].

[Mirt98b] Brian Mirtich: V-Clip: Fast and Robust Polyhedral Collision Detection, *ACM Transactions on Graphics*, Vol. 17, No. 3, 1998.

[Mirt98c] Brian Mirtich: Efficient Algorithms for Two-Phase Collision Detection, In *Practical Motion Planning in Robotics: Current Approaches and Future Directions*, First Edition, Wiley-VCH, 1998.

[MiSc01] Victor J. Milenkovic and Harald Schmidl: Optimization-Based Animation, *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, 2001.

[Mora04] Ádám Moravánszky: Dense Matrix Algebra on the GPU, In *ShaderX², Shader Programming Tips & Tricks with DirectX9*, First Edition, Wordware Publishing Inc., 2004.

[MoTe04] Ádám Moravánszky and Pierre Terdiman: Fast Contact Reduction for Dynamics Simulation, In *Game Programming Gems 4*, First Edition, Charles River Media Inc., 2004.

[MoWi88] Matthew Moore and Jane Wilhelms: Collision Detection and Response for Computer Animation, *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, ACM Press, 1988.

[Murt88] Katta G. Murty: *Linear Complementarity, Linear and Nonlinear Programming*, URL: http://ioe.engin.umich.edu/people/fac/books/murty/linear_complementarity_webbook/, [2005-11-22].

[NDoc05] Open Source Project: *NDoc Code Documentation Generator for .NET*, URL: http://ndoc.sourceforge.net, [2005-11-22].

[PlBa88] John C. Platt and Alan H. Barr: Constraint Methods for Flexible Models, *Computer Graphics*, Vol. 22, No. 4, 1988.

[PTVF03] William H. Press et al. : *Numerical Recipes in C++*, Second Edition, Cambridge University Press, 2003.

[Raib90] Eric Raible: Matrix Orthogonalisation, In *Graphics Gems I*, First Edition, Academic

Press, 1990.

[Redo04] S. Redon: Continuous Collision Detection for Rigid and Articulated Bodies, *Lecture notes for the course "Collision Detection and Proximity Queries". ACM SIGGRAPH*, 2004.

[ReKC02] Stephane Redon, Abderrahmane Kheddar, and Sabine Coquillart: Gauss' Least Constraints Principle and Rigid Body Simulation, *Proceedings of the International Conference on Robotics and Automation*, 2002.

[SaSc98] Jörg Sauer and Elmar Schömer: A Constraint-Based Approach to Rigid Body Dynamics for Virtual Reality Applications, *Proceedings of the ACM symposium on Virtual reality software and technology*, ACM Press, 1998.

[Schm02] Harald Schmidl: *Optimization-Based Animation*, Ph. D. Thesis, University of Miami, 2002.

[Shab01] Ahmed A. Shabana: *Computational Dynamics*, Second Edition, John Wiley & Sons Inc., 2001.

[Shoe85] Ken Shoemake: Animating Rotation with Quaternion Curves, *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, ACM Press, 1985.

[Shoe94] Ken Shoemake: *Quaternions*, URL: ftp://ftp.cis.upenn.edu/pub/graphics.shoemake/quatut.ps.Z, [2005-10-03].

[Smit04] Russ Smith: Constraints in Rigid Body Dynamics, In *Game Programming Gems 4*, First Edition, Charles River Media Inc., 2004.

[Smit05] Russel Smith: *Open Dynamics Engine*, URL: www.ode.org, [2005-11-02].

[Stra03] Gilbert Strang: *Lineare Algebra*, First Edition, Springer-Verlag, 2003.

[StTr96] D.E. Stewart and J.C. Trinkle: An Implicit Time-Stepping Scheme for Rigid Body Dynamics with Inelastic Collisions and Coulomb Friction, *International Journal for Numerical Methods in Engineering*, Vol. 39, 1996.

[Subv05] CollabNet: *subversion.tigris.org*, URL: http://subversion.tigris.org, [2005-11-22].

[TPBF87] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer: Elastically Deformable Models, *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, ACM Press, 1987.

[TPSL97] Jeff Trinkle, Jong-Shi Pang, Sandra Sudarsky, and Grace Lo: On Dynamic Multi-Rigid-Body Contact Problems with Coulomb Friction, *Zeitschrift für Angewandte Mathematik und Mechanik*, Vol. 77, No. 4, 1997.

[Verl67] Loup Verlet: Computer Experiments on Classical Fluids, Thermodynamical Properties of Lennard-Jones Molecules, *Physical Review*, Vol. 159, 1967.

[WaWa92] Alan Watt and Mark Watt: *Advanced Animation and Rendering Techniques - Theory and Practice*, First Edition, ACM Press, 1992.

[WiBa97] Andrew Witkin and David Baraff: *Physically Based Modeling: Principles and Practice - Differential Equation Basics*, URL: http://www.cs.cmu.edu/~baraff/pbm/pbm.html, [2005-10-01].

[WiFB87] Andrew Witkin, Kurt Fleischer, and Alan Barr: Energy Constraints On Parameterized Models, *Computer Graphics*, Vol. 21, No. 4, 1987.

[WiGW90] Andrew Witkin, Michael Gleicher, and William Welch: Interactive Dynamics, *Computer Graphics*, Vol. 24, No. 2, 1990.

[Wiki05] *Wikipedia - The Free Encyclopedia*, URL: http://www.wikipedia.org, [2005-11-10].

[Will01] Richard Williams: *The Animator's Survival Kit*, First Edition, Faber and Faber Limited, 2001.

[Witk97a] Andrew Witkin: *Physically Based Modeling: Principles and Practice - Particle System Dynamics*, URL: http://www.cs.cmu.edu/~baraff/pbm/pbm.html, [2005-10-01].

[Witk97b] Andrew Witkin: *Physically Based Modeling: Principles and Practice - Constrained Dynamics*, URL: http://www.cs.cmu.edu/~baraff/pbm/pbm.html, [2005-10-01].

Index

A

ABM..... 81
acceleration.....6, 10pp., 16p., 22p., 25, 28, 35, 41pp., 47, 50, 52, 64pp., 70, 80, 116p., 119
adaptive time step method..... 46
analytical method..... 64
angular impulse..... 13, 23, 42, 60, 118p.
angular momentum.....13pp., 17p., 20, 25, 36, 80, 85, 117, 119
angular velocity.....12, 17, 20p., 23, 25, 36, 42, 96, 118p.
Articulated Body Method..... 81
articulated rigid body..... 41, 108

B

backtracking method..... 46
backward Euler integration..... 37
ball-and-socket joint..... 41, 52pp., 63, 89
Baumgarte stabilization... 73pp., 77, 92, 95, 107
behavior function..... 49
bilateral constraint..... 50
bisection time step method..... 46
body coordinate system.....15, 85
body frame.....15, 103, 110
body space..... 15pp., 19pp., 85, 88pp., 115, 121
box friction..... 70

C

center of mass..... 14pp., 19p., 22p., 25p., 33, 35p., 42, 64p., 75p., 86p., 89p., 116p.
centripetal force..... 17
Chasles' theorem..... 16
classical physics..... 6
coefficient of friction..... 27, 70, 118
coefficient of restitution..... 57, 59, 62, 117
colliding contact..... 39, 45, 56, 59p., 62p., 104
collision....1, 3, 14, 34p., 38pp., 55pp., 64, 68p., 71, 75p., 81, 83p., 86pp., 92, 94, 98p., 101pp., 110pp., 117
collision detection....35, 39pp., 55p., 58, 76, 84, 86pp., 92, 94, 98p., 101pp., 110p., 114
collision law..... 57p., 62
collision solver..... 35, 39, 41, 45
conjugate gradient method..... 31, 66
conservation theorem..... 14
conservative advancement..... 46p.
constrained motion..... 41
constraint..... 1, 3, 29p., 33pp., 40pp., 48pp., 57pp., 62pp., 73pp., 79pp., 83pp., 89pp., 94p., 98p., 101pp., 107, 110p., 113p., 116p., 123pp.
constraint anchor point.....52pp., 59, 63p., 74pp., 91p.
constraint equation..... 49p., 52pp., 57, 62, 64, 66pp., 74, 80, 92
constraint force..42p., 45, 51, 64pp., 74, 80, 85, 92, 111
constraint function.....49pp., 57p., 65, 116
constraint solver.....3, 30, 35, 41pp., 48p., 51, 73, 91p., 94, 101pp.
constraint stabilization.....73p.
contact..... 1, 3, 27p., 31, 33, 35p., 39pp., 51, 55pp., 62pp., 75, 79, 81, 83, 88, 91p., 94p., 98pp., 103p., 110pp., 123, 125p., 128pp.

contact caching..... 111p.
contact graph..... 46, 63
contact group..... 81, 112
contact normal..... 55, 57, 60, 92
contact reduction..... 112
contact region..... 39p.
coriolis term..... 22, 24p.
Coulomb's law of friction..... 27
cross-product matrix..... 21, 117

D

damping constant..... 26
Dantzig's algorithm..... 31, 67
deformable body..... 7, 33p.
degree of freedom..... 7p., 15, 50, 52p., 80
density..... 15, 19, 88p., 118
dependent limit..... 31, 70, 92
deviation function..... 49
drag..... 22, 26, 35p., 64, 89p., 99, 111
dynamic body..... 86
dynamic friction..... 27, 61, 67, 70
dynamic simulation SDK..... 6
dynamics..... 1pp., 5pp., 10p., 29, 31, 33, 38, 50, 64, 71, 80p., 83, 110, 113p.
dynamics engine..... 6, 83

E

eager strategy..... 43
elastic collision..... 14, 56p., 59, 62
energy..... 14
equality constraint..... 50p., 54, 64pp., 73
equations of motion..... 11, 22pp., 36, 50, 65, 80, 85
ERP..... 73p., 91p., 107, 116
error correction. 3, 35, 42pp., 47, 59, 73pp., 83, 94, 101, 107
error reduction parameter..... 73, 91, 116
Euler angles..... 7p.
Euler integration..... 37p., 45, 68, 77
Euler's method..... 37
Euler's theorem..... 7
explicit Euler integration..... 37p., 45, 68, 77
explicit time step method..... 43, 45, 47

F

fix-point-iteration method..... 47
fixed time step method..... 43, 46pp., 63, 106
fluid dynamic drag..... 26, 89p., 111
force..... 1, 5pp., 10p., 13pp., 22pp., 31, 33pp., 41pp., 47, 50p., 53p., 57pp., 62, 64pp., 73pp., 80, 83pp., 89p., 92, 94, 97pp., 102, 111, 117, 119
force computation..... 35p., 38p., 42, 44, 48
forward Euler integration..... 37
fractioned time step..... 47
freezing..... 47
friction..... 1, 22, 26pp., 31, 35p., 57p., 60pp., 67pp., 81, 88, 92, 103, 105, 111p., 118
friction cone..... 28, 61, 69p.

G

Galilean system..... 16
Gauss-Seidel method..... 31
general constraint..... 41, 53, 62, 64, 68, 71, 75, 79, 83, 94p., 98p., 101, 104
generalized constraint force vector..... 64
generalized coordinates..... 80
generalized force vector..... 24, 117
generalized mass matrix..... 25
generalized position vector..... 24, 36, 117
generalized velocity vector..... 24, 36, 117
Gilbert-Johnson-Keerthi (GJK) algorithm..... 40
gimbal lock..... 8
GJK..... 40
Gram-Schmidt orthonormalization..... 8
gravity..... 1, 22, 25, 35, 62, 64, 70, 75, 85, 89p., 97, 99, 117

H

hinge joint..... 41, 53p., 63, 80, 98, 104p., 109
holonomic constraint..... 50, 52p., 80

I

implicit Euler integration..... 37p., 68
implicit time step method..... 45, 47
impulse..... 13p., 22p., 29, 31, 35, 42p., 51, 56, 59pp., 67pp., 73pp., 83, 92, 95, 101, 106p., 111, 117pp.
impulse-based simulation..... 62, 83, 107
inequality constraint..... 50p., 54, 64, 66p.

inertia matrix..... 20, 76, 88p., 121
inertia tensor..... 19p., 22, 89
inertial frame..... 16
inertial system..... 16
inertial torque..... 22

J

Jacobi method..... 31
Jacobian..... 52pp., 66, 80, 92, 117
joint.....1, 35, 40p., 48p., 52pp., 63, 73, 80p., 89, 92, 98, 104pp., 109

K

kinematics..... 6
kinetic energy..... 14, 23, 57, 96p., 119
kinetic friction..... 27p.
kinetics..... 6

L

Lagrange multiplier method..... 65
Lagrange multipliers..... 64pp., 69, 118
Lagrange's equation of motion..... 80
lazy strategy..... 43
LCP..... 29pp., 66pp., 75p., 81, 92, 95, 103, 106, 112, 116p.
leap-frog method..... 38
Lemke's algorithm..... 31
limit.....31, 35, 41, 51, 54, 56, 63pp., 69p., 80p., 86, 92, 97pp., 103, 109p., 117
linear complementarity problem... 2, 5, 29p., 64
linear impulse..... 13, 23, 42, 60, 117, 119
linear momentum..... 13pp., 36, 85, 117, 119

M

mass.....10p., 13pp., 22pp., 33pp., 42, 46, 52, 54, 59, 64pp., 75p., 83pp., 97, 111, 114, 116p., 119, 121
mass moment of inertia..14, 17pp., 33, 89, 117, 119, 121
mass point..... 11, 89
maximal coordinates..... 80
mechanics..... 1p., 5pp., 13, 64, 112p.
midpoint method..... 38, 93
mixed linear complementarity problem..... 30
MLCP..... 30p., 67pp., 75p., 92, 95
moderate strategy..... 43
modern physics..... 6
moment of force..... 13
moment of inertia coefficients..... 18p.

N

net force..... 11
Newton-Euler equations..... 22
Newton's impact law..... 57
numerical drift..... 8
numerical integration..... 35, 37p., 43pp., 48, 85, 93, 101, 110

O

ODE..... 36, 38, 73
ordinary differential equation..... 36
orthogonal matrix..... 8
overlapping contact..... 40

P

parallel axis theorem..... 19
particle..... 5, 11pp., 17pp., 26, 79p., 117
penalty method..... 58p., 71, 73, 91, 94, 101
penalty spring..... 59, 91, 94
penetrating contact..... 40, 46, 55
penetration depth..... 40, 47, 63, 95
physically based modeling..... 6
physics..... 1p., 5p., 10, 33, 84, 112pp.
physics engine..... 6, 113
plastic collision..... 14, 56p.
position update.... 35p., 38p., 41pp., 65, 75, 101
post-stabilization..... 75
principal axis transformation..... 20
principle of virtual work..... 64
products of inertia..... 18, 20, 121

Q

QP..... 30
quadratic programming problem..... 30
quaternion..... 7, 9p., 20, 25
quaternion multiplication..... 10, 21, 117

R

radius vector..... 12, 117
ragdoll..... 41, 59, 108p.
reduced contact set..... 39
reduced coordinates..... 80
resting contact.... 3, 35, 39pp., 44pp., 49, 56pp., 62, 66, 68, 71, 75, 79, 81, 83, 88, 92, 94p., 101, 104
resting contact solver..... 35, 39, 41, 44p.
retroactive detection..... 46
rigid body..... 1pp., 5pp., 10, 13, 15pp., 29, 31, 33pp., 38pp., 46, 49, 52p., 58, 63, 73, 79pp., 83pp., 94, 96, 99, 101pp., 107pp., 113p., 117, 121
rigid body dynamics... 2p., 5, 7, 29, 31, 38, 80p., 83, 110, 113p.
rigid body physics..... 6, 10
rigid body simulation..... 5, 9, 14, 23, 25, 33p., 37pp., 56, 79, 81, 83, 86, 103, 108
robustness..... 2
rotation matrix..... 7pp., 20
Runge-Kutta method..... 38, 42, 93

S

semi-implicit Euler integration..... 37p.
semi-implicit time step method..... 45, 47
separating contact..... 39, 45, 56, 62p.
shock propagation..... 46, 63
similarity transformation..... 20
skew symmetric matrix..... 21, 117
sleeping 35, 47p., 84, 89, 96p., 100pp., 107, 111
soft body..... 7
SOR..... 31
spatial variables..... 17, 76
spring constant..... 26, 38, 59
stability..... 2, 38, 99, 111
static body..... 86, 102
static equilibrium..... 6
static friction..... 27p., 61p., 70, 111
statics..... 6
stiction..... 27
stiff system..... 38, 59, 68
successive over relaxation..... 31
symplectic Euler..... 37

T

tensor..... 19
time step..... 20, 35, 37p., 43pp., 50p., 55, 58, 61p., 68, 73p., 76, 79, 92, 94, 96pp., 100pp., 106p., 111, 118
time step method..... 35, 43, 45pp., 62, 96, 102, 106
time-stepping method..... 68
torque..... 13p., 22pp., 26, 35p., 41, 50, 64p., 85p., 90, 118p.
total force..... 10p., 14, 22, 35, 85
touching contact..... 55
transfer of axis theorem..... 19

U

unconstrained motion..... 41
unilateral constraint..... 50

V

V-Clip..... 40
validator..... 98, 100, 102
velocity....6, 10pp., 16p., 20p., 23pp., 28, 35pp., 41pp., 50p., 53pp., 62pp., 68pp., 74pp., 79p., 85p., 95pp., 101, 117pp.
velocity update..... 35, 38p., 42, 45, 62
velocity-Verlet method..... 38
Verlet integration..... 38, 79
viscous drag..... 26, 36
Voronoi Clip (V-Clip) algorithm..... 40

W

work..... 14
world coordinate system..... 16, 85
world frame..... 16
world space..... 15pp., 19p., 85, 116

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Magisterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Ternberg, am 19. März 2006 Helmut Garstenauer

Lebenslauf

Persönliche Angaben

Name:	Helmut Garstenauer
Geburtsdatum:	30. November 1979
Geburtsort:	Kirchdorf, Österreich
Eltern:	Stefanie und Konrad Garstenauer
Adresse:	Weingartenstraße 35, 4452 Ternberg, Austria
E-Mail:	helmut.garstenauer@gmx.net
Familienstand:	Ledig

Ausbildung

Sept. 1986 – Juli 1990	Volksschule Ternberg, Oberösterreich
Sept. 1990 – Juli 1994	Hauptschule Ternberg, Oberösterreich
Sept. 1994 – Juli 1999	Höhere Technische Bundeslehranstalt Steyr (Höhere Abteilung für Elektronik – Ausbildungszweig Technische Informatik)
Juni 1999	Reife- und Diplomprüfung mit ausgezeichnetem Erfolg bestanden
März 2000 – Jetzt	Studium der Informatik an der Johannes Kepler Universität Linz Erster Studienabschnitt Informatik mit ausgezeichnetem Erfolg abgeschlossen

Grundwehrdienst

Juli 1999 – März 2000	Jägerschule Klagenfurt und Heeresunteroffiziersakademie Enns

Arbeit

Nov. 1999 – Mai 2005	Freier Mitarbeiter bei TEC-IT Datenverarbeitung GmbH
Jän. 2006 – Jetzt	Software-Entwickler bei TEC-IT Datenverarbeitung GmbH