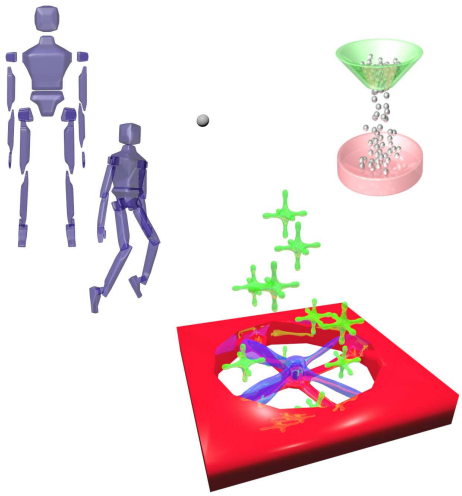
Stable, Robust, and Versatile Multibody Dynamics Animation



This is a Ph.D. dissertation in Computer Science at The Department of Computer Science, University of Copenhagen, Denmark. Supervisor has been Knud Henriksen.

Kenny Erleben

November 2004
(Revised version, April 2005)

This dissertation was set in LaTeX by the author.

Contents

List of Figures

List of Tables

Abstract

This dissertation is concerned with the theory and difficulties of stable, robust, and versatile multibody dynamics simulation for the purpose of computer animation. The subjects covered touch many aspects in simulation and animation including convergence rate, numerical errors, time-stepping, error-correction et cetera.

From an algorithmic viewpoint, the most novel contributions are considered to be a velocity-based shock-propagation time-stepping scheme, and a method for computing general scripted motion. A formal presentation of contact graphs and their usage is given. Also discussed is insight into speed-up methods, object oriented design of constraints, and optimized and easy implementation of iterative linear complementarity problem (LCP) solvers. Contributions to the subject of collision detection include improvements of using signed distance maps for collision detection and contact point generation for box-box primitives.

The focus of the dissertation is technical and theoretical. Great effort has been made to show clearly how one should approach an implementation of the presented material. An implementation of all the presented algorithms and data structures has been made publicly available in [113].

Preface

To put this dissertation in a proper perspective, I feel it is important to give a little insight into my background history and motivation for working in the field of physics-based animation.

As a first year undergraduate student in physics, with a fascination of classical mechanics, the course of my studies was layed down, inspired by computer games such as Scorched Earth (Shareware Computer Game by Wendell Hicken, 1991) just to mention one. Here, simple Newtonian physics was used to compute trajectories of bombs being fired by small tanks placed in a 2D landscape. The idea itself of using a computer combined with the laws of physics to predict events in a completely dynamic way were so interesting and appealing that today I am still captivated with this fascination. It did not take long before my study interests moved from physics towards computer science. In particular, computer graphics and computer animation became the major topics forming a common theme in my (under-) graduate studies.

Being the sole person at the time in the computer science department having this weird craving for doing computer physics, it took considerable time to build up the necessary skills and knowledge for performing simulations more impressive than computing the trajectory of a thrown ball. A lot of computer science disciplines were needed to be learned: Numerical methods, computational geometry, and operational research, as well as trained skills in mathematics and physics.

In May 2001, a study of rigid body simulation methods ended up as my Master Thesis. Written as an introductory textbook in simple rigid body simulation, the main contribution of this work was a unifying modular concept for building rigid body simulators. Simultaneously while writing my Master Thesis, I was working in the industry (3DFacto ApS) developing simulation tools similar to virtual prototyping for a configuration software application. Thus, when starting on my Ph.D. study in November 2001, a solid theoretical foundation in physics-based simulation was already formed together with real-life practical experience. With this background in mind, it is no surprise that during my Ph.D. study the topics have spaned widely. A small list follows below with topics and references for my publications.

- Approximating Heterogeneous Bounding Volume Hierarchies [46].
- An Improved Modular Design for Rigid Body Simulation [56].
- Scripted Bodies using Kinematical Splines [54].
- Contact Graphs in Multibody Dynamics Simulation [48, 49].
- Multi-Scale Singularity Bounding Volume Hierarchy [133, 134].

- Thin Shell Tetrahedral Meshes [50, 51].
- Balance Strategies [116].
- Collision Detection of Deformable Volumetric Objects [55].
- A Simple Plane Patcher Algorithm [47].

To put this dissertation in perspective, it should be mentioned that the work on multi-scale singularity bounding volume hierarchies and balance strategies are based on earlier research co-published with Kerawit Somchaipeng and Camilla Pedersen.

Being a Ph.D. student at DIKU, there has been a multitude of collaborations and initiatives going on in parallel with my studies. During the first year of my Ph.D., the Dynamics Network was formed as a contact network for people in Denmark doing physics-based animation. Later, collaboration with 3DLab, Anybody Technology, and IO-Interactive was started, and this collaboration has resulted in multiple discussions on simulation methods, physics modeling, et cetera. Especially the first two have motivated my increasing interest in simulating deformable objects, such as the soft tissue of humans. The Graphics Group at IMM, DTU has always been supportive and helpful. Lately, they have contributed to the OpenTissue project, an Open Source project started at DIKU. Most recently, DIKU has co-chaired the 45th Conference on Simulation and Modeling[1], where I was a member of the national organizing committee. During the last three months of my Ph.D. study I visited David Stewart at the Department of Mathematics, University of Iowa.

Besides my Ph.D. dissertation the two other major results of my labor are:

The OpenTissue Project: An Open-source Library for Physics-based Animation [113]. A low level application programming interface (API) and a playground for future technologies in middle-ware physics for games and surgical simulation.

Physics-based Animation Textbook: An advanced textbook for graduate students in computer science. This book is the first of its kind, and we, the authors: K. Erleben, J. sporring, K. Henriksen, and H. Dohlmann, feel it fills a gap in the graphics community for a textbook specific for this topic.

Although my study has taken me on quite a round trip topic-wise, my research has focused on three main problems in physics-based animation:

Collision Detection:

- Better methods for building spatial data structures for collision detection of rigid bodies.
- Reducing the computational disadvantages of working with deformable objects, especially self-collision.

[1]The conference is organized in cooperation between Technical University of Denmark, Copenhagen University and Aalborg University and the two organizations: Scandinavian Simulation Society and Danish Simulation Society.

Control:

- Better methods for controlling an animation, like making a body move along a spline in a kinematically correct way, or making an articulated figure keep its balance.

Efficiency, Performance, and Robustness:

- Better data structures for caching and exploiting previously computed results.
- Evaluation of various speed-up ideas. It is interesting to see how performance and simulation quality behaves when a short-cut is used for getting better running time.
- Pitfalls in time-stepping schemes, error-correction, numerical methods for solving LCPs, object design for multibody constraints, new simulator paradigms for obtaining better performance, and robustness towards faulty input and simulation failures.

Traditionally, these three problems have been claimed to be the major bottlenecks by just about every researcher in the field, as can be seen by the ACM SIGGRAPH proceedings in the period from 1987 and up to today 2004. Lately, the field has put much attention towards solving these problems.

Due to time and space limitations I have chosen to limit myself to only present results of my work with multibody dynamics during the course of my Ph.D. study. Thus no work prior to my Ph.D. is published in this dissertation. Some of the presented results have been published over the duration of my Ph.D. study. Here follows a small list of topics in this dissertation:

- Kinematical spline driven scripted bodies.
- A new concept-based module design for building rigid body simulators.
- A introduction to velocity based complementarity formulations of multibody dynamics.
- An object oriented design for constraints in multibody dynamics.
- A formal presentation of usage of contact graphs in multibody dynamics.
- Using first order world simulation to handle simulation errors by projection.
- A presentation of the theory behind an effective iterative LCP solver for Multibody dynamics simulation.
- Combining velocity based complementarity formulation with shock-propagation.

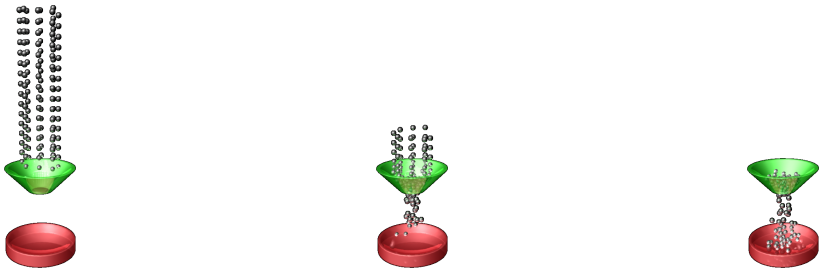


Figure 1: 128 balls falling through a funnel using a velocity based complementarity formulation using constraint stabilization. Complementarity problems were solved with Path [115]. Rendering was done using Maya®.

Figure 2: A robot assembles itself during simulation using constraint stabilization. Simulation was done using a velocity based complementarity formulation. Complementarity problems were solved with Path [115]. Rendering was done using Maya®.

As far as I know, these topics are novel and unique contributions in the field of physics-based animation.

I consider this dissertation to be an advanced text in physics-based animation of multibody dynamics. Therefore, I assume that the reader is familiar with the field, and is familiar with physics and mathematics at a graduate level. To give the reader an idea of the required level of background knowledge, the following works should be considered basic knowledge by the reader: [19, 16, 100, 140]. A textbook or two on classical mechanics will also not hurt, for instance [85, 65]. If the reader has no idea of what these references are then there is little chance of fully understanding every part of this dissertation. I hope that with this dissertation the reader will gain skills and knowledge for producing high-quality and complex animations of multibody dynamics. If the reader has tried to implement the methods in the cited background work, it will be no surprise that these methods are difficult to implement in a general setting. Sometimes one is even left wondering if the methods work at all. This dissertation fills the gap. If the reader applies the theory presented, a simulator producing the images in Figures 1-3 can be implemented within a reasonable amount of time.

Thanks to family and friends for support and love. Thanks to colleagues, collaborators, and students for creating a motivating and inspiring working

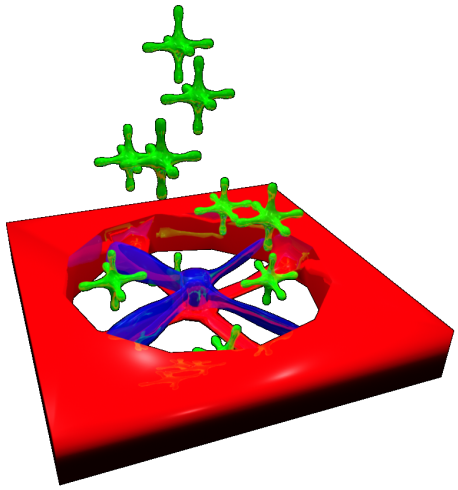


Figure 3: 10 Jacks thrown into a motorized ventilator. Simulation was done using a velocity based complementarity formulation. Complementarity problems were solved with Path [115]. Rendering was done using Maya®.

environment. Thanks to Stig Skelboe, Matthias Teschner and Robert Bridson for serving on my defense commitee. Thanks to Dan Erik Petersen for proofreading.

April 2005
Kenny Erleben

Chapter 1

Introduction to Physics-based Animation

For a long period of time people in computer graphics have been trying to increase realism and believability in computer generated animations and pictures. Some of the original work includes [10, 144, 106, 71, 23, 118, 11, 22].

The general belief is that as we get better and better at rendering images, the lack of physical realism and believability will be more obvious and therefore more annoying to the common observer. The main argument in achieving the goal of more realism has been to use physics to model the behavior and movement of computer models. Today, these efforts have accumulated in what is usually referred to as *physics-based animation* or modeling.

The field was first named in a course in the 1987 ACM SIGGRAPH (the Association for Computing Machinery's Special Interest Group on Computer Graphics) conference, "Topics in Physically-Based Modeling" organized by Alan H. Barr.

Physics-based animation is a highly interdisciplinary field, based on theory from engineering [156, 140, 117, 44], physics [18], and mathematics. Most noticeable are simulation models based on traditional engineering methods used in robotics [39, 59] and solid mechanics [73, 143]. The use of forward dynamics in this field is a way of life, and a popular one too, but not all feel that this is the answer to all our problems.

In a movie production pipeline, it is generally believed that using physics inhibits the creativity and esthetics of an animator. The reason for this is quite obvious. It is hard to be true towards a physical model, while at the same time using every aspect of the "Principles of Animation" [91, 63]. The simple fact that animators do not respect the laws of physics and have no shame in putting their characters in unnatural poses underlines the desire for not being 100% physically correct.

In recent years a new field, maned "plausible simulation" [25], has manifested itself. Techniques, like sampling the entire range of possible simulations using forward dynamics [78], and optimization of physical constraints [153, 120, 119, 57, 95] have been proposed. Animating physically realistic human characters is especially challenging. One might think that this is much simpler than chaotic simulations of natural phenomena like water [62] and smoke [60]. However, human observers are fine tuned to recognize human cues from motion patterns such as e.g. emotions and gender. Any new effort toward improving the physical realism of animating human characters is therefore valuable both to

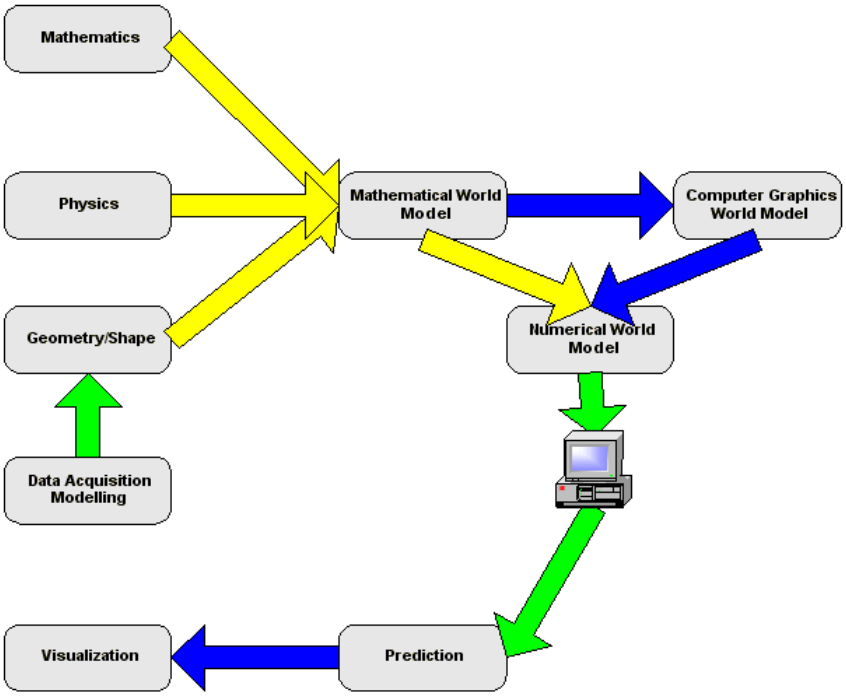


Figure 1.1: Schematic overview of physics-based animation and simulation. Yellow arrows are known from traditional engineering, blue arrows depict typical extra steps taken in computer graphics, green arrows show steps common to both engineering and computer graphics.

the application and the research.

This dissertation is not concerned with "Principles of Animation", nor does it attempt to qualify or quantify believability or plausibility of the presented methods and algorithms. It is not concerned with the rendering process, nor animation systems. Instead, this dissertation is firstly devoted to give the reader a firm and solid foundation for understanding how the mathematical models are derived from physical and mathematical principles, and secondly, how the mathematical models are solved in a efficient, robust, and stable manner on a computer.

Physics-based animation is a highly inter-disciplinary field, making it difficult to draw sharp lines/borders between it and other fields such as engineering and scientific computing. In the following, a rough outline of the differences in these fields are sketched.

One way to look at physics-based animation is that we take a lot of theory from physics and mathematics, add some geometry to it, and mix it all together to obtain a mathematical model of our real world. Once this model is obtained, it can be remodeled into a numerical model which can be implemented on a computer. Predictions about the real world can now be computed using the computer. These predictions can say something about where we expect things to move (forward kinematics and dynamics), or it can tell us something about how we should affect an object in order to obtain some desired movement (inverse kinematics and dynamics). This view of the world is schematized in Figure 1.1. This view of the world is often taken in engineering disciplines. The goal is often to say something about what we can expect to happen in the real world over a long duration of time and with high accuracy.

There is a wide range of techniques used to obtain the numerical model. These techniques have been developed with great care such that the numerical results converge towards the "results" of the mathematical model in the limit of very small time-steps and small spatial discretizations.

However, it is often the case that neither the mathematical model nor the numerical techniques known from engineering are useful for computer animation. Computer animation favors high visual detail, high performance, and low memory consumption. The engineering approach is often not interested in momentary visual details, since their problems often involve answering what happens over 100 years. In order to accurately predict these events, computation time or memory consumption is often high compared to those used in computer graphics. If an engineering application should compute whether a bridge would hold for 100 years, it would be justified to have tens or hundreds of computers produce the result over days or weeks. In contrast, in computer animation we would rather be able to visualize the crashing bridge on our laptop in real-time.

Due to these differences between engineering and computer graphics, computer scientists remodel the mathematical models to favor visual details and choose numerical techniques favoring speed, robustness, and stability over accuracy and convergence. These tradeoffs result in a "computer graphics model". Of course, a lot of the work and models in computer graphics are based on models and methods used in engineering.

The remodeling and extra steps typically done by computer graphics scientists are depicted in Figure 1.1 with blue arrows. The arrows are not unique to computer graphics. There are examples in scientific computing where visualization is just as important, but in many cases there are differences. For instance, in some cases in engineering flow fields may be visualized by arrows or particle traces instead of real smoke, which is what we want to see in computer graphics.

1.1 Scientific Computing Versus Computer Graphics in Practice

The engineering methods are dominated by the field of *scientific computing*. Looking at some of the applications, it becomes clear that the diversity is large.

The numerical wind tunnel at RISOE `www.risoe.dk/vea-aed/numwind/` uses a hyperbolic grid generator for both two-dimensional and three-dimensional domains. Navier-Stokes fluid-flow equations are solved using EllipSys2D/3D code which is an incompressible finite volume code. The computation of a stationary wind mill rotor takes roughly 50 CPU hours. Using 14 CPUs (3.2GHz Pentium M, 2GB RAM) in a cluster, makes the simulation take about 4 hours. Non-stationary computations take three to four times longer [80].

In atomic-scale materials, physical simulations of plastic deformation in nano crystalline copper are done on a cluster (Niflheim, 2.1 Teraflops) of everyday PCs. A simulation involves 100 nodes running in parallel and often takes weeks to complete (`www.dcsc.dk`).

Fluid mechanics involves a wide range of flow problems. Three-dimensional non-stationary flows typically requires $10 - 100 \times 10^6$ grid nodes and uses up to 1000 CPU hours per simulation (`www.dcsc.dk`).

Solid mechanics are also simulated (`www.dcsc.sdu.dk/`) but often not visualized [149].

In quantum chemistry, problems often involve 10^9 variables and simulations can take up to 10 days or more to complete [128].

Computational astrophysics (`http://www.nbi.ku.dk/side22730.htm`) involves simulations of galaxy, star and planet formations. Smoothed particle hydrodynamics and adaptive mesh refinement are some of the methods used. Computations often take weeks or months to complete [110].

Weather reports are simulated continuously and saved to disk regularly at DMI (`www.dmi.dk`) and information is saved every 3 hours. 48-hour predictions are computed on 2.5 grid nodes using a time-step size of two minutes. The total number of computations are on the order of 10^{12} and solved on very large super computers [127].

Real-time is often not a very useful term in scientific computing. For instance, in chemistry, simulated time is on the order of pico seconds but the computation takes days to complete. The main idea of the simulations is often to see a chemical process in slow motion in order to observe what is happening. This is something that is not doable in a laboratory. In contrast, astrophysics and sea simulations are on slower time scales and are therefore simulated at higher rates. In sea flow simulations, a couple of hundred years are simulated per day. In conclusion, turn-around time is often of the order of 24 hours to 30 days [149].

From this short survey of scientific computing it seems that large super computers and clusters are often used. The amount of data is astronomical and computation times cover a wide range from minutes to hours, from hours to days, and weeks and months are not unheard of. Visualization ranges from simple arrow illustrations of flow fields to quite advanced scientific visualization.

Looking at physics-based animation in the graphics literature, a slightly different picture is shown than the one seen in scientific computing.

Smoke simulations for large scale phenomena [121] using half a million particles takes 2-10 secs. of simulation time per frame, while the rendering time takes 5-10 minutes per frame (2.2 GHz Pentium 4). In [95], key frame control of smoke simulations takes between 2-24 hours to perform on a 2GHz Pentium 4. Target driven smoke simulations [58] on a 2.4 GHz Pentium 4 in 2D on a 256^2 grid takes 50 secs.. It takes 35 minutes for a 1 second animation in 3D on a 128^3 grid.

In [61] suspended particle explosions are animated. The simulation time ranges from 4-6 secs. per frame on a 3GHz Pentium 4. With render times included, a simulated second is on the order of minutes. Animation of viscoelastic fluids ([64]) using a 40^3 grid runs at half an hour per second of animation on a 3GHz Pentium 4. Animation of two way interaction between rigid objects and fluid [34] using a $64 \times 68 \times 84$ grid on a 3GHz Pentium 4 with 1 GB RAM takes 401 simulation steps for one second of animation with average CPU time per simulation step of 27.5 secs.

Robust cloth simulation without penetrations [28] of a 150×150 node piece of cloth runs at 2 minutes per frame on a 1.2 GHz Pentium 3. Untangling of cloth [20] for a cloth model with 14K vertices yields an additional cost of 0.5 secs simulation time per frame on a 2GHz Pentium 4. Changing mesh topology [105] during simulation for 1K triangles runs at 5-20 minutes per frame, and for 380K tetrahedra, runs at 20 minutes per frame. Stacked rigid body simulation [70] with simulations of 500-1000 objects takes 3-7 minutes per frame on average.

Finally, there is the entire field of interactive and real-time applications such as virtual reality, virtual prototyping and computer games. These distinguish themselves in two ways: Turn-around time is instant and end-users are interacting with the world.

In contrast to computers used in scientific computing, game consoles such as the Play Station 2, only has 32 MB RAM and 6.2 GFLOPS (`www.playstation.com`) and everyday living room PCs are for the most part inferior to those PCs used by computer graphics researchers. This has called for some creative thinking in computer game development to reach stringent performance requirements. Furthermore, there must be set time aside for other tasks in such applications as computer games. For instance, in the Hitman game from IO-Interactive, only 5-10% of the frame time is used for physics simulation [79]

The future may change these hardware limitations. For instance, recent GPUs do not suffer from latency problems and promise 60 GFLOPS (`www.nvidia.com`). Also PPUs (`www.ageia.com`) seems to be an emerging technology. Finally, Cell chips (`www.ibm.com/news/us/en/2005/02/2005_02_08.html`) are also promising.

From this short literature survey of recent work on physics-based animation in computer graphics, it is quite clear that methods are used which have frame times ranging from the order of seconds to hours running on single PCs. In conclusion, design work both in computer graphics and scientific computing requires reasonable low turn-around times.

This dissertation is mainly concerned with the demands of interactive and real-time applications, and not scientific computing nor special effects creation.

1.2 Classification

At the highest level, the field of physics-based animation and simulation can roughly be subdivided into two large groups called kinematics and dynamics. Kinematics is:

> The study of motion without considerations of mass or forces.

Dynamics is:

> The study of motion taking mass and forces into consideration.

But the story does not end here, because kinematics and dynamics come in two flavors or subgroups:

- Inverse.
- Forward.

In the first subgroup, one typically knows where to go, but needs to figure out how to do it. As an example, one might know the end position of a tool frame of a robot manipulator, without knowing what forces and torques to apply at the actuators in order to get to the final destination. In other words, inverse kinematics and dynamics computes the motion "backwards". Forward kinematics and dynamics work exactly in the opposite way. Using the same example as before, one typically knows the starting position of the tool frame and the forces and torques acting through the actuators. The goal is then to predict the final destination. The difference between inverse and forward kinematics/dynamics is illustrated in Figure 1.2.

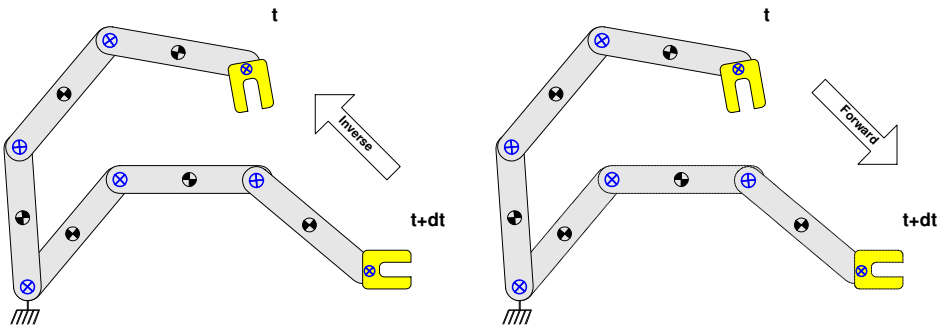


Figure 1.2: Example showing the difference in forward and inverse kinematics/dynamics. In the inverse case, the starting and ending positions of the tool frame (yellow) is known, whereas in the forward case, the end position must be predicted.

1.3 Examples

There are numerous techniques and methods in physics-based animation and simulation, and you have probably already heard several buzz-words. In the table below, we have tried to classify some of the most popular techniques and methods according to the four subgroups introduced in the previous sections:

	Inverse	Forward
Kinematics	• Cyclic Coordinate Descent • Jacobian Method	• Spline Driven Animation • Key Frame Animation (Interpolation) • Closed Form Solutions • Free Form Deformation
Dynamics	• Recursive Newton Euler Method • Optimization Problems	• Featherstone's Method (The Articulated-Body Method) • Composite-Rigid-Body Method • Particle Systems, Mass-Spring Systems • Finite Element Method • Constraint Based Simulation

1.4 Introduction to Multibody Dynamics

Multibody dynamics deals with simulating multiple rigid bodies possibly connected to each other through joints. Sometimes the term rigid body simulation is used to distinguish between simulators handling jointed mechanism or not.

Particles and rigid bodies are often the first concepts introduced in physics, and they are regarded as the basic theoretical building blocks. The same view is also often applied in physics-based animation: Rigid body simulation is considered the basic starting ground for many practitioners.

Many consider multibody dynamics to be more simple to simulate than chaotic and turbulent natural phenomena such as water, smoke, or fire. In our opinion, this is a misconception, and although the free motion of a single rigid body floating in empty space is embarrassingly simple to simulate, rigid bodies are ideal models of the world. However, in the real world everything will deform. For rigid bodies, this idealization result in difficult discontinuities, causing the numerics to become ill-conditioned and sometimes even unsolvable.

The dynamics and mathematics of rigid bodies have been known since Newton. Nevertheless, nearly 400 years later, it is still a topic of active interest and scientific publication, and since the 80's there have been papers on rigid body simulation in nearly every ACM Siggraph Proceedings. The explanation is that even though the physics and mathematics is well-established, it is not easily solved on a computer, and there is a constant demand to simulate more and more rigid bodies faster and faster. Furthermore, in animation we often want to simulate unreal things that should appear plausible. Hence, methods are needed that are incredibly fast, stable and robust, and tolerant for faulty configurations. None of these goals are the main focus in classical mechanics.

The traditional approach for analyzing systems in classical mechanics often deals with system in equilibrium. This is of little interest in animation, where we want to see objects in motion and colliding with each other. At equilibrium, the animation is usually over. In contrast, robotics has a long tradition for simulating mechanics. A state of equilibrium is often the goal in this field. Furthermore, problems are more concerned with kinematics of a single mechanism, controlling, planning or computing the motion trajectory in a known and controlled environment. In animation nothing is known about what should happen, and often only animation of several mechanisms is interesting. Besides, robotics tends to be aimed at only simulating the joints of a mechanism and not the contact forces with the environment or other mechanisms.

The quest for bigger and faster simulators seems endless and is mainly driven forward by the computer gaming industry and the movie industry: Rigid bodies are attractive in computer games, since they fit in nicely with the level of realism, and they are fairly easy to use to plan game events and build game levels. Basically, the industry knows how to use rigid bodies to create interesting game effects. The movie industry has moved beyond rigid body simulation to deformable objects, natural phenomena, humans et cetera.

This dissertation is a thorough treatment of state-of-the art in multibody dynamics. The theory and methods we present are highly biased towards our own work in the field, and the reader should not expect a complete, in depth, and detailed walk-through of every method and paradigm in this field.

Our contribution on multibody dynamics in this dissertation will begin with a formal,

conceptual modular design. The modular design is useful for explaining many details of rigid body simulators, such as interaction with the collision detection engine or the time-stepping strategies.

Hereafter, a method for creating arbitrary moving scripted bodies is presented. A scripted body is an object fully under the control of an animator. However, all other bodies in the simulation should interact in a physically plausible way with the scripted body.

Following this, a thorough introduction to constraint based simulation using complementarity formulations is given. Constraint based methods is probably the kind of method that resembles a physicist's view of the world. Constraint based methods set up the equations of motion, and solve explicitly for the contact and constraint forces, using the fact that Newton's second law can be integrated to calculate the complete motion of the objects. Our treatment of constraint based simulation contains insights on how to deal with error correction, time-stepping and managing a wide range of constraints in an object oriented manner.

Next, we will treat and present contact graphs. The contribution is twofold: A formal presentation of contact graphs and various speed-up techniques based on exploiting contact graphs.

The final contribution in this dissertation consists of the construction of a new kind of simulator combing velocity based complementarity formulations with shock propagation. Several techniques such as iterative LCP solvers and error correction are used to yield a fast, stable and robust rigid body simulator. A little insight is given into some of the collision detection problems encountered in the construction of this new simulator.

The aim with the multibody dynamics theory presented in this dissertation is to equip the reader with the skills for building stable and robust multibody simulators. That is to say, simulators capable of simulating large configurations without breaking down or giving up on unexpected errors or faulty starting conditions. We have put some focus on high performance, but this is more from an algorithmic point of view and not our most important concern. Due to space considerations we have left out recursive coordinate methods (also called minimal coordinate methods). For the interested reader we refer to the work of Mirtich [100] and Featherstone [59].

Chapter 2

A Modular Approach to Rigid Body Simulators

It is widely accepted that implementing a rigid body simulator is both difficult and time consuming with a steep learning curve due to the massive amount of theory. Possible reasons may be that:

- Dynamic simulation covers a wide range of research fields in computer science: Algorithms, computer graphics, computational geometry, numerical methods, et cetera.
- The literature of the field often presents a single algorithm or data structure and usually only in the context of a specific paradigm.

In the following, we are going to look at the "big picture," that is to say the glue which binds all the smaller pieces into a large simulator.

A rigid body simulator is often broken down into smaller pieces, each responsible for handling a specific task in the simulator, e.g. computing a collision impulse, determining the time of impact, and so on. We call such small pieces modules. In the following a unified, general purpose modular design for rigid body simulators is presented. A benefit of the modular design is its generality that allows for understanding and comparing the following simulator paradigms:

- Constraint-based Methods
- Penalty-based Methods
- Impulse-based Methods
- Collision Synchronization
- Hybrids

The modular design was originally developed for educational reasons [45], and has been used as the framework for a course in rigid body simulation [2]. It was a rapid development tool, allowing students to build their rigid body simulators within a few weeks. The modular design has also been used commercially in an application for visual configuration [3].

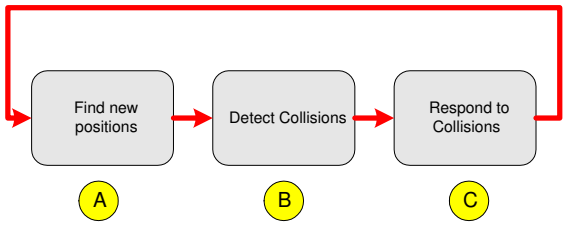


Figure 2.1: The Simulation Loop.

It is possible to implement a general application programmers interface (API) based on the modular design presented here. However, much performance and efficiency is lost if full support for all simulator paradigms is kept. Therefore, we recommend using the modular design as a concept from which a tailoring may be made for the particular simulator chosen.

The following chapter is organized into two parts. Firstly, the reader is taken on a detailed tour through a simulation loop by which a frame is computed. Secondly, the variations on the modular design for the various simulator paradigms is discussed.

2.1 A Modular Design

This section outlines the general purpose modular design by walking through the computation of a single frame in the simulator, and we will henceforth call this a simulation.

A frame is a single picture in a movie. In rigid body simulation, we are simulating a configuration which is a collection of user specified objects. At any given time the configuration has a state which is simply the concatenation of the states of all the objects in the configuration. The state of an object is typically given by its position and velocity. A frame is therefore equivalent to a snapshot of the configuration state at a given point in time. The time between two consecutive frames is called the frame-time, and a typical value is 1/25 secs. The state of the configuration at the start of the simulation is called the initial configuration and the configuration state at the end of the simulation is called the final configuration. The corresponding points in time are likewise referred to as the initial and final time. If the simulator computes a configuration state between the initial and final time, then the state is referred to as an in between or intermediate state.

The Simulation Loop [104] is a simple, three state iterative loop, shown in Figure 2.1. State A computes new positions, State B runs the collision detection, and State C, applies forces and/or impulses to avoid penetration.

At the highest level, the simulator design is split into two components: A collision detection component and a simulation component. This is shown in Figure 2.2. This particular bisection is convenient for two reasons: Firstly, collision detection may easily be the single most computationally complex element of the simulator, and secondly, the collision detection is a purely geometric problem, while the simulation mostly concerns physically inspired partial differential equations of motion. In terms of The Simulation Loop, States A and C are seen to belong to the simulation component and State B belongs to the collision detection component. The usage of the word collision can be a little confusing. When talking about collision detection, collision refers to a pure geometric

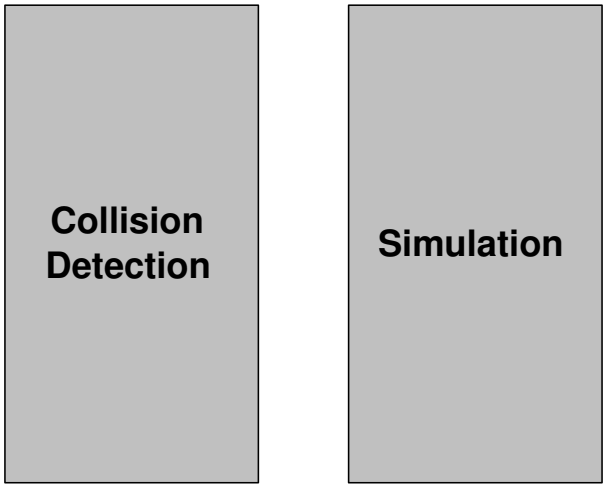


Figure 2.2: At the top level a simulator consist of two components.

problem: does object geometry penetrate or not? Collision detection comes in two flavors: the discrete case, where one tests for interference at a single point in time and the continuous case, where one tests for any collision over a time-interval [122, 123, 125, 126]. On the other hand collision response deals with the physical aspect: how to alter motion of objects such that they do not fly into/through each other. Dealing with the physical aspect, geometrically colliding objects are divided into physically colliding objects or resting objects.

2.2 The Simulation Component

The simulation component is responsible for computing the physical motion of the objects in the system, and the design consists of four modules:

- Time-Control Module
- Motion Solver Module
- Constraint Solver Module
- Collision Solver Module

These modules are illustrated in Figure 2.3 and will be described in the following sections.

2.2.1 The Time-Control Module

The time-control module is the central part of the simulator, controlling when and how all other modules in the simulator are invoked. The simulation loop is initiated by a request to the time-control module for a simulation of the configuration from the initial time to the final time. After completion, the resulting configuration state is returned (see Figure 2.3). The time-control module may use *fixed* and/or *adaptive time-stepping* algorithms to simulate forwardly, where adaptive algorithms are either *backtracking* or

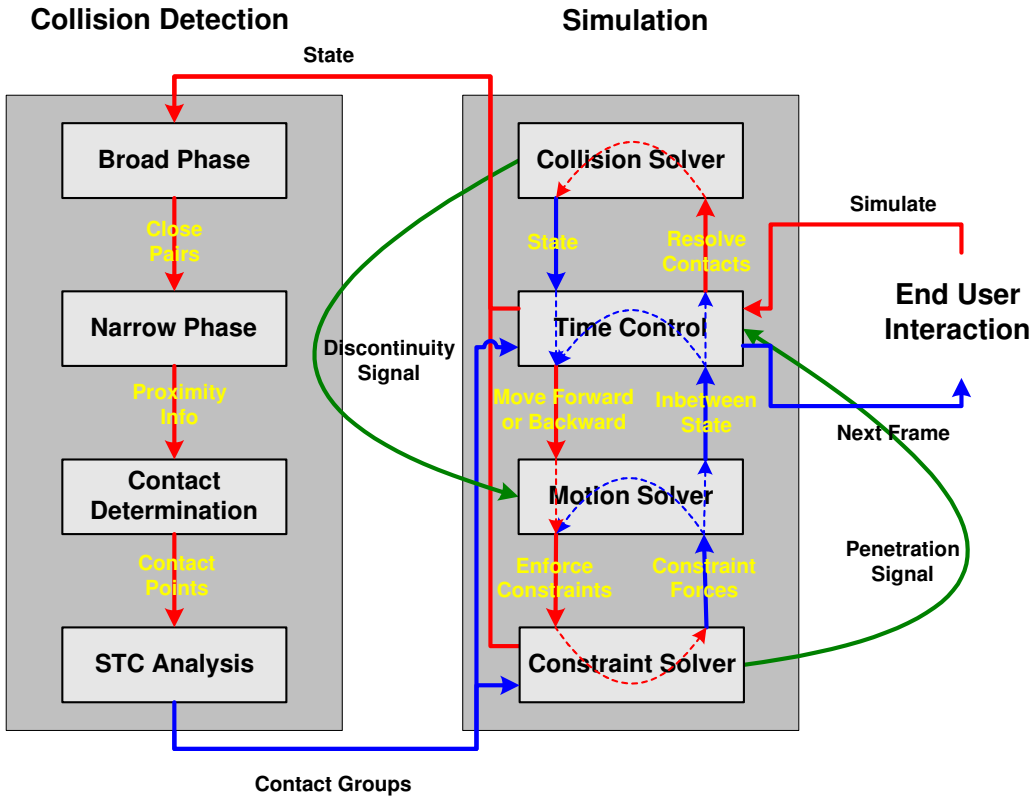


Figure 2.3: Complete General Purpose Modular Design. Red arrows show invocation, blue arrows show returned results, and green arrows are special cases. Dashed lines indicate computations by modules. Analogous to humans, the time-control is the heart, red arrows are the aorta, blue arrows the veins, and green arrows the nerves.

one-sided. These are also known as retroactive detection (RD) or conservative advancement (CA) [130] algorithms.

The present modular design is known to work for many simulator paradigms, and the chosen paradigm strongly influences the interaction of the time-control with both the motion solver and the collision solver. It is our experience that the modular design works for all paradigms discussed in Section 2.4, but the implications of the chosen paradigm will be postponed for discussion in that section. Instead, the main ideas of the three algorithm types will be reviewed in the time-control module.

With *fixed time-stepping* algorithms [106, 144, 118, 23], the time-control module asks the motion solver module to simulate forward by a fixed step size until the final frame is reached. The fixed step size is typically an order of magnitude smaller than the frame-time requested by the user. Nevertheless, both deep penetrations and overshooting (also called tunneling, i.e. objects flying through each other without detecting a collision) can happen if the fixed step-size is chosen to be large when compared to object sizes and velocities. Methods that overcome these drawbacks are for example Stewart's method [140], which can take bigger time-steps without causing penetrations. Another example is collision synchronization [96], also known as optimization-based animation, which is capable of taking steps of the same size as the frame-time.

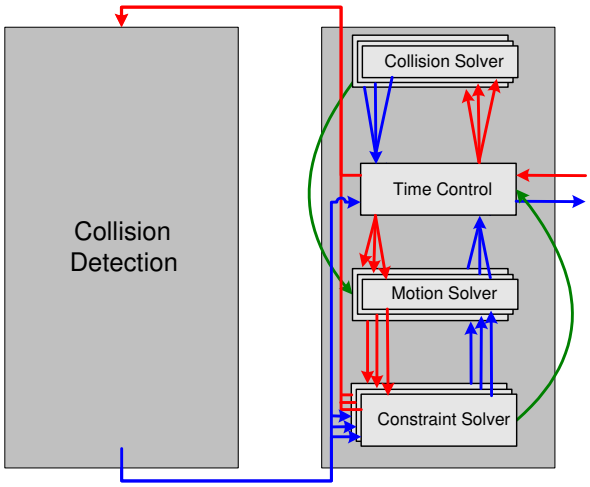


Figure 2.4: The Timewarp Algorithm. The object motion is desynchronized and global simulation time is synchronized in the time-control.

Adaptive time-stepping algorithms originate from traditional root search algorithms. These algorithms search for the first point in time, when the objects in the configuration are in touching contact. Both the backtracking and one-sided approach algorithms use the collision detection to do this.

The *backtracking time-stepping* algorithm [12, 101] asks the motion solver to simulate forward some time-step, followed by a query to the collision detection for possible penetrations. In the case of a penetration, the time-control will ask the motion solver to backtrack to the last known "good" state. The forward step will now be repeated with a smaller time-step, and the process is repeated until the objects either are separated or in touching contact. In the latter case, the collision solver is asked to handle any collisions, and the process repeats until the end of the simulation is reached.

The *one-sided* algorithm [100, 42, 76, 31] never allows for penetrations. Instead, an estimate for the earliest time of impact (TOI) is computed, and the estimate is used to simulate forward. The time of impact is calculated by first asking the collision detection for the smallest distances between all pairs of objects in close proximity, and secondly using the velocities and accelerations of the objects to calculate a conservative estimate for the time of impact between each pair of objects. The time-control picks the smallest time of impact and asks the motion solver to simulate forward to the time of impact. Afterwards, the collision solver is invoked to handle any collisions, and the pattern repeats itself until the end of the simulation is reached. As objects move closer and closer, and times of impact are recomputed, the estimates become increasingly accurate until the estimate is considered exact within some given precision.

Hybrid time-control algorithms exist [12] that actively choose the appropriate algorithm depending on various criteria. Finally, an alternative approach has been suggested which is reminiscent of distributed algorithms [103], see Figure 2.4, where object motion is desynchronized by applying separate instances of the collision solver, motion solver, and constraint solver modules on each contact group. The global simulation time is synchronized in the time-control.

To conclude, it is emphasized that the computation of a frame is typically the result

Symbol	Description
$Y(t)$	The state function
$\vec{r}$	The position of center of mass
q	The orientation as a quaternion
$\vec{P}$	The linear momentum
$\vec{L}$	The angular momentum w.r.t. center of mass
$\vec{v}$	The linear velocity of the center of mass
$[0, \vec{\omega}]$	The angular velocity as a quaternion
$\vec{F}$	The total force
$\vec{\tau}$	The total torque w.r.t. center of mass
$\vec{a}$	The linear acceleration of the center of mass
$\vec{\alpha}$	The angular acceleration

Table 2.1: Notation used in typical ODEs.

of several iterations of the time-control. Figure 2.3 illustrates the interaction between the time-control module and the other parts of the simulator.

2.2.2 The Motion Solver Module

The motion solver module is frequently called from the time-control module, and it is responsible for the continuous movement of all objects in the configuration according to the equations of motion. These equations are either given as ordinary differential equations (ODE) [18] or as scripted motion [100, 53, 54]. The ODE for the motion of an object is typically given as:

$$\frac{d}{dt}\begin{bmatrix} \vec{r} \\ q \\ \vec{P} \\ \vec{L} \end{bmatrix} = \begin{bmatrix} \vec{v} \\ \frac{1}{2}[0, \vec{\omega}]\, q \\ \vec{F} \\ \vec{\tau} \end{bmatrix},$$

where the notation is described in Table 2.1. Scripted motion is typically given as:

$$Y(t) = \{\vec{r}, q, \vec{v}, \vec{\omega}, \vec{a}, \vec{\alpha}\}\,.$$

Both equations describe the state of a single object, and typical configurations contain both objects governed by ODEs and objects governed by scripted motion.

The ODE typically depends on constraints and external forces, and in contrast the scripted motion is independent on any forces in the configuration. This leads to a specific order in which the motion is solved for mixed configurations. The motion solver will use an ODE solver to compute the new state for those objects governed by ODEs. However, for each integration step in the ODE solver, the motion solver will first compute the new state of the scripted bodies, then all external forces, and finally any constraint forces acting on the objects. To compute the constraint forces, the motion solver sends the current intermediate state of the configuration to the constraint solver. The constraint solver then computes any constraint forces and returns them to the motion solver which finally applies them to the ODEs.

Typically, the motion solver will record all the intermediate states [100] for backtracking purposes or as valuable information for the time-control module. To our knowledge, there does not exist a method for computing conservative time of impacts for articulated bodies, and the time-control may therefore verify previously computed estimates of time of impacts by using the state tracking information.

From the above description, it is apparent that there is a two-loop structure in a single computation of a frame: The outer loop in the time-control module and the inner loop in the ODE integration steps of the motion solver. The size of the integration step depends on the object configuration, the desired accuracy, and the numerical stability of the system. To our experience, integration time steps are typically of size $0.01-0.001$ secs., and in some difficult cases such as stacked environments the time step can be extremely small: 10^{-6} secs. or lower. For real-time simulation, the inner loop should run at a rate of a thousand of times or more per second, and this is thus a natural performance bottleneck. The collision synchronization [96, 130] paradigm is specifically designed to attack the problem of small integration steps in rigid body simulation. Figure 2.3 shows how the motion solver module interacts with other modules in the simulator.

2.2.3 The Constraint Solver Module

The motion solver invokes the constraint solver to retrieve the constraint forces which are used by the ODEs to compute the continuous motion of objects. Constraint forces are imposed in order to prevent objects from penetrating each other. Constraint forces are often distinguished as either bilateral ("=") or unilateral ("$\geq$") constraints. For instance, joint forces act as constraints on the links of articulated figures and are therefore bilateral, whereas contact forces are unilateral constraints arising from resting contact between objects. The total interaction of the constraint solver module with the remainder of the simulator is shown in Figure 2.3.

To compute the contact forces [29, 145, 9, 8, 7, 6, 38, 59, 4, 117, 101, 16, 138, 18], the constraint solver queries the collision detection for the contact regions between the objects. For constraint forces there exist essentially two different computing approaches: Analytical or penalty-based. *Analytical approaches* use a system of equations and solves the constraints analytically. Conversely, *penalty-based approaches* allow for penetrations and add penalty forces at points of penetration.

There are numerous variations on how and when the constraint solver is invoked in the inner ODE loop:

Eager: Invoke collision detection and recompute constraint forces for each step in the loop.

Moderate: Run collision detection for the first ODE integration step, and reuse the contact regions for the rest of the loop.

Lazy: Compute constraint forces for the first step in the ODE integration, and reuse them for the rest of the loop.

The correct choice depends on the accuracy needed versus the speed of the simulator. The eager strategy gives highly accurate simulation but requires many calls to the collision

detection module which implies that the simulator will be slow. Conversely, the lazy strategy will be fast but imprecise.

The other variations are computationally more tractable but also more likely to produce wrong simulation results because the contact regions can change tremendously during the continuous movement of an object. As an example, consider a high speed ball rolling off a table top in a standard gravity field. For the lazy strategy, the normal force from the table top is applied to the ball even when it no longer touches the table top (this was coined road-runner physics by Baraff [17]). In contrast, the eager strategy will register that the contact region has changed, and the ball will drop under gravity as it leaves the table top. Many people favor speed over precision and thus choose the lazy strategy [38].

The constraint solver invokes the collision detection with a much higher rate than the time-control module. Therefore it might happen that a penetration which the time-control module has overlooked is detected in the constraint solver. If this happens, the constraint solver should notify the time-control about the penetration in order for the time-control to be able to take appropriate action.

2.2.4 The Collision Solver Module

The collision solver module is called from the constraint solver, and it computes collision impulses and applies the impulses to all the colliding objects in the configuration, as shown in Figure 2.3. The application of an impulse to an object causes a discontinuous change of the object's motion, and the motion solver should be notified of such discontinuous change, in order for it to be able to update any state information it might store. Recall that the motion solver only handles the continuous movement of the objects, i.e the smooth movement. A collision is like a bump in the smooth motion. Alternatively, rigid body collisions may be solved by compliant contact models [87], where the rigid body assumption is eased and wave- and time propagation in the collisions can be modeled. Compliant contact models will not be treated further in this dissertation.

Computing impulses can be done by using either an algebraic law, an incremental law, or a full deformation law [35]. *Algebraic laws* solve a system of equations and are usually very fast to compute. They describe the net collisional interaction by relations between pre- and post-collision quantities. *Incremental laws* use a microscopic collision model which is integrated over the collision. Incremental laws are computationally more intensive than algebraic laws. A *full deformation law* solves a partial differential equation describing how the physical quantities change during a collision. Full deformation laws are not seen in real-time rigid body simulation for two reasons. Firstly, it is almost impossible to determine the starting conditions for the partial differential equations, and secondly, the partial differential equations are very computationally demanding.

Impulses can be applied to the objects either through simultaneous impulses or by sequential (also called propagating) impulses [11, 35, 100]. In simultaneous impulse response, all impulses are computed in a single step and applied at the same time. In contrast, for sequential impulses the impulses are computed and applied one by one.

It is noticed that the contact regions where the impulses should be applied have already been computed when the collision detection is invoked by the time-control, and thus the contact regions may conveniently be given as arguments to the collision solver.

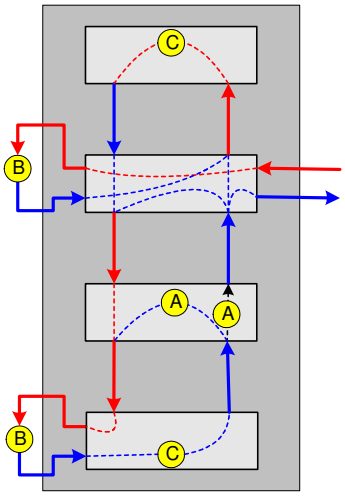


Figure 2.5: The states of The Simulation Loop (see Figure 2.1) shown at the corresponding places in the modular design of the simulation component (right part of Figure 2.3).

Thus there is no reason for invoking the collision detection in the collision solver.

2.2.5 The Simulation Loop

The time-control, the motion solver, the constraint solver, and the collision solver module complete the simulation component of the simulator, as shown in Figure 2.3. If we compare the flow in the modular design with the states described in The Simulation Loop in Figure 2.1, the states may be rediscovered as an integrated part of the simulation component as shown in Figure 2.5. It is seen that the modular design has a loop similar to The Simulation Loop, nevertheless, it is also clear that more details are to be found in the modular design: the collision response is separated into a continuous and discontinuous part, and the loop starts by handling discontinuous collision response.

2.3 The Collision Detection Component

Collision detection is a purely geometrical problem of intersecting objects, and it is also the computationally most intensive part of a simulation. Most collision detection algorithms concern inter-object collisions [106, 37, 77, 32, 155, 88, 102, 86, 148, 90, 67, 18, 43, 147, 28, 83, 68, 84, 70]. Other algorithms are concerned with first point of contact [31, 42, 136, 126, 125, 123, 122]. This is termed continuous collision detection.

The collision detection component is called when the simulation component has computed the positions of all objects in the system. The collision detection then examines the geometry of all the objects in order to find touching and penetrating objects. The modules of the collision detection component are:

- The Broad Phase Collision Detection Module
- The Narrow Phase Collision Detection Module

- The Contact Determination Module
- The Spatial-Temporal Coherence (STC) Analysis Module

These modules are shown in the left part of Figure 2.3 and will be discussed in the following sections.

2.3.1 The Broad Phase Collision Detection Module

The naïve collision detection algorithm compares the possible intersection of all objects with all objects, implying $n(n-1)/2$ intersections for n objects. The purpose of the broad phase collision detection module [75, 76, 74, 77] is to prune the number of comparisons by a coarse scale analysis of the system, such that only object pairs in close proximity are tested by the narrow phase collision detection. The implication is that the computational burden of the collision detection component is considerably reduced.

Typical algorithms used for broad phase collision detection use exhaustive search, sweep and prune [37, 18] (also called coordinate sorting), or (multilevel) grids [100, 38] (also known as hierarchical hash-tables).

To determine close proximity, objects are often approximated by bounding volumes, and in some cases, the broad phase collision detection algorithm might benefit from using information about the motion of the objects. The object motion can be used to construct sweeping volumes which act as bounding volumes enclosing the motion of the objects in the near future [100]. Alternatively, space-time bounds [31, 76] may be used instead. The sweeping volumes or space-time bounds are usually delimited in time by a look-ahead time-step argument to the broad phase collision detection module (not shown in Figure 2.3). The use of sweeping volumes or space-time bounds gives near collision information about objects, which is especially useful for one-sided time-control approaches.

2.3.2 The Narrow Phase Collision Detection Module

The narrow phase collision detection module examines pairs of objects in order to discover if the objects are colliding or not. The algorithms used by the narrow phase collision detection module are usually capable of returning more information than a yes-no answer. Often contact points, penetrating features and other proximity information is returned [106, 37, 77, 32, 155, 88, 102, 86, 148, 90, 67, 18, 43].

There is a large number of narrow phase collision detection algorithms reported in the literature, and it is out of the scope of this dissertation to report all of them, and since there are only minor implications on the modular design of the choice of narrow phase collision detection algorithm, we refer the reader to the references for further details. Nevertheless, the choice of algorithm does influence the contact determination module to be described in the following section.

2.3.3 The Contact Determination Module

The proximity information returned from the narrow phase collision detection module can be used with great advantage in the contact determination module [101, 81, 142, 18].

The contact determination module computes the contact regions between touching or penetrating objects.

In mathematical terms, the contact region is the intersection of the two object surfaces in touching or penetrating contact. For polygonal objects, contact regions are often represented as contact formations (CF), consisting of principal contacts (PC), i.e. pairs of geometric features, one from each object. The contact determination is a pure geometric problem of determining the contact region. However the problem is non-trivial considering uncertainties and non-uniqueness of representation.

For physics-based simulation, the computed contact regions can be recomputed as support regions [26, 29], also known as contact analysis [26]. Typically, there exists multiple solutions for the contact forces of a contact region. However, by computing the support region, the contact forces may be uniquely determined. This implies that there are fever constraints to solve for, making the computation faster.

It has been discussed [26] whether the computation of support regions should be placed in the collision detection or in the simulation components, since support regions are more related to physics than geometry. We prefer to associate the computation of support regions with the collision detection component, thus collecting all the geometry computations in one component.

2.3.4 The Spatial-Temporal Coherence Analysis Module

The final module in the collision detection is the spatial-temporal coherence (STC) analysis module. This module analyzes the configuration, detects independent contact groups, and exploits caching for computational efficiency.

Contact groups are groups of objects which are in either direct or indirect touching or penetrating contact. The groups may be used both by the constraint solver and the collision solver, since impulses and constraint forces can be computed for the objects in each group independently from objects in other groups.

The usage of contact groups can be taken even further by applying time warping to the simulator [103]. This means that each group is simulated independently of other groups, and synchronization between groups is only needed when objects from different groups interact. This is illustrated in Figure 2.4.

The contact group computation need not be postponed to the very last minute in the collision detection, but it is useful to have a constantly up-to-date contact graph with relevant information from the broad phase, narrow phase, and contact determination modules. This implies that Figure 2.3 is a little misguiding with respect to the contact group computation, but we have omitted the details for clarity.

It should be noted that some simulators [112, 150] do not compute contact groups in the collision detection component, but keep the contact groups in their time-control modules.

For an in depth treatment of spatial temporal coherence analysis we refer the reader to Chapter 5.

The simulation and collision detection components described above with their various modules completes the description of the general purpose modular design. The entire modular design is shown in Figure 2.3. There is no reason why the modular design is not applicable to other types of simulators such as particle systems and soft body simulators.

These differ mainly from the present description in the collision detection algorithms. We will now turn the attention to variations of the general purpose modular design due to the chosen simulator paradigms.

2.4 Simulator Paradigms

In rigid body simulation there exists five different kinds of simulator paradigms: Impulse-based, constraint-based, penalty-based, collision synchronization, and hybrids. In the following sections each paradigm will be discussed in detail.

2.4.1 Constraint-based Methods

A constraint-based simulator [59, 11, 14, 16, 129, 29, 117, 8, 154, 40, 146] is a very complex simulator paradigm utilizing all the modules of the general modular design, and as such, this paradigm has been treated in detail in Section 2.1. Constraint-based simulators do not allow for penetrations and are typically very good at handling complex configurations with static contacts.

One approach to the constraint-based simulator is based on Differential Algebraic Equations (DAEs) [33], where different constraints are used depending on the state. Recently, complementary formulations [94] have received much attention [11, 12, 13, 14, 15, 16, 18, 139, 145, 6, 7, 140, 4, 129, 29, 92, 154, 146]. For a more detailed insight into these methods and their historical development we refer the reader to [138].

2.4.2 Penalty Methods

Penalty-based simulators [14, 106] are simpler than constraint-based, and are preferred by many due to the simplicity of the modeling of physical interactions and since they may easily be extended to handle soft bodies.

The fixed time-stepping algorithm is often chosen, since the penalty methods allow for penetration of the objects. One major challenge for penalty methods is the collision detection, since both penetration depth and penetration points must be computed. Penalty forces are computed as the negative gradient of an energy function, and are used to contravene penetration. Hook's law for springs is a popular choice. Deep penetrations may cause stiff ODEs, and numerical stability is therefore of major concern with penalty methods. Figure 2.6 shows a typical penalty-based simulator. It should be noted that there is no collision solver, since collisions are handled when penetrations occur by the constraint solver which will add penalty forces to reduce but not necessarily remove penetrations.

Both numerical stability and minimal penetrations require small time-steps in the time-control module which is why some people [106] have tried to increase time-steps by augmenting the penalty-based simulator with a collision solver.

2.4.3 Impulse-based methods

Impulse-based simulators [71, 100] simulate all physical interactions between the objects in the configuration as collision impulses. Impulse-based simulators do not allow for penetrations. Static contacts such as one object resting on another is modeled as a series

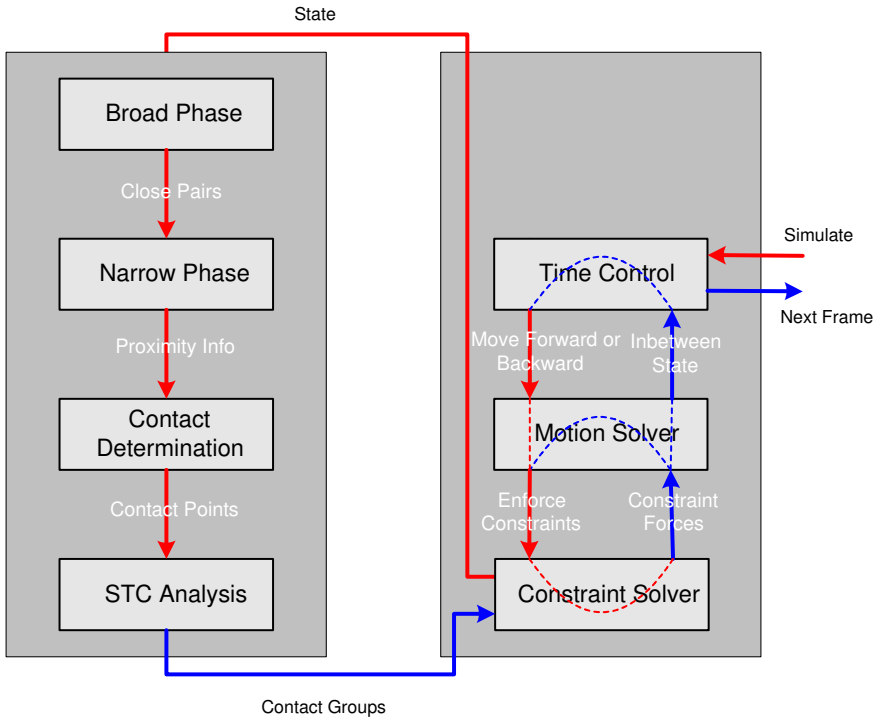


Figure 2.6: A typical penalty-based simulator.

of small micro-collisions occurring at a very high frequency. Except for static contacts, impulse-based simulators are computationally effective for systems having many objects moving at high speeds, and impulse-based simulators are therefore a good choice for real-time simulation.

As Figure 2.7 indicates, this kind of paradigm is particularly simple to implement. Typically a one-sided-approach is used together with sequential impulses based on some sort of incremental law. Impulses need only be applied at the closest points between two objects, and contact determination or spatial-temporal coherence analysis may therefore be omitted.

Alternatively to simulating static contacts as an impulse-train of impacts, the impulse may be computed as the time integral of the contact forces [140], in which case the modular design more resembles the constraint-based method, except that a nonlinear complementary problem for the contact impulses is solved in the constraint solver.

Recently, a new approach to impulse-based simulation has been suggested [70], in which a new time-stepping method is proposed, where collision resolution and contact handling are calculated in between the position and velocity updates. This is illustrated in Figure 2.8. Firstly, it should be noted that the new time-stepping approach completely changes the internal workings of the motion solver. Secondly, the collision resolving and contact handling is slightly changed, since for each iteration the predicted object positions are recomputed and used for the collision detection in a eager strategy. Thirdly, penetration constraints are not strictly enforced, and small penetrations are allowed. Finally, contact and collision handling are only done once per time-step, producing a plausible motion, while allowing for larger time-steps.

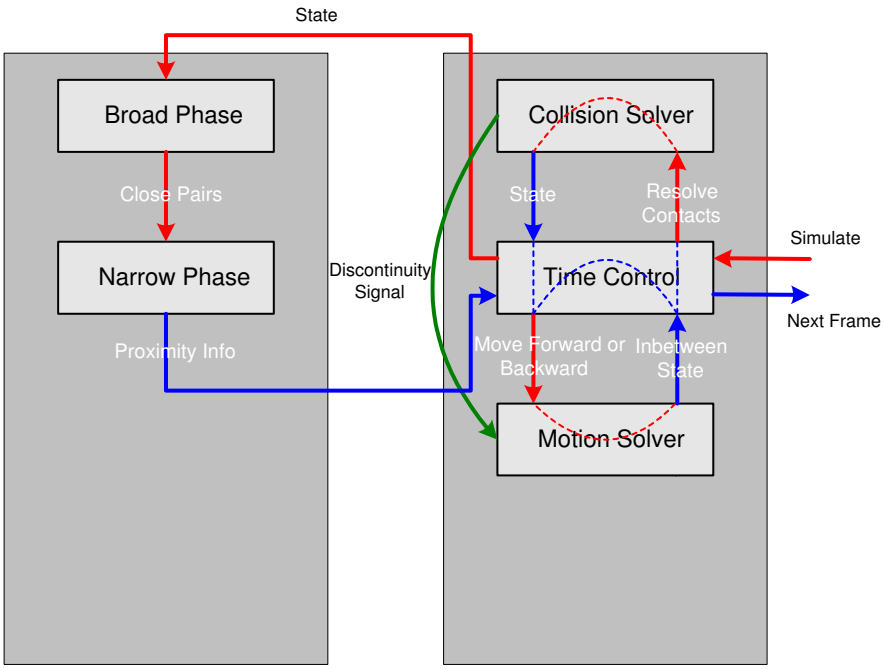


Figure 2.7: An impulse-based simulator using Mirtich's approach.

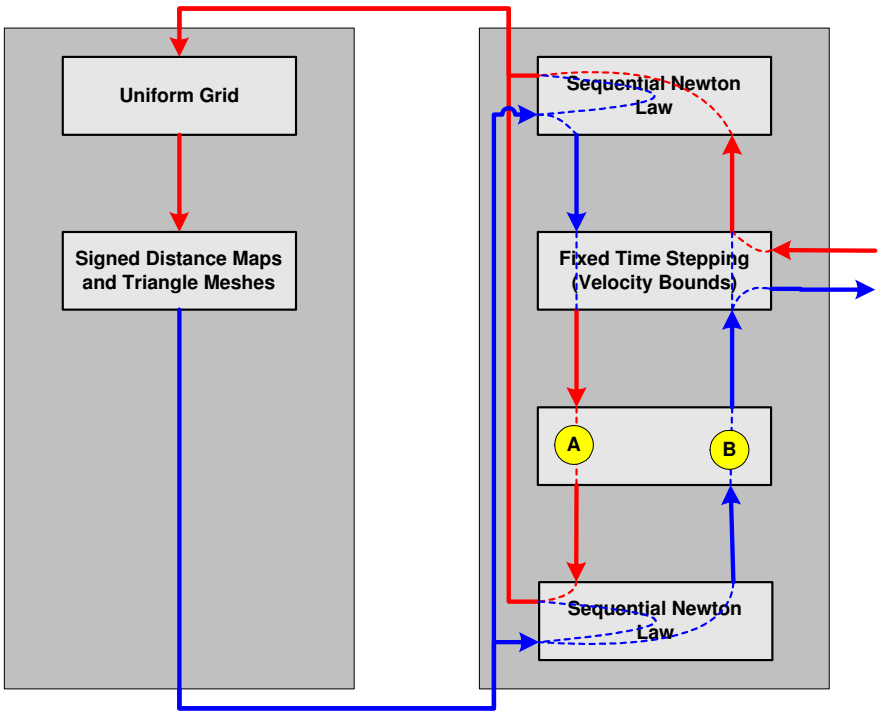


Figure 2.8: An impulse-based simulator using Guendelmann's approach. Velocity update occurs at the circled A, the position update occurs at the circled B, and both the collision resolving and the contact handling iterates several times with the collision detection. A contact graph is only used in the constraint solver.

2.4.4 Collision Synchronization

Collision synchronization also known as Optimization-based Animation [96, 130] extends position-based physics [97] and aggressively attacks the problem of small time-steps which all of the paradigms mentioned in the previous sections suffer from. Larger time-steps is achieved by synchronizing collision at the end of each frame with a so-called position-update method which sets up and solves a quadratic programming (QP) problem. The solution to the quadratic programming problem gives optimal non-penetrating positions for all objects at the end of the frame, and plausible motion is generated by finding a plausible physical motion agreeing with the solution. According to [96, 130] the paradigm may be used directly in any modularized simulator that mimics The Simulation Loop (see Figure 2.1).

2.4.5 Hybrids

Hybrid simulators [99, 100] attempt to combine several of the mentioned paradigms such that the weakness of one paradigm can be handled by another. The paradigms mentioned above may be programmed in the modular design, and hybrids thereof can naturally also be handled by the general purpose modular design.

2.5 Comparison With Existing Simulators

In this section we will review some widely available commercial and open-source rigid body simulators in terms of the modular design. The inner workings of many commercial simulators are naturally protected by their owners, in which case we may only guess at their implementation. Here we will discuss Open Dynamics Engine (ODE) [112], and Vortex [150].

2.5.1 Open Dynamics Engine

The Open Dynamics Engine [112] is a free, industrial quality library for simulating articulated rigid body dynamics developed by Russell Smith. In Figure 2.9 we have expressed Open Dynamics Engine version 0.035 in terms of the modular design presented here.

The user interfaces to Open Dynamics Engine by supplying a time-control mechanism that invokes the collision detection system. The collision detection then returns an array of contact points which the user must convert into so-called "joint contacts," before the motion solver can be invoked by calling "dWorldStep." This stepping method start by detecting "islands" equivalent to contact groups. Contact forces are computed independently for each contact group with a method similar to Stewart's [139, 140] on the entire group.

The source code of Open Dynamics Engine does not appear to be split into a motion solver and constraint solver, since the same function "dInternalStepIsland_xx" computes the constraint impulses and the position update. Drifting and penetration problems are handled by using an error reduction parameter (ERP) for controlling the amount of correction force based on a penetration depth penalty principle. Constraint force mixing

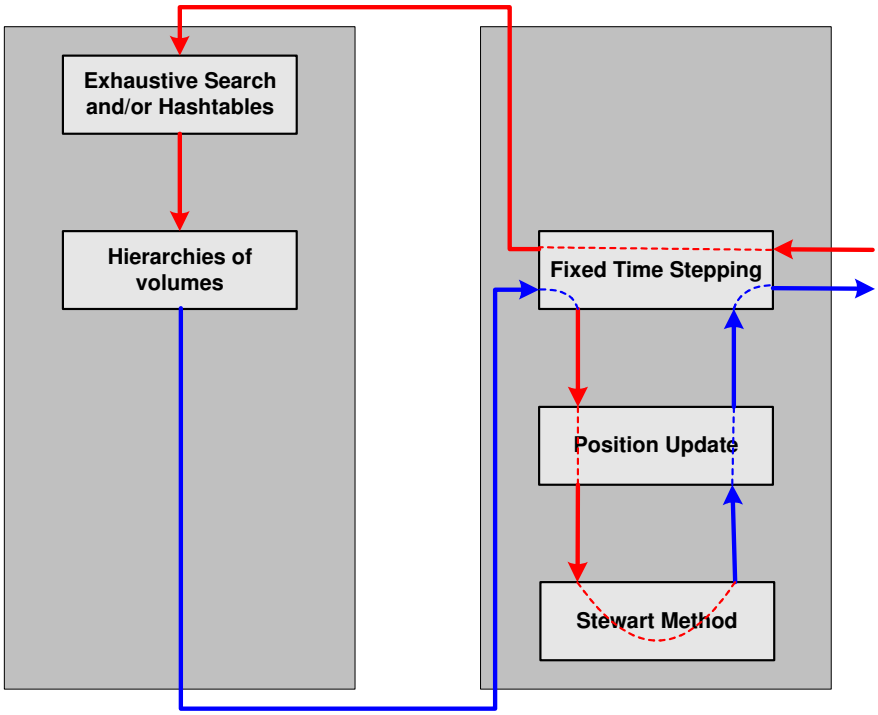


Figure 2.9: The Open Dynamics Engine version 0.035.

(CFM) allows for non-hard constraints by adding positive constants to the diagonal of the system matrix.

The Open Dynamics Engine may be implemented in the modular design, although it does not require the full complexity of the modular design. It should be noted that the Open Dynamics Engine detects contact groups in the simulation part rather than in the collision detection engine. However, this is of little significance and could easily be changed to accommodate the modular design.

2.5.2 Vortex

Vortex [150] is a commercial physics engine for real-time visualization and simulation developed by CMLabs. It is a constraint-based simulator based on a linear complementary formulation, and it provides two models of friction: A box-type and a scaled box-type. The box-type uses two tangential directions with upper limits on the friction force along the tangents, implying that friction does not scale with the magnitude of the normal force. The scaled box-type is identical to the box-type except it uses an extra iteration.

Vortex uses two algorithms for time-control: A fixed time-stepping algorithm and a time-of-impact algorithm. Fixed time-stepping comes in two flavors: an original mode ignoring collisions, and a so called stable stepper which resolves collisions before the system is stepped forward. Drift in joint constraints is handled by relaxation which is back-projection by a gamma factor. Non-singular configurations can be helped by increasing an epsilon parameter. The epsilon parameter is used to alter the system matrix so constraints appear more springy.

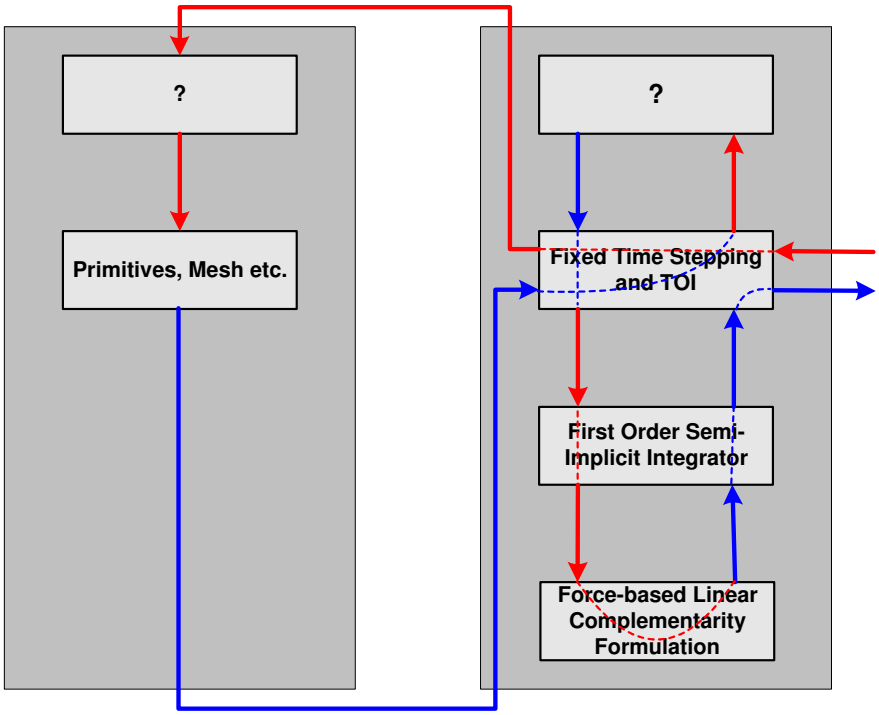


Figure 2.10: The Vortex Simulation Kit version 2.01.

The ODEs describing the equations of motion are solved by a first order semi-implicit integrator. Vortex appears to be using eager-evaluation of contact forces.

Broad phase collision detection is based on axis-aligned bounding-boxes (AABBs), and it is possible to control the number of axes used to determine overlap. We have not been able to discover what kind of algorithms are used for broad-phase collision detection or resolving collisions. Vortex contains a large number of algorithms for narrow phase collision detection between various types of geometries. There is no explicit concept of contact determination. Instead, it seems to be part of the narrow-phase algorithms. Like the Open Dynamics Engine, contact groups are not determined in the collision detection module, but instead they are determined in the stepper routine.

Vortex may be implemented using the modular design as illustrated in Figure 2.10.

2.6 Discussion

There is no such thing as a general purpose simulator capable of simulating everything, but it might be possible to explain and examine a wide spectrum of simulator types in a common framework.

The modular design is an attempt at such a framework. It can handle rigid body simulators regardless of the choice of simulator paradigm. However, we believe the modular design is much more versatile, and in the future we hope to show how widely applicable the modular design is.

Our experience is that it is possible to implement a general application programmers interface (API). However much performance and efficiency is lost if full support

for all paradigms is kept. Therefore, we often prefer to use the modular design and its eight modules as the right concept from which a tailoring may be made for a particular simulator.

Some of the interaction we have shown in the modular design was added for performance reasons. For instance, the green signal lines are meant to be used for lazy copying of state information. To give a specific example in the motion solver one would often communicate with an ODE solver by copying state information from bodies to arrays and then pass these arrays along to an ODE solver routine. But why not re-use the state information that was already present in an array in the last invocation of the motion solver? This way one can save time by avoiding to have to set up the arrays from scratch at the next invocation of the motion solver. The green signal line could be used to initiate an update of the part of an array containing state information about a body, where the state has changed. The simulators we have seen do not bother with this lazy copying. Instead, all state information is retrieved and copied to state arrays on every invocation of the ODE solver.

Likewise, the penetration signaling is often ignored. We usually implement this signaling by returning appropriate error codes from our methods, and not as a kind of distributed mechanism as Figure 2.3 might indicate.

The simulators we have reviewed do not explicitly use a time-control and a motion solver module. Instead, all the functionality is typically implemented in a single stepper routine on the simulator component level. The simulators do not even have a constraint solver module. Instead, matrices are often set up in the stepper routine and a LCP solver or something similar is invoked directly. In short, it is difficult to find an existing simulator where the modules we have outlined exist as independent objects.

In our opinion, there is no right or wrong modular design. Nevertheless, the modular design has been successfully used for education, commercial usage and research. In these settings, the design has two major benefits. Firstly, it handles all simulator paradigms in the same unified framework. Secondly, it provides modularity and overview.

Chapter 3

Spline Driven Scripted Motion

In this chapter, we present some ideas and theory for extending classical spline driven animation such that it can be applied in a rigid body simulator. The need for this rather specialized behavior arises due to what is called scripted bodies in dynamic simulation.

This work was done in collaboration with Knud Henriksen, and an early version of the work can be found in [54]. Here, a more detailed and elaborate version is presented.

In dynamic simulation a scripted body is used to avoid simulating the physically correct trajectories of the body. This is computationally favorable, and if one knows that the body is practically unaffected by the physical interactions of the other bodies in the configuration then the error is negligible. Consider for instance a vibratory part singulator which shakes small parts into recesses for automated assembly. The trajectory of the vibrator is unaffected by the physical interaction with the smaller parts. When dynamic simulation is used in animation we imagine that scripted bodies would be even more interesting, because they provide the animator with the means of constraining a body to move in a physically unexpected or exaggerated way, but still all other bodies will interact in a physically meaningful way with the scripted body.

Scripted bodies move very much like traditional objects known in animation. One specifies a trajectory by defining some key-positions at certain points in time, and the objects have to move along this trajectory "hitting" the key positions at the exact point in time where they are defined.

Generally speaking the problem is to determine the velocities and accelerations of the scripted bodies. These are needed in order to compute the physical interactions with the physically simulated bodies. In Chapter 4, velocity based complementarity formulations are discussed. These types of simulation do not require information about accelerations. However, to keep things general, we will here consider both velocity and acceleration.

People often use finite differences at key positions to estimate velocities and accelerations of the scripted bodies. This method is not well described in the literature and is mostly due to its trivial simplicity. Finite differencing is a successful technique for a wide range of applications, but there are some drawbacks.

The storage usage is linear in the number of key positions. In interactive and real-time applications there is often only a limited amount of storage available which means that there is a limit on how long motions and how many scripted bodies that can be simulated at the same time. A spline offers a more compact description of the trajectory of a scripted body than the equivalent key positions and the spline is therefore attractive from

a storage viewpoint. Furthermore, the spline offers a continuous description of the entire trajectory. On the other hand, using finite differences on key-positions naturally suffers from sampling artifacts if key-positions are not sampled densely enough. Furthermore, the spline representation offers a smooth description of the motion. Key-positions may have unwanted discontinuouties if too low-order differences are used although such artifacts would hardly be noticed in real-time applications. Finally, the spline driven approach decouples the actually geometry of the path traveled by a scripted body from the speed along the path. This provides an animator with an extra degree of freedom.

There are mainly two difficulties that arise. Firstly, there is the general problem of re-parameterization of the splines into their natural parameters (this is a general problem in spline driven animation and not specific for dynamic simulation). Secondly, scripted bodies can interact with physical bodies in a simulator and therefore there is a need for knowing something about their motion and not just their positions and orientations. The interactions typically involve computation of "time of impact", sweeping volumes, collision impulses and contact- and constraint forces. What we need are several different time derivatives specifying the motion of the object. These are:

$$\vec{v}(t), \vec{a}(t), \vec{\omega}(t) \quad \text{and} \quad \vec{\alpha}(t) \tag{3.1}$$

That is, the linear velocity $\vec{v}(t)$, the angular velocity $\vec{\omega}(t)$, the linear acceleration $\vec{a}(t)$, and the angular acceleration $\vec{\alpha}(t)$. Putting it all together we are actually looking for a function $\vec{Y}(t)$ which specifies the scripted motion, i.e. the position $\vec{r}(t)$ and orientation $q(t)$, of a scripted body in a simulator, such that.

$$\vec{Y}(t) \mapsto \{\vec{r}(t), \vec{v}(t), \vec{a}(t), q(t), \vec{\omega}(t), \vec{\alpha}(t)\}\,. \tag{3.2}$$

Here $q(t)$ denotes a quaternion representation of the orientation. In correspondence with rigid bodies the function $\vec{Y}(t)$ can be seen as the state function of a scripted body. We define the scripted motion problem as finding a solution for the $\vec{Y}$-function. In this chapter we have concentrated on handling the linear motion only. However, with small changes the techniques can be applied to handle the rotational part as we discuss in Section 3.5.

This chapter is not about how to handle interactions with scripted bodies in a dynamic simulator nor is it about dynamic simulation as such. The reader is not required to know anything about dynamic simulation, but it would probably give a better picture of why we have stated our problem as we have.

Much of the theory and definitions concerning splines and spline driven animation is presented in a rather compact form to save space. The reader should refer to our references for more details if they are needed.

We have organized the chapter in such a way that we start by presenting our motivation for solving the scripted motion problem. Then we will outline the basic idea in spline driven animation and the problems with computing the derivatives of the motion. Hereafter, we present solutions for all the technicalities in the scripted motion problem. Finally, we discuss degenerate cases and future work before we draw our conclusions.

We have been working with dynamic simulation for a little while and when we began to embark upon the task of adding scripted bodies to our simulator we were surprised to see that most textbooks and papers on the subject simplified the problem by assuming that the state functions are known. Other approaches used implicit functions and such.

In either case we felt that the theory we encountered was not very easy for an animator to use compared to systems such as 3DS MAX, Rhino, Maya®et cetera. We think that dynamic simulation has a potentially huge application area in animation and it would be a shame not to pursue this. So we wanted to extend the traditional animation techniques which are intuitive to use and well known by most animators, such that they can be used in a dynamic simulator.

Another benefit which arises from our work is a unification of scripted body motion and dynamic simulation. Typical simulators have specialized algorithms and implementations to take care of each kind of scripted motion type: Implicit functions, sinusoidal waves, polynomials et cetera. Applying spline driven animation means that a wide range of motion specification and applicability can be done with the same type of scripted motion. When implementing a simulator it basically justifies looking on scripted motion as a black box making use of an unknow state function $\vec{Y}(t)$ and as such it makes life a little more easy for those people that implement simulators. Let us summarize:

- The work we present allows animators to use dynamic simulation with traditionally well known animation techniques.
- The solution we present allows a dynamic simulator developer to look at the motion of scripted bodies as though it were a (not needed to know) state function $\vec{Y}(t)$.

3.1 The Basic Idea

Systems such as 3DS MAX, Rhino, and Maya®offer a wide range of different splines and curves for key-frame animation. However, we will restrict ourselves to cubic splines.

Cubic splines are a good choice for specifying the trajectory of the linear motion because they are twice differentiable. It means that the motion along the spline is "smooth". It also means that the velocity and acceleration can be found by direct differentiation of the cubic spline. Our preferred choice is cubic nonuniform B-splines. This class of splines is easily converted into a composition of cubic curve segments (such as cubic Bezier curves see[52] for details) and they support nonuniform key positions.

Assume we have a trajectory given by a space spline $\vec{C}(u)$ parameterized by the global parameter u. We use the space spline to find points in space given a value u that is

$$\vec{C}(u) \mapsto (x, y, z) \tag{3.3}$$

However, the parameter u is not an intuitive parameter for humans to use. It is quite difficult for a human to predict exactly which point on a spline corresponds to a given value u. Humans are much better at thinking in terms of the arc length, s, instead of the spline parameter u. Therefore we would like to use a re-parameterization like

$$U(s) \mapsto u \tag{3.4}$$

Such that we have

$$\vec{C}(U(s)) \mapsto (x, y, z) \tag{3.5}$$

This re-parameterization makes sense since there is a one to one mapping between the parameter u and the arc length parameter s. This is due to the fact that if

$$u_1 < u_2 \tag{3.6}$$

then

$$s(u_1) < s(u_2) \tag{3.7}$$

A difficulty occurs if an animator uses a space spline. In this case, he would be interested in specifying how fast an object moves along the space spline. In other words, he might not want the object to move as the parameter u would dictate. Instead, he wants to specify a set of (t, s)-pairs, such that when the animation arrives at the time t then the object would have traveled the distance s along the space spline. These pairs of time and distance are usually represented by a so called velocity spline (The term velocity is historical and perhaps a little confusing in the context of dynamic simulation. A better word would probably be "traveled distance")

$$\vec{V}(v) \mapsto (t, s) \tag{3.8}$$

Looking at equation (3.4) we see that equation (3.8) isn't really usable. We are interested in changing the equation such that we can find an arc length function which maps from the time domain to the arc length.

$$s(v(t)) \approx s(t), \tag{3.9}$$

The details of this will be shown later. The velocity curve has to fulfill the property that there exists exactly one s-value for every possible t-value. Otherwise the motion that is dictated by the velocity curve would be impossible. It would require an object to be at two positions at the same time. This means that the graph of $\vec{V}$ must be a function of t. Putting it all together we see that the $\vec{C}$-function has been re-parameterized in the following way:

$$\vec{C}(U(s(t))) \mapsto (x, y, z) \tag{3.10}$$

This final re-parameterization allows an animator to work intuitively with the space curve in terms of its arc length and he can specify movements in terms of the animation time parameter, t, totally independent of the global parameter u.

Until now things seem pretty easy, however, we can not derive analytical functions for equations (3.4) and (3.9), at least not in the framework where $\vec{C}$ and $\vec{V}$ are both cubic splines. We have to look for a numerical solution of equation (3.10). It is not hard to imagine that things get even worse when we look for the first and second derivatives of equation (3.10). By applying the chain rule and product rule we get

$$\frac{d\vec{C}(U(s(t)))}{dt} = \frac{d\vec{C}}{du}\frac{dU}{ds}\frac{ds}{dt} \tag{3.11}$$

and

$$\frac{d^2\vec{C}(U(s(t)))}{dt^2} = \frac{d^2\vec{C}}{du^2}\left(\frac{dU}{ds}\right)^2\left(\frac{ds}{dt}\right)^2 + \frac{d\vec{C}}{du}\frac{dU}{ds}\frac{d^2s}{dt^2} + \frac{d\vec{C}}{du}\frac{d^2U}{ds^2}\left(\frac{ds}{dt}\right)^2 \tag{3.12}$$

Knowing that we work with cubic nonuniform B-splines or cubic curve segments the terms:

$$\frac{d^j\vec{C}(u)}{du^j} \qquad \text{for} \quad j = 0, 1, 2 \tag{3.13}$$

are easily computed, but the remaining terms in equations (3.11) and (3.12) are somewhat more difficult to compute since we can not find analytical representations for the U- and S-functions. That is the difficult terms are

$$\frac{d^2U}{ds^2}, \quad \frac{dU}{ds}, \quad \frac{d^2s}{dt^2} \quad \text{and} \quad \frac{ds}{dt}. \tag{3.14}$$

3.2 The Linear Scripted Motion Function

Now let us attack the problem of determining the values of the functions $U(s)$ and $s(t)$ given s- and t-values and the difficulties in computing their derivatives.

3.2.1 The Arc Length Function

If we have a cubic curve, such as a cubic Bezier curve, in 3-dimensional space

$$\vec{C}(u) = \begin{bmatrix} x(u) \\ y(u) \\ z(u) \end{bmatrix} = \begin{bmatrix} a_x u^3 + b_x u^2 + c_x u + d_x \\ a_y u^3 + b_y u^2 + c_y u + d_y \\ a_z u^3 + b_z u^2 + c_z u + d_z \end{bmatrix} \tag{3.15}$$

Or in matrix notation

$$\vec{C}(u) = \begin{bmatrix} u^3 \\ u^2 \\ u \\ 1 \end{bmatrix}^T M = U^T M. \tag{3.16}$$

The arc length function of the cubic curve is defined by the following function

$$S(u) = \int_0^u \mid \frac{d\vec{C}(u)}{du} \mid du \tag{3.17}$$

$$= \int_0^u \left(Au^4 + Bu^3 + Cu^4 + Du + E\right)^{1/2} du, \tag{3.18}$$

where

$$A = 9(a_x a_x + a_y a_y + a_z a_z), \tag{3.19a}$$
$$B = 12(a_x b_x + a_y b_y + a_z b_z), \tag{3.19b}$$
$$C = 6(a_x c_x + a_y c_y + a_z c_z) + 4(b_x b_x + b_y b_y + b_z b_z), \tag{3.19c}$$
$$D = 4(b_x c_x + b_y c_y + b_z c_z), \tag{3.19d}$$
$$E = (c_x c_x + c_y c_y + c_z c_z). \tag{3.19e}$$

Unfortunately, we can not find an analytical expression for this integral so we have to do a numerical integration in order to find the value of $S(u)$ given a u-value. The function

$$\mid \frac{d\vec{C}(u)}{du} \mid = \left(Au^4 + Bu^3 + Cu^4 + Du + E\right)^{1/2} \tag{3.20}$$

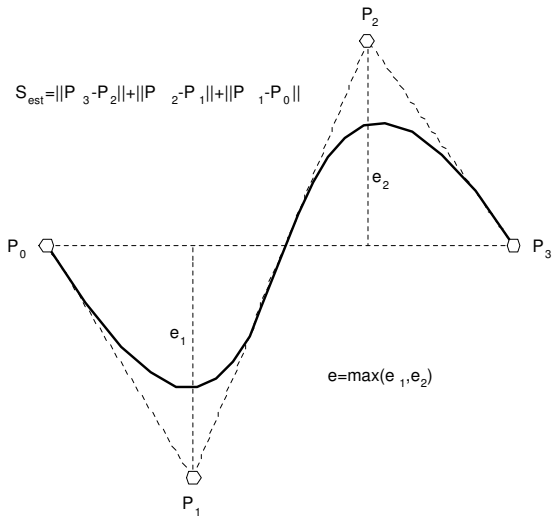


Figure 3.1: The arc length estimate of a Bezier curve and the flatness measure.

is the arc length integrand. By applying Horner's rule for factoring polynomials we can rewrite the equation into a more computationally friendly form

$$| \frac{d\vec{C}(u)}{du} | = \sqrt{(((Au + B)u + C)u + D)u + E}. \tag{3.21}$$

With the theory we have developed so far, we can pick any numerical integration routine which fulfills our requirements to performance and accuracy and then apply it in order to compute the arc length.

If one uses cubic Bezier curves then there is another approach to evaluate the arc length which is based on the convex hull property of Bezier curves and the de Casteljau algorithm [69]. The algorithm is quite simple and easily explained. Let the control points of a Bezier curve segment be $\vec{P}_0$, $\vec{P}_1$, $\vec{P}_2$ and $\vec{P}_3$. Now an estimate for the arc length is computed by

$$S_{est} = \sum_{i=0}^{2} |\vec{P}_{i+1} - \vec{P}_i| \tag{3.22}$$

If the Bezier curve is sufficiently flat then this estimate would be very close to the real value of the arc length. In order to determine if the Bezier curve is flat enough we measure the orthogonal distance, ε, of the two control points which are not the end points, to the line between the two end control points.

$$\varepsilon = \max\left(dist(\vec{P}_1, \overline{P_0P_3}), dist(\vec{P}_2, \overline{P_0P_3})\right). \tag{3.23}$$

This is illustrated in Figure 3.2. If ε is greater than some threshold value, then the curve is not flat enough. To handle the problem one uses the de Casteljau algorithm to subdivide the Bezier curve into two smaller Bezier curves. By the convex hull property of Bezier curves, we know that their convex hulls are smaller and tighter fitting, so we simply repeat the algorithm on each of these new Bezier segments. Every time we find a Bezier

```
Algorithm recursiveArcLength(B,eps)
  for i=0 to 2
    est += dist(B.P[i+1],B.P[i])
  next i

  L = line(B.P[0],B.P[3])

  err = max(dist(B.P[1],L),dist(B.P[2],L))

  if err > eps then
    (B1,B2) = subdivide(B)
    est = 0
    est += recursiveArcLength(B1,eps)
    est += recursiveArcLength(B2,eps)
  end if

  return est
End algorithm
```

Figure 3.2: Recursive computation of the arc length of a Bezier curve.

```
Algorithm recursiveArcLengthAt(u,B,eps)
  (B1,B2) = subdivideAt(u,B)
  return recursiveArcLength(B1,eps)
End algorithm
```

Figure 3.3: Recursive computation of arc length at given parameter value.

curve which is flat enough, we halt the recursive subdivision and add the estimated arc length to the total computed arc length so far. A pseudo code version of the algorithm can be seen in Figure 3.2. The beauty of the algorithm is that it computes the arc length adaptively, unlike some numerical integration routines like the extended Simpson which have a fixed step size. Furthermore, unlike those integration routines which can integrate with adaptive step size, the algorithm is far simpler to implement and quite fast.

There is one subtlety we have overlooked. The algorithm given above computes the arc length of the entire Bezier curve segment. We are interested in finding the arc length of the segment of the Bezier curve running from 0 to some value u. So we should do an initial subdivision to get the segment corresponding to the parameter interval we want to compute the arc length of. Figure 3.3 illustrates the idea.

In our case of cubic Bezier curves, the de Casteljau algorithm takes the form

$$\vec{P}_{k,i}(u) = (1-u)\vec{P}_{k-1,i}(u) + u\vec{P}_{k-1,i+1}(u) \tag{3.24}$$

Where

$$k = 1, \ldots, 3 \tag{3.25a}$$
$$i = 0, \ldots, 3-k \tag{3.25b}$$

And

$$\vec{P}_{0,i}(u) = \vec{P}_i \tag{3.26}$$

Actually, de Casteljau can be used to compute a point on the cubic Bezier curve because $B(u) = P_{3,0}(u)$. However we are going to use it for computing the new control points for the subdivided Bezier curves. The new control points for the Bezier curve representing the first half of the subdivision are

$$\vec{P}_{k,0} \quad \text{for} \quad k = 0, \ldots, 3 \tag{3.27}$$

and the control points for the Bezier curve representing the last half are

$$\vec{P}_{3-i,i} \quad \text{for} \quad i = 0, \ldots, 3 \tag{3.28}$$

3.2.2 Arc Length Re-parameterization

In this section we want to attack the problem of determining the re-parameterization function U, such that

$$\vec{C}(U(s)) \mapsto (x(s), y(s), z(s)) \tag{3.29}$$

The function $U(s)$ is obviously the inverse function of the arc length function $S(u)$. That is

$$U(s) = S(u)^{-1} \tag{3.30}$$

In the last section we saw that $S(u)$ could only be solved numerically. Therefore it is impossible to invert it in order to obtain the function $U(s)$. Fortunately, we can compute $S(u)$, so we can turn our problem into a root search problem. Given a value s, we search for a value of u such that

$$|s - S(u)| < \varepsilon. \tag{3.31}$$

If we find such a u-value then we have actually found $U(s)$ within the numerical tolerance ε. Recall that we have a one to one monotonic relation between the parameter u and the arc length s. This property ensures that exactly one root exists and our problem has a solution. Figure 3.4 shows a schematic overview of the process.

3.2.3 Time Re-parameterization

In an animation one usually knows a t-value and wants to find the corresponding s-value. As we have explained previously this sort of information is typically described by using a so called velocity spline. That is

$$\vec{V}(v) \mapsto (t(v), s(v)) \tag{3.32}$$

From this we need to find a function

$$s(v(t)). \tag{3.33}$$

The problem is very similar to the arc length parameterization problem. The main difference is that this time we are looking for a corresponding coordinate and not a "measure"

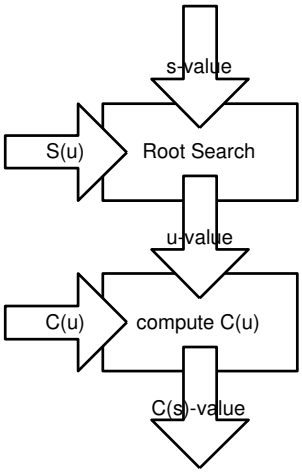


Figure 3.4: Schematic drawing of the method for arc length re-parameterization.

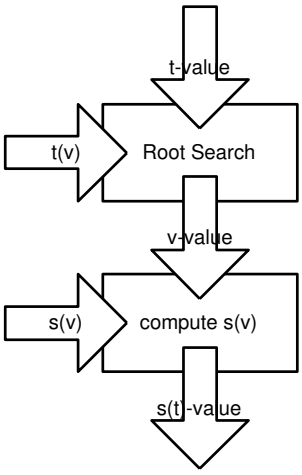


Figure 3.5: Schematic drawing of the method for time re-parameterization.

of the spline. However, the problem can still be solved in a similar fashion. That is, as a root search problem. Given a t-value we search for a v-value such that

$$|t - t(v)| < \varepsilon \tag{3.34}$$

When we find such a v-value then we have actually found the value of $v(t)$ and we can compute the s-value directly as $s(v)$. The time of the animation is a strictly increasing function as the animation evolves, so our root problem is guaranteed to have exactly one unique solution. This means our problem is solvable. Notice that the velocity spline has to be physically meaningful as we have explained earlier. This guarantees that we can always find an unique value for s given a t-value. Figure 3.5 illustrates the numerical process.

3.2.4 Computing the Derivatives

From equations (3.11) and (3.12) we see that we need to compute the following values

$$\frac{d^2\vec{C}}{du^2}, \quad \frac{d\vec{C}}{du}, \quad \frac{d^2U}{ds^2}, \quad \frac{dU}{ds}, \quad \frac{d^2s}{dt^2} \quad \text{and} \quad \frac{ds}{dt} \tag{3.35}$$

We already know how to handle the derivatives of $\vec{C}$ with respect to u. This leaves us with the remaining four derivatives

$$\frac{d^2U}{ds^2}, \quad \frac{dU}{ds}, \quad \frac{d^2s}{dt^2} \quad \text{and} \quad \frac{ds}{dt} \tag{3.36}$$

We already know that we can not solve the S and U functions analytically. Fortunately, it turns out that we can solve their derivatives analytically.

Let us start by equation (3.17). We can rewrite the integrand as the dot product of the derivative of the space spline that is

$$S(u) = \int_0^u \sqrt{\frac{d\vec{C}(u)}{du} \cdot \frac{d\vec{C}(u)}{du}} du \tag{3.37}$$

This is a bit more convenient to work with. Now, let us differentiate the arc length function. That is, we need to differentiate the integral as a function of its limits. By doing this we get the following expression.

$$\frac{dS(u)}{du} = \sqrt{\frac{d\vec{C}(u)}{du} \cdot \frac{d\vec{C}(u)}{du}} \tag{3.38}$$

If we look at the inverse mapping then we will get

$$\frac{dU(s)}{ds} = \frac{1}{\sqrt{\frac{d\vec{C}(U(s))}{du} \cdot \frac{d\vec{C}(U(s))}{du}}} \tag{3.39}$$

Recall that we already know the value of $U(s)$. This was computed during the arc length re-parameterization phase of the space spline and the gradient of the space spline was easily computed. All in all, we have an analytic expression for the first derivative of U with respect to s. Now, let us differentiate this derivative with respect to s.

$$\frac{d^2U(s)}{ds^2} = \frac{d}{ds}\left(\frac{1}{\sqrt{\frac{d\vec{C}(U(s))}{du} \cdot \frac{d\vec{C}(U(s))}{du}}}\right) \tag{3.40a}$$

$$= -\frac{\frac{d}{ds}\left(\frac{d\vec{C}(U(s))}{du} \cdot \frac{d\vec{C}(U(s))}{du}\right)}{2\left(\frac{d\vec{C}(U(s))}{du} \cdot \frac{d\vec{C}(U(s))}{du}\right)^{3/2}} \tag{3.40b}$$

$$= -\frac{\left(\frac{d^2\vec{C}(U(s))}{du^2} \cdot \frac{d\vec{C}(U(s))}{du} + \frac{d\vec{C}(U(s))}{du} \cdot \frac{d^2\vec{C}(U(s))}{du^2}\right)\frac{dU(s)}{ds}}{2\left(\frac{d\vec{C}(U(s))}{du} \cdot \frac{d\vec{C}(U(s))}{du}\right)^{3/2}} \tag{3.40c}$$

Cleaning up a bit we finally get

$$\frac{d^2U(s)}{ds^2} = -\frac{\frac{d\vec{C}(U(s))}{du} \cdot \frac{d^2\vec{C}(U(s))}{du^2}}{\left(\frac{d\vec{C}(U(s))}{du} \cdot \frac{d\vec{C}(U(s))}{du}\right)^2} \tag{3.41}$$

Again we notice that this is an analytic expression since we already know the value of $U(s)$. Now, let us turn our attention towards the computation of the derivatives of the arc length function with respect to t. From equation (3.8) we already know that.

$$\vec{V}(v) \mapsto (t(v), s(v)) \tag{3.42}$$

Since $\vec{V}(t)$ is a cubic spline so is each of its coordinate functions. This means that we can find an analytic expression for

$$\frac{dt(v)}{dv} \tag{3.43}$$

by straightforward computations. In fact, this would be a second order polynomial. Likewise, we can compute

$$\frac{ds(v)}{dv} \tag{3.44}$$

We also know that

$$\frac{ds(v(t))}{dt} = \frac{ds(v(t))}{dv}\frac{dv(t)}{dt} \tag{3.45}$$

Looking at the last term we see that this is in fact the inverse mapping of equation (3.43), so we end up having

$$\frac{ds(v(t))}{dt} = \frac{ds(v(t))}{dv}\frac{1}{\frac{dt(v(t))}{dv}} \tag{3.46}$$

The value of $v(t)$ is already known. It was found by the root search that we performed in the time re-parameterization phase of the space spline. Now let us look at the second derivative with respect to t. From equation (3.46) we get

$$\frac{d^2s(v(t))}{dt^2} = \frac{d^2s(v(t))}{dv^2}\frac{dv(t)}{dt}\frac{1}{\frac{dt(v(t))}{dv}} + \frac{ds(v(t))}{dv}\frac{-1}{\left(\frac{dt(v(t))}{dv}\right)^2}\frac{d}{dt}\left(\frac{dt(v(t))}{dv}\right) \tag{3.47a}$$

$$= \frac{\frac{d^2s(v(t))}{dv^2}}{\left(\frac{dt(v(t))}{dv}\right)^2} - \frac{\frac{ds(v(t))}{dv}}{\left(\frac{dt(v(t))}{dv}\right)^2}\frac{d^2t(v(t))}{dv^2}\frac{dv(t)}{dt} \tag{3.47b}$$

$$= \frac{\frac{d^2s(v(t))}{dv^2}}{\left(\frac{dt(v(t))}{dv}\right)^2} - \frac{\frac{ds(v(t))}{dv}\frac{d^2t(v(t))}{dv^2}}{\left(\frac{dt(v(t))}{dv}\right)^3} \tag{3.47c}$$

Again we see that we have an analytic expression. Looking at all our equations for the derivatives we see that we can get into trouble if we ever have

$$\frac{d\vec{C}(U(s))}{du} = 0 \quad \text{or} \quad \frac{dt(v(t))}{dv} = 0 \tag{3.48}$$

As it turns out, these degenerate cases do not cause any difficulties in practice and we have devoted the next section to discuss it.

3.3 Degenerate Cases

There are two ways to handle the degenerate cases. The first one is by repairing and the second one is by prevention.

If the first derivative of a spline vanishes, it will occur at a single parameter value [52]. That is, in any small neighborhood around this parameter value the first derivative would be non-zero. This suggests that we can remove the discontinuity by interpolating the velocities and accelerations of the scripted motion in a small neighborhood around the time value that caused the first derivative of the space spline (or the velocity spline for that matter) to become zero. Alternatively, one could also use a Taylor expansion around a nearby point in time to extrapolate the values of the scripted motion function.

This method will work fine as long as we are not having a cusp. Fortunately, in most cases cusps are easy to see with the naked eye and are therefore easily removed by remodeling the spline.

From an animators viewpoint, this may be very prohibitive if a cusp is what he really wants. One possible solution would be to model the motion by two separated space splines meeting at the cusp.

The second way of handling the degenerate cases implies that the splines are constructed in such a way that their first derivative never becomes zero. These kinds of splines are called regular splines [52].

This is a rather tedious approach. Luckily, we do have some pointers from spline theory which can help us to avoid most of the difficult cases. For instance, we could make sure that we do not have any knot values with a multiplicity greater than 1, or any succeeding control points that coincide or any sequence of three or more control points being collinear. This would definitely guarantee that the first derivative is always non-zero at any knot value. However, it does not eliminate problems in between the knot values – here cusps could occur. These cusps could be removed by either adjusting the control points or changing the interval size of the two knot values in question.

The drawback of this method is that it requires some assistance on behalf of an animator to iteratively manipulate the splines. An automated spline interpolation which guarantees that the first order derivative does not become zero would be preferable.

3.4 Testing

We have implemented a small framework for testing our theory. In this section we will outline our strategy and present our first results.

Four open nonuniform cubic B-splines were used in our test. Two three-dimensional space splines and two two-dimensional velocity splines. We did construct a simple and an advanced space spline, and the two velocity splines were a constant-velocity spline and an exaggerated ease-in-ease-out velocity spline. See Figure 3.6.

We have combined both space splines with both velocity splines and plotted the scripted motion position (red balls), velocity (green arrows) and acceleration (blue arrows) at equidistant time intervals. In the plots we have scaled the velocity and acceleration vectors to 1/12'th of their true magnitude to make it more easy for the viewer to verify the correctness of our visualizations.

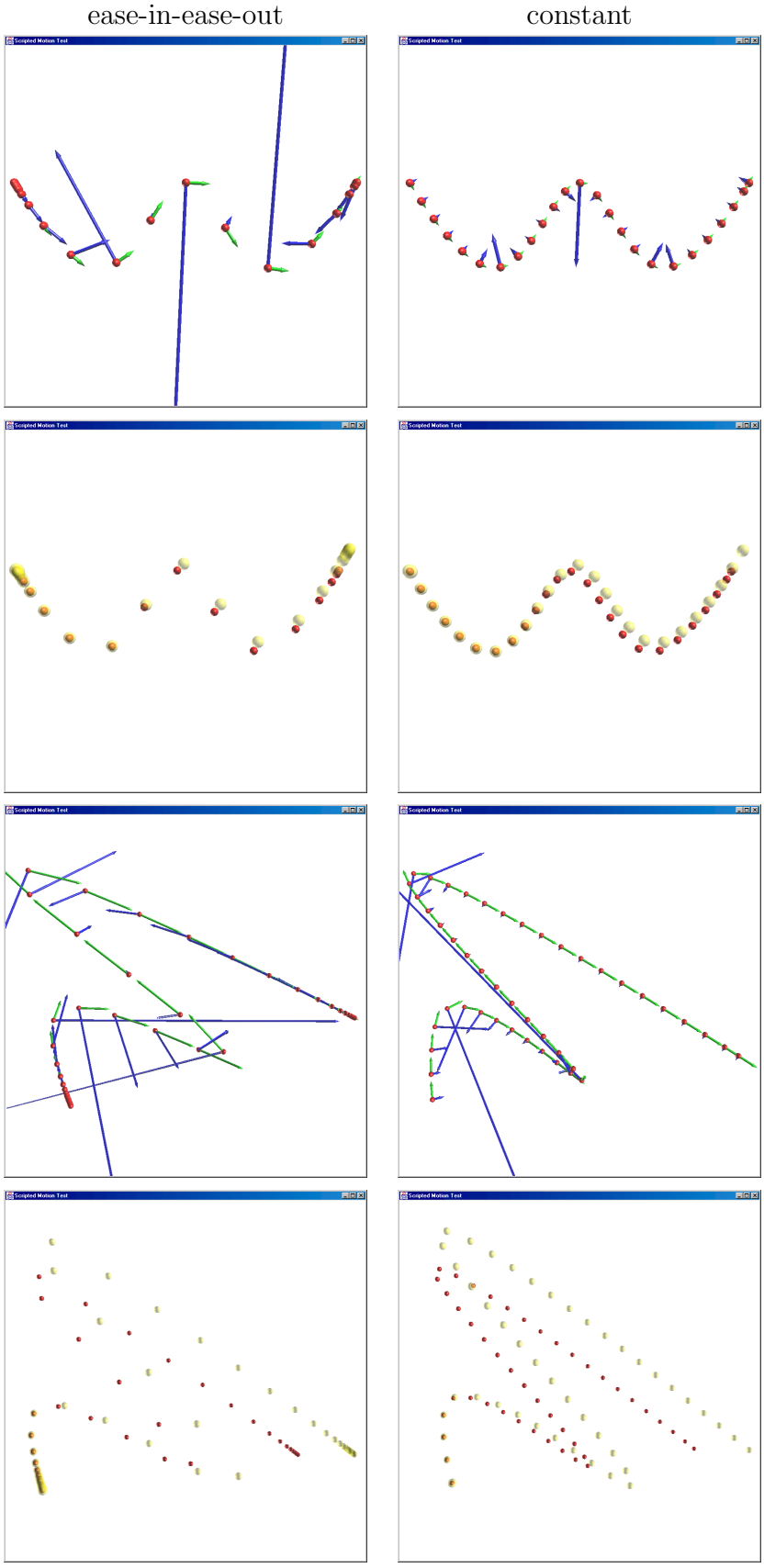


Figure 3.6: Numerical evidence of the correctness of a spline driven scripted motion.

It can be hard to see if the accelerations really are correct, because the blue arrows do tend to grow very rapidly. So in order to prove that our accelerations really are physically correct, i.e. they "produce" the motion we want, we came up with the following test strategy. We used the scripted motion to construct a coupled first order differential equation like

$$\frac{d}{dt}\begin{bmatrix} \vec{r}(t) \\ \vec{v}(t) \end{bmatrix} = \begin{bmatrix} \vec{v}(t) \\ \vec{a}(t) \end{bmatrix} \tag{3.49}$$

We then used a fourth order Runge Kutta ODE solver with adaptive step size to solve for the positions (yellow balls). The idea was to compare the positions computed by the ODE solver with the positions computed by the scripted motion function (red balls). If these positions are close to each other then in our opinion it would indicate that the accelerations are computed correctly.

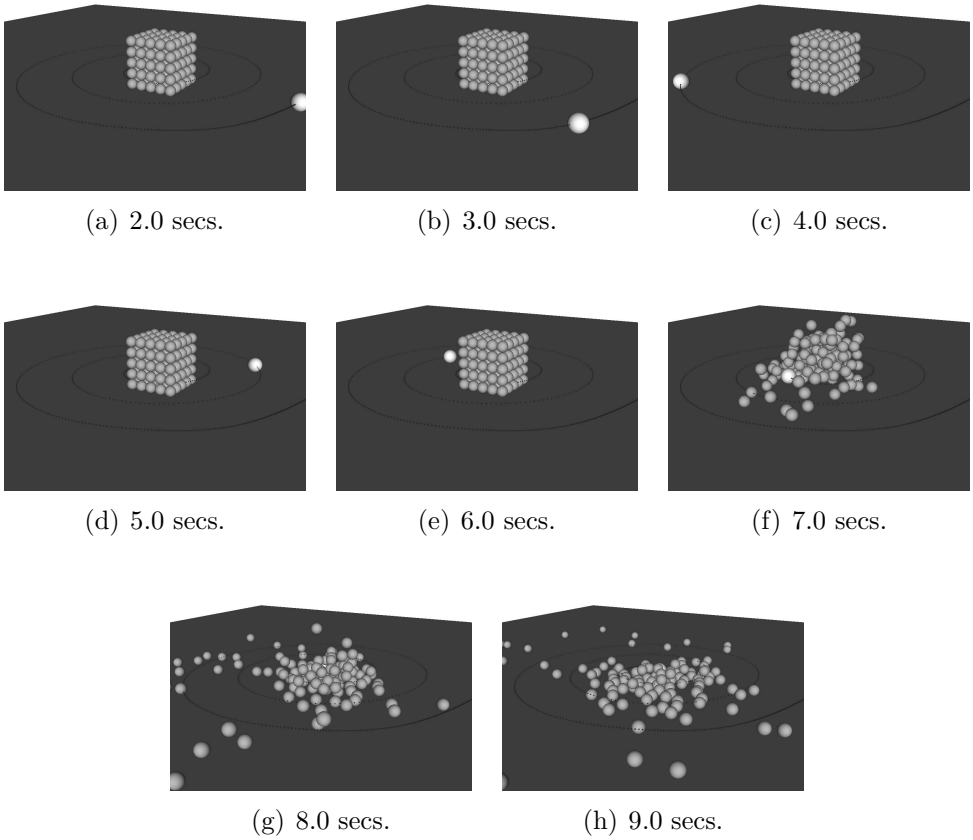
In our opinion, the results seen in Figure 3.6 show two problems. The first problem is best seen on those plots having constant velocity. Positions appear to be equidistant everywhere except at the endpoints of the space splines. Our investigations revealed that the problem is dependent on the choice of the numerical integrator which is used to solve the arc length integrand. The problem also depends on the required numerical accuracy in the "root search" re-parameterizations. We are therefore lead to believe that this problem is caused by numerical imprecision and numerical instability of the arc length integrand integration.

The second problem is much more obvious. If we compare the yellow balls to the red balls then it appears as though we keep the shape of the space spline, but we get somewhat off track. It is also seen that we get most off track when the space spline bends rapidly. If we look at the corresponding acceleration vectors at the places where we get the most off track, we see that at these places the vectors change rapidly both in direction and magnitude. This gives us the clue that the off track problem is due to the fact that the accelerations are forming a stiff differential equation. This actually makes perfect sense since any physical real world force which could cause the acceleration changes we see must be similar to that from a very stiff spring.

Our solution works but is influenced by numerical imprecision and instability, or at least our implementation is. The first problem we encountered is unpleasant, but can be dealt with by extending the space spline beyond the ending position one really wants. This ensures that the velocity spline will not overshoot the maximum traveling distance of the space spline.

The second problem is actually an advantage to us. If scripted bodies are used in a simulator and their scripted motion actually corresponds to stiff differential equations then it is much more tractable to have an analytical function computing the state of the scripted body instead of using an ODE solver.

Figures 3.7-3.10 show spline driven scripted motion used in a rigid body simulator. In all simulations friction was 0.25, restitution was 0.25, and simulation time step was 0.01 seconds. Black dots denote key frame positions along space splines at 0.01 seconds in between. The simulator from Chapter 6 was used to generate all the figures.



(a) 2.0 secs. (b) 3.0 secs. (c) 4.0 secs.

(d) 5.0 secs. (e) 6.0 secs. (f) 7.0 secs.

(g) 8.0 secs. (h) 9.0 secs.

Figure 3.7: A spline driven scripted object moves into 125 balls organized into a regular cube stack. Real world clock time per frame 0.01 seconds.

3.5 The Rotational Motion

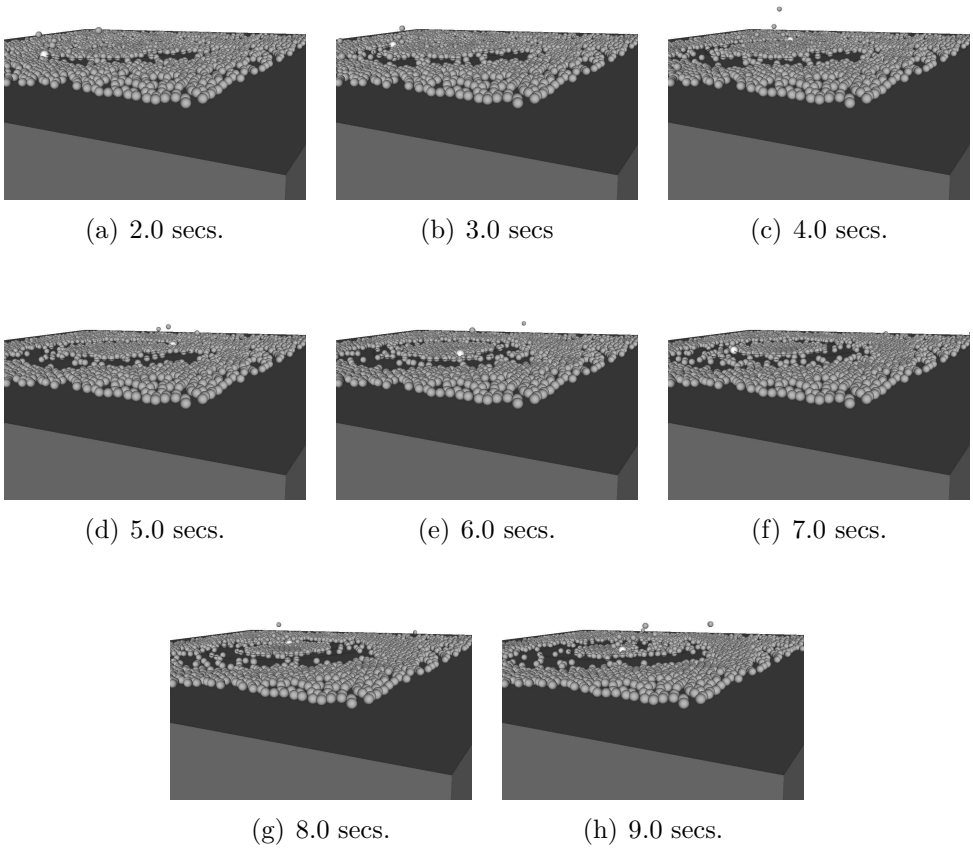
We have not implemented rotational motion, but, theoretically, the machinery can be extended from linear motion to rotational motion. For completeness, we shortly present the basic idea here. In the following, the orientation of a scripted body is given by the quaternion $q(t)$, and the corresponding rotation matrix is written as $R(q)$.

Let us assume that we can compute $q' \equiv \frac{dR}{dt}$ and $q'' \equiv \frac{d^2R}{dt^2}$ by using some rotational-spline representation. Our problem is to determine the physical quantities $\vec{\omega}$ and $\vec{\alpha}$, i.e. the angular velocity and angular acceleration. From physics, we know that if a body rotates with angular velocity $\vec{\omega}$ then its change of orientation would be

$$\frac{dq}{dt} = \frac{1}{2}\,[0, \vec{\omega}]\,q \tag{3.50}$$

From this, we derive

$$\frac{d^2q}{dt^2} = \frac{1}{2}\,[0, \vec{\alpha}]\,q + \frac{1}{2}\,[0, \vec{\omega}]\left(\frac{1}{2}\,[0, \vec{\omega}]\,q\right) \tag{3.51}$$



(a) 2.0 secs. (b) 3.0 secs (c) 4.0 secs.

(d) 5.0 secs. (e) 6.0 secs. (f) 7.0 secs.

(g) 8.0 secs. (h) 9.0 secs.

Figure 3.8: A spline driven scripted object moves inward in a spiral motion while interacting with 2000 Balls on a table. Real world clock time per frame 0.2-0.3 seconds.

That is

$$q' = \frac{1}{2}\left[0, \vec{\omega}\right] q \tag{3.52a}$$

$$q'' = \frac{1}{2}\left[0, \vec{\alpha}\right] q + \frac{1}{4}\left[0, \vec{\omega}\right]^2 q \tag{3.52b}$$

Since we know q, q', and q'' we can easily isolate $\vec{\omega}$ and $\vec{\alpha}$ as follows

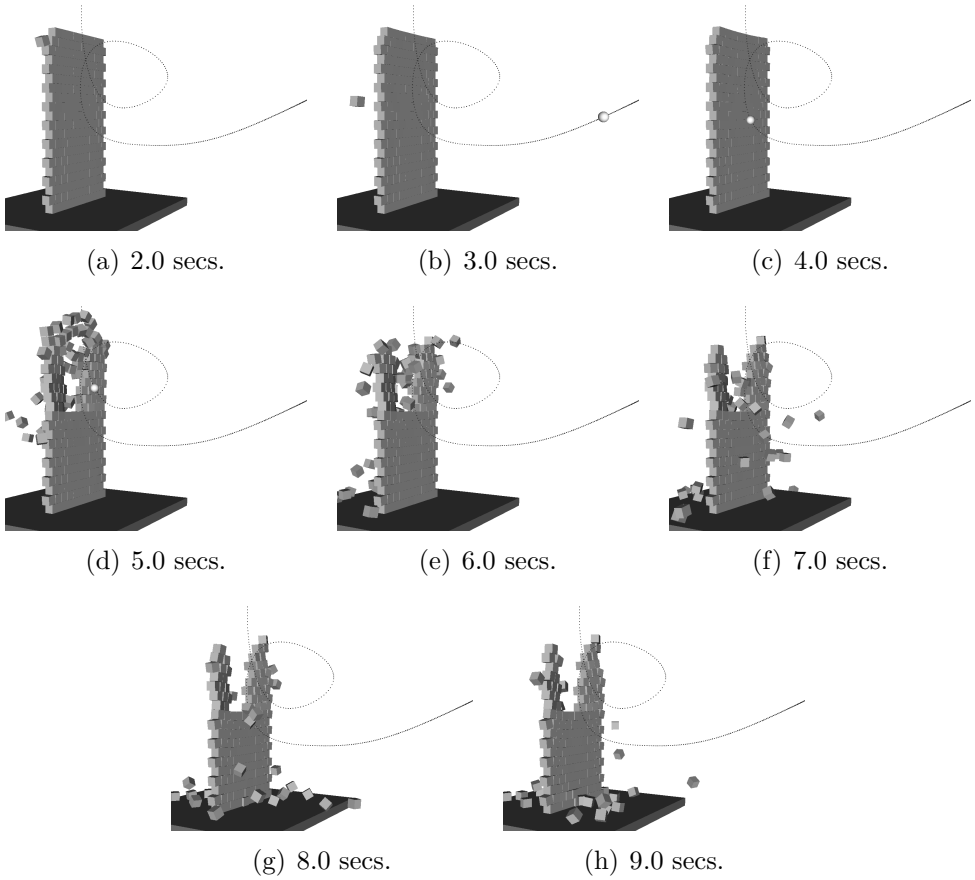
$$\left[0, \vec{\omega}\right] = 2q'q^* \tag{3.53a}$$

$$\left[0, \vec{\alpha}\right] = 2\left(q'' - \frac{1}{4}\left[0, \vec{\omega}\right] q'\right) q^* \tag{3.53b}$$

In other words, if we can compute up to the second derivative of the rotational "spline" motion, then we can compute the physical quantities we are looking for.

3.6 Discussion

In this chapter we have presented a method for computing the first and second derivatives of traditional spline driven motion in the time domain of the animation/simulation. We



(a) 2.0 secs. (b) 3.0 secs. (c) 4.0 secs.

(d) 5.0 secs. (e) 6.0 secs. (f) 7.0 secs.

(g) 8.0 secs. (h) 9.0 secs.

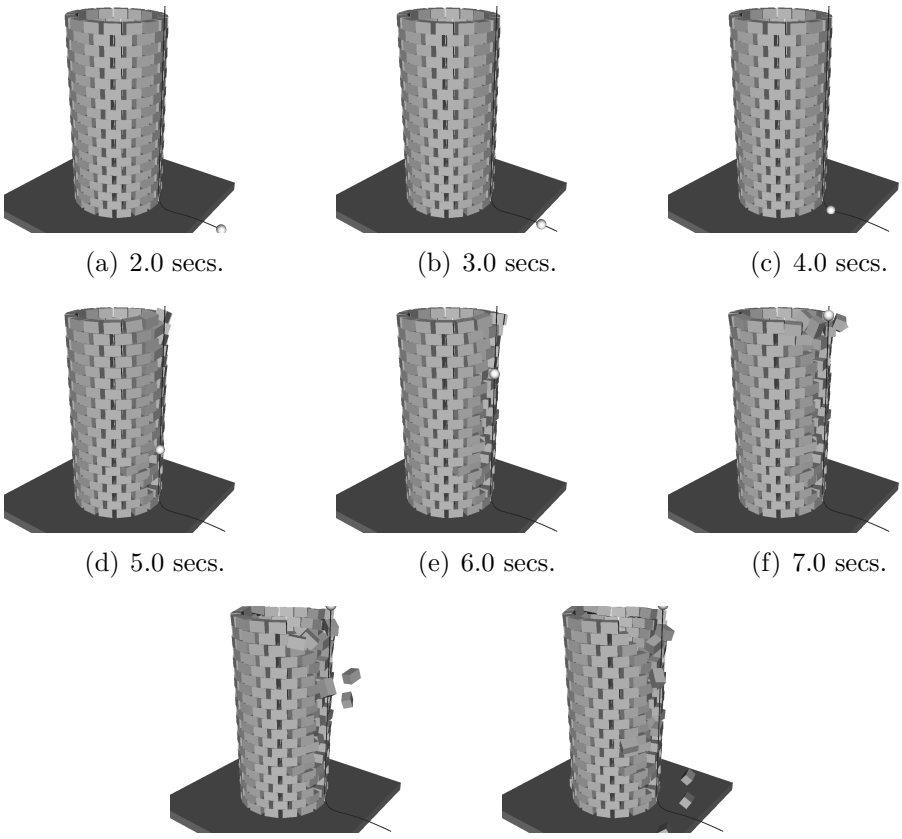
Figure 3.9: A spline driven scripted object is colliding with a 200 brick wall. Real world clock time per frame 0.02-0.03 seconds

have clearly shown how the method is applicable to handle the linear motion part of the scripted motion problem and we suggested how the method could be used to handle the rotational motion part as well.

The degenerate cases did not cause great problems in practice, but it sure would be nice to have a spline interpolation method which could guarantee that the first derivative never becomes zero. To our knowledge, no such interpolation method exists today.

The work we have presented in this chapter focuses on the linear motion part of the scripted motion problem. It appears to us that with very little effort the work could be extended to handle the rotational motion part as well, by, for instance, letting each of the Euler angles be described by a one dimensional space spline, or setting up a "center of interest" space spline and/or a "view-up" space spline. The latter cases do, however, pose some minor difficulties in the computation of the derivatives i.e. angular velocity and acceleration. We have presented ideas for handling these difficulties, as explained in Section 3.5.

However, we feel that it would be much more interesting to look at an interpolation method which uses quaternions, like a squad spline. Unfortunately, the squad spline can not be used. It has only C^1 continuity at its break points and we clearly require C^2 in order to be able to work with angular acceleration. To our knowledge, there does not exist a quaternion interpolation method with C^2 continuity everywhere.



(a) 2.0 secs. (b) 3.0 secs. (c) 4.0 secs.

(d) 5.0 secs. (e) 6.0 secs. (f) 7.0 secs.

(g) 8.0 secs. (h) 9.0 secs.

Figure 3.10: A spline driven scripted object interacts with a 320 brick tower. Real world clock time per frame 0.03-0.04 seconds.

Chapter 4

Constraint-Based Rigid Body Simulation

4.1 Introduction

Two formulations are mainly used for constraint based simulation: acceleration-based and velocity-based formulations. Although, formulations based on kinetic energy [96, 130] and motion space [124] exist.

In acceleration-based formulations, the exact contact force at a given time is found and then used in an ordinary differential equation describing the motion of the bodies in the scene. In a sense, an acceleration-based formulation sees the instantaneous picture of the configuration, while on the other hand a velocity-based formulation sees the effect of the dynamics over an entire time interval. Imagine that the true physical contact force, $\vec{f}_{\text{true}}(t)$, is known. The impulse $\vec{J}$ in the time interval Δt is then given as

$$\vec{J} = \int_0^{\Delta t} \vec{f}_{\text{true}}(t)dt. \tag{4.1}$$

and with Newton's second law of motion one can solve for the velocity, $\vec{v}^{\Delta t}$, as follows

$$\int_0^{\Delta t} m\frac{d\vec{v}}{dt}dt = \int_0^{\Delta t} \vec{f}_{\text{true}}(t) \tag{4.2}$$

$$m\left(\vec{v}^{\Delta t} - \vec{v}^{0}\right) = \vec{J}, \tag{4.3}$$

here, superscripts denote time, i.e. $\vec{v}^{\Delta t} = \vec{v}(\Delta t)$. A new position can now be found by integrating the velocity. The "force", $\vec{f}$, which we try to solve for in a velocity-based formulation can be interpreted as

$$\vec{J} = \Delta t\vec{f}, \tag{4.4}$$

which, numerically, will produce the same movement as if we had known the true contact force and computed the time integral. Since velocity-based formulations solve for impulses, they are also called impulse-based formulations, not to be mistaken with impulse-based simulation which is an entirely different simulation paradigm discussed in Chapter 2.

On the other hand, an acceleration-based formulation would try to compute the force, $\vec{f}_{\text{true}}(t)$. Therefore, acceleration-based formulations are often also termed force-based formulations. The force will be used to solve for the acceleration of the motion, which is then integrated once to yield velocities and integrated twice to yield a new position.

Acceleration-based formulations cannot handle collisions and one must stop at the point of collision and switch to an impulse-momentum law [11, 8, 117, 35]. Furthermore, acceleration-based formulations suffer from indeterminacy and inconsistency [13, 138]. The velocity-based formulations suffer from none of these drawbacks. Another advantage of the velocity-based formulation is that it does not suffer from the small time-step problem to the same extent as the acceleration-based formulation, meaning that larger time-steps can be taken during the simulation. The small time-step problem is described by Milenkovic and Schmidl [96, 130].

Velocity-based formulations in constraint-based methods are widely popular and used, e.g. Open Dynamics Engine [112], Karma from MathEngine [82], and Vortex from Critical Mass Labs [150]. In the following, we will present the classical velocity-based constraint formulation [140, 141], give a possible object oriented implementation design, and discuss various practical issues.

Many papers and books written on velocity-based formulations use a rather high and abstract level of mathematical notation together with a great amount of "long forgotten" analytical mechanics. There is a widespread notation and many small variations. In the following, we will follow the work presented in [4, 129, 141, 112].

4.2 Previous Work

Impulse-based simulation was introduced by Hahn [71], where time integrals of contact forces were used to model the interactions. In recent years, Mirtich [100] worked with impulse-based simulation using a slightly different approach, where contact forces were modeled as collision impulse trains. Hahn's and Mirtich's work represent two different ways of thinking of impulse-based simulation: as a time integral and as a sum of delta functions approximating the corresponding time integral. Mirtich's work led to a new simulation paradigm. Hahn's work could be interpreted as an early predecessor to the velocity-based formulation.

Stewart and Trinkle [140, 141, 138] made a new impulse-based method. Their method later became known as "Stewart's Method". Like Hahn's method, Stewart's method also computes the time integrals of the contact forces. It has since inspired many: Anitescu and Potra [4], which extended the method to guarantee solution existence, Sauer and Schömer [129] extended it with a linearized contact condition, Song, Pang, and Kumar [135] used a semi-implicit time-stepping model for frictional compliant contact, and most recently, an implicit time-stepping method for stiff multi-body dynamics by Anitescu and Potra [5], and Hart and Anitescu introduced a constraint stabilization [72] method.

Even though Stewart's method is an impulse-based method, it looks and feels like a typical constraint-based method, such as the one formulated by Baraff [16, 18], Trinkle, Pang, Sudarsky and Lo [145], Trinkle, Tzitzoutis and Pang [146], and Pfeiffer and Glocker [117]. These are termed acceleration-based formulations.

Stewart and Trinkle originally used position constraints in their formulation. These position constraints ensure non-penetration, but suffer from existence problems unless all contact normals are linearly independent.

Anitescu and Potra's velocity-based formulation always guarantees a solution, but

there is one drawback: penetrations might occur when the system moves along concave boundaries of the admissible region in configuration space. Another side effect from the velocity-based formulation is that separating contacts moving towards each other require special attention. If these are present they will be treated as if they were in colliding contact. This could leave objects hanging in the air. Anitescu and Potra do propose a time-stepping algorithm that correctly handles this side effect, but other authors tend to forget the problem.

Stewart and Trinkle suggest that simple projection can be used to eliminate this problem. However, care must be taken to avoid losing energy in the process. However, it is not mentioned how one should do the projection.

Stewart and Trinkle [141] suggest using contact tracking (it could be coined "retroactive detection" as well) to avoid the problem of explicitly determining all potential active contact constraints. That is, once a solution is found, the constraints at time $t + \Delta t$ is checked and if any one is found to be violated, they are added to the set of active contact constraints and the problem is re-solved. Of course, this increases the amount of computations. Often, this is not desirable. For instance, in the Open Dynamics Engine, a fixed time-stepping method is chosen and any errors that occurr are reduced by using penalty forces. The strategy is to not to try to prevent errors, but to fix or reduce them if they occur. Another issue which is not obvious, at least to us, is if it is possible at all to use a retroactive detection of contact constraints when we know there is a problem with concave boundaries of the admissible region of configuration space.

Sauer and Schömer [129] use a linearized contact condition. They claim this is for potential contact constraints and they require the assumption that during their "fix-point-iteration", contact constraints do not change topologically, i.e. $(E, E) \mapsto (V, E)$ is not allowed. However, it is not obvious to us why this is required.

In the summary section of Stewart and Trinkle's paper [141] there are some thoughts on how their method can be extended to handle a non-zero coefficient of restitution. They suggest stopping at collision times, which in our opinion is not an attractive choice. Actually, their thoughts are based on the work by Anitescu and Potra. In the Open Dynamics Engine a slightly different approach is used similar to [11].

4.3 Equations of Motion

From classical mechanics we have the Newton-Euler equations describing the motion for all bodies. For the i'th body, the mass of body i is given by m_i, the inertia tensor by I_i, the position of the center of mass by $\vec{r}_i$, and the velocity of the center of mass as $\vec{v}_i$. The orientation is represented by the quaternion q_i and the angular velocity by $\vec{\omega}_i$. The Newton-Euler equations for the i'th body look like this (summations are taken over all

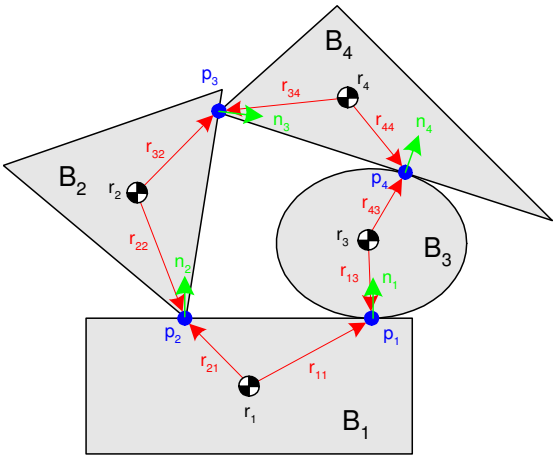


Figure 4.1: Illustration of the convention and notation of the contact normals.

contact points):

$$\dot{\vec{r}}_i = \vec{v}_i \tag{4.5a}$$

$$\dot{q}_i = \tfrac{1}{2}\vec{\omega}_i q_i \tag{4.5b}$$

$$\dot{\vec{v}}_i = m_i^{-1} \sum_{j_k=i} \vec{f}_k - m_i^{-1} \sum_{i_k=i} \vec{f}_k + m_i^{-1} \vec{f}_i^{\text{ ext}} \tag{4.5c}$$

$$\dot{\vec{\omega}}_i = I_i^{-1} \sum_{j_k=i} \vec{r}_{kj} \times \vec{f}_k - I_i^{-1} \sum_{i_k=i} \vec{r}_{ki} \times \vec{f}_k \tag{4.5d}$$

$$- I_i^{-1} \vec{\omega}_i \times I_i \vec{\omega}_i + I_i^{-1} \vec{\tau}_i^{\text{ ext}} \tag{4.5e}$$

The dot-notation means the time derivative $\frac{d}{dt}$, and is used to ease readability. Observe that $\vec{f}_k$ denotes the contact force at the k'th contact. For the time being, we will ignore joints and motors. The effect of all external forces on the center of mass is given by $\vec{f}_i^{\text{ ext}}$ and the total torque from external forces is given by $\vec{\tau}_i^{\text{ ext}}$.

For notational convenience we introduce a contact table. Consider a total of K contacts, and assign an unique number k to each contact. For each contact, we know the indices i_k and j_k of the two incident bodies. We use the convention that $i_k < j_k$. We also have a contact normal $\vec{n}_k$ and a contact point $\vec{p}_k$, both specified in the world coordinate system, and with the convention that the contact normal points from the body with the smallest index to the body with the largest index. This is illustrated in Figure 4.1. Notice that we can never have $i_k = j_k$. For each contact we can compute a vector from the center of mass, $\vec{r}_i$, of an incident body with index i, to the point of contact $\vec{p}_k$ that is

$$\vec{r}_{ki} = \vec{p}_k - \vec{r}_i \tag{4.6}$$

The Newton-Euler equations can now be written as

$$\dot{\vec{s}} = \mathbf{S}\vec{u} \tag{4.7a}$$

$$\dot{\vec{u}} = \mathbf{M}^{-1} \left(\mathbf{C}\mathbf{N}\vec{f} + \vec{f}_{ext} \right). \tag{4.7b}$$

Where we have introduced some matrix notation which we will describe in the following. The position and orientation of n bodies may be concatenated into a single generalized position and orientation vector, $\vec{s} \in \mathbb{R}^{7n}$:

$$\vec{s} = [\vec{r}_1, q_1, \vec{r}_2, q_2, \cdots, \vec{r}_n, q_n]^T . \tag{4.8}$$

Similarly, we can write the generalized velocity vector $\vec{u} \in \mathbb{R}^{6n}$ as

$$\vec{u} = [\vec{v}_1, \vec{\omega}_1, \vec{v}_2, \vec{\omega}_2, \cdots, \vec{v}_n, \vec{\omega}_n]^T . \tag{4.9}$$

For the time being, the frictional effects will be ignored, implying that the contact force can be written as

$$\vec{f_k} = f_k \vec{n}_k. \tag{4.10}$$

This means that we only need to remember the magnitude, f_k, of the normal force, and these can now be concatenated into a single vector $\vec{f} \in \mathbb{R}^K$

$$\vec{f} = [f_1, f_2, \cdots, f_K]^T . \tag{4.11}$$

The external forces, torques, and velocity dependent forces can also be concatenated into a vector, $\vec{f}_{\text{ext}} \in \mathbb{R}^{6n}$,

$$\vec{f}_{ext} = \left[\vec{f}_1^{\text{ ext}}, \vec{\tau}_1^{\text{ ext}} - \vec{\omega}_1 \times I_1 \vec{\omega}_1, \cdots, \vec{f}_n^{\text{ ext}}, \vec{\tau}_n^{\text{ ext}} - \vec{\omega}_n \times I_n \vec{\omega}_n\right]^T . \tag{4.12}$$

Given $q_i = [s_i, x_i, y_i, z_i]^T \in \mathbb{R}^4$, we can write the rotation as a matrix $\mathbf{Q}_i \in \mathbb{R}^{4\times 3}$ as:

$$\mathbf{Q}_i = \frac{1}{2} \begin{bmatrix} -x_i & -y_i & -z_i \\ s_i & z_i & -y_i \\ -z_i & s_i & x_i \\ y_i & -x_i & s_i \end{bmatrix} . \tag{4.13}$$

where $\frac{1}{2}\vec{\omega}_i q_i = \mathbf{Q}_i \vec{\omega}_i$. The rotations can now be concatenated into a matrix $\mathbf{S} \in \mathbb{R}^{7n\times 6n}$,

$$\mathbf{S} = \begin{bmatrix} \mathbf{1} & & & & \mathbf{0} \\ & \mathbf{Q}_1 & & & \\ & & \ddots & & \\ & & & \mathbf{1} & \\ \mathbf{0} & & & & \mathbf{Q}_n \end{bmatrix} , \tag{4.14}$$

Matrix $\mathbf{S}$ is also illustrated in Figure 4.2. The generalized mass matrix $\mathbf{M} \in \mathbb{R}^{6n\times 6n}$ is,

$$\mathbf{M} = \begin{bmatrix} m_1\mathbf{1} & & & & \mathbf{0} \\ & \mathbf{I}_1 & & & \\ & & \vdots & & \\ & & & m_n\mathbf{1} & \\ \mathbf{0} & & & & \mathbf{I}_n \end{bmatrix} \tag{4.15}$$

where $\mathbf{1}$ is the identity matrix. The layout of the mass matrix is illustrated in Figure 4.3. The matrix of contact normals $\mathbf{N} \in \mathbb{R}^{3K \times K}$ is,

$$\mathbf{N} = \begin{bmatrix} \vec{n}_1 & & & \mathbf{0} \\ & \vec{n}_2 & & \\ & & \ddots & \\ \mathbf{0} & & & \vec{n}_k \end{bmatrix}, \tag{4.16}$$

as shown in Figure 4.4, and the matrix of contact conditions $\mathbf{C} \in \mathbb{R}^{6n \times 3K}$ is,

$$\mathbf{C}_{lk} = \begin{cases} -\mathbf{1} & \text{for } l = 2i_k - 1 \\ -\mathbf{r}^{\times}_{ki_k} & \text{for } l = 2i_k \\ \mathbf{1} & \text{for } l = 2j_k - 1 \\ \mathbf{r}^{\times}_{kj_k} & \text{for } l = 2j_k \\ \mathbf{0} & \text{otherwise} \end{cases}. \tag{4.17}$$

Here $\mathbf{r}^{\times} \in \mathbb{R}^{3\times 3}$ is the skew-symmetric matrix given by

$$\mathbf{r}^{\times} = \begin{bmatrix} 0 & -r_3 & r_2 \\ r_3 & 0 & -r_1 \\ -r_2 & r_1 & 0 \end{bmatrix}. \tag{4.18}$$

It is easy to show that $\mathbf{r}^{\times}\vec{a} = \vec{r} \times \vec{a}$. Every column of $\mathbf{C}$ corresponds to a single contact and every row to a single body see Figure 4.5.

Using an Euler-scheme, we can write the discretized equations of motion as follows,

$$\vec{s}^{t+\Delta t} = \vec{s}^{t} + \Delta t \mathbf{S} \vec{u}^{t+\Delta t}, \tag{4.19a}$$

$$\vec{u}^{t+\Delta t} = \vec{u}^{t} + \Delta t \mathbf{M}^{-1} \left(\mathbf{C}\mathbf{N}\vec{f}^{t+\Delta t} + \vec{f}_{ext} \right). \tag{4.19b}$$

Here, superscripts denote the time at which a quantity is computed. Notice that the matrices depend on time. If they are evaluated at time t, then we have a semi-implicit method, and if they are evaluated at time $t + \Delta t$ we have an implicit method.

Equation (4.19a) is called the position update. It is a good idea to renormalize the quaternions stored in $\vec{s}^{t+\Delta t}$. A modified position update is outlined in Section 4.14. Equation (4.19b) is called the velocity update.

4.4 The Contact Condition

The projection matrix, $\mathbf{P}_k \in \mathbb{R}^{3K\times 3}$ will be needed for further analysis of the k'th contact point. It is defined as

$$\mathbf{P}_k^T = \left[\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \ldots, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \ldots, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \right]. \tag{4.20}$$

That is, the k'th 3×3 sub matrix is set to the identity matrix. The normal component of the relative contact velocity of the k'th contact point is given by

$$\vec{n}_k^T \mathbf{P}_k^T \mathbf{C}^T \vec{u} = \vec{n}_k^T \left(\vec{v}_{j_k} + \vec{\omega}_{j_k} \times \vec{r}_{kj_k} \right) - \vec{n}_k^T \left(\vec{v}_{i_k} + \vec{\omega}_{i_k} \times \vec{r}_{ki_k} \right). \tag{4.21}$$

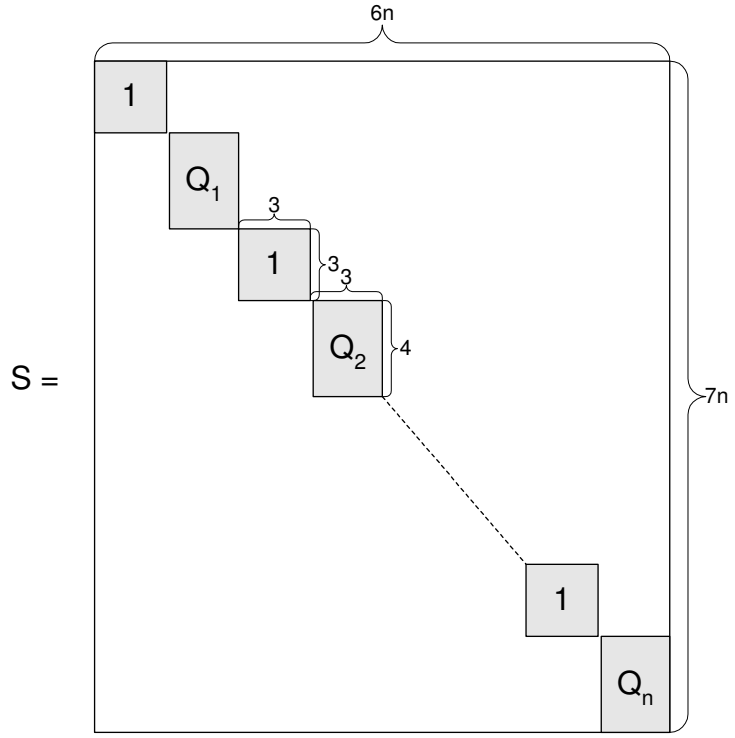


Figure 4.2: The **S** matrix layout.

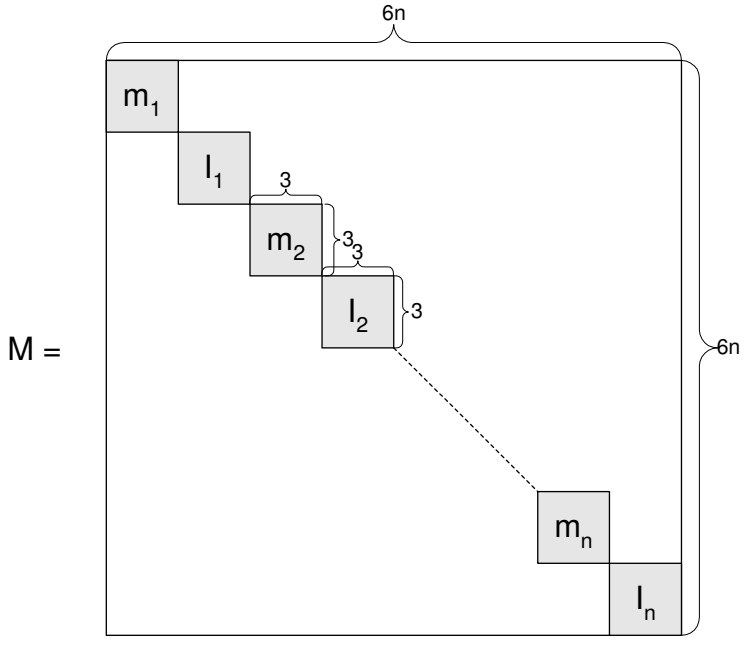


Figure 4.3: The **M** matrix layout.

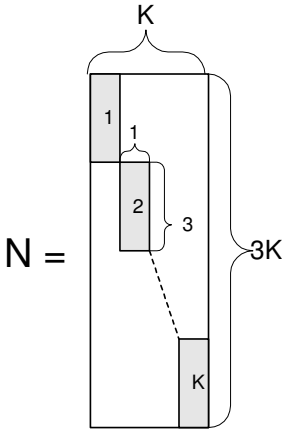


Figure 4.4: The **N** matrix layout.

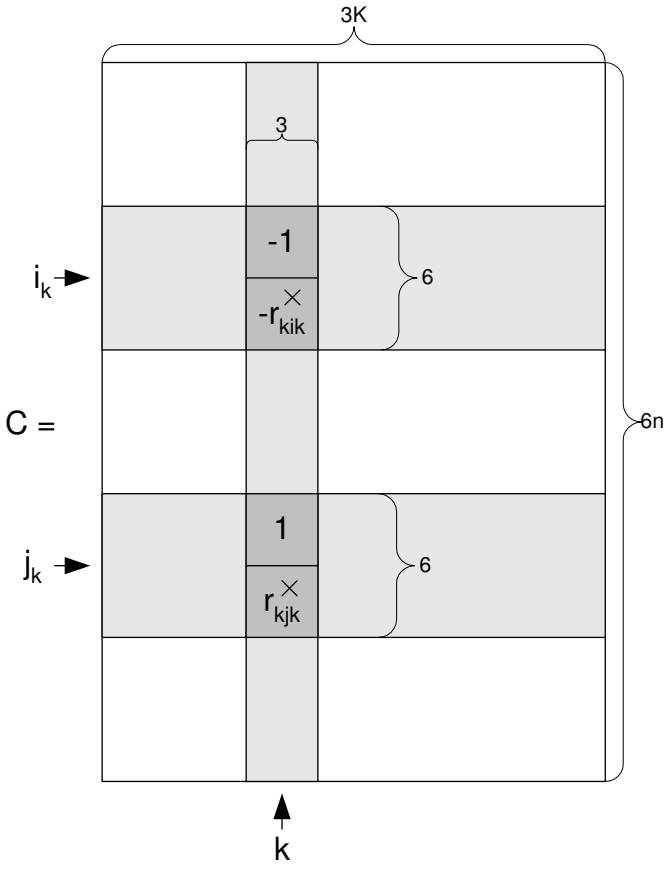


Figure 4.5: The **C** matrix layout.

Notice that multiplying by the projection matrix will mask out the k'th contact conditions. If body B_{i_k} and B_{j_k} touch at contact point $\vec{p}_k$ at time t, then the complementarity condition for the velocities must hold,

$$\vec{n}_k^T \mathbf{P}_k^T \mathbf{C}^T \vec{u}^{t+\Delta t} \geq 0 \quad \text{compl. to} \quad f_k \geq 0. \tag{4.22}$$

A complementarity condition means that if we have two conditions, then one is non-zero and the other is zero or vice versa. Stewart and Trinkle [140] originally used a complementarity condition on position. Anitescu and Potra [4] discovered that using the velocity complementarity problem formulation guarantees solution existence. Sauer and Schömer [129] expanded the velocity formulation further by handling future potential contact points.

If there is no contact at the potential contact point $\vec{p}_k$ at time t, the following linearized complementarity condition holds,

$$\vec{n}_k^T \mathbf{P}_k^T \mathbf{C}^T \vec{u}^{t+\Delta t} \geq \frac{\nu_k}{\Delta t} \quad \text{compl. to} \quad f_k \geq 0. \tag{4.23}$$

Later, we will go into details on the linearization. If we use $\nu_k = 0$ for all touching contacts then we can formulate a complementarity condition for all contacts, both touching and potential ones as

$$\mathbf{N}^T \mathbf{C}^T \vec{u}^{t+\Delta t} \geq \frac{\vec{\nu}}{\Delta t} \quad \text{compl. to} \quad \vec{f} \geq 0, \tag{4.24}$$

with $\vec{\nu} = [\nu_1, \ldots, \nu_K]^T \in \mathbb{R}^K$. Inserting (4.19b) into (4.24) gives

$$\mathbf{N}^T \mathbf{C}^T \left(\vec{u}^t + \Delta t \mathbf{M}^{-1} \left(\mathbf{C}\mathbf{N} \vec{f}^{t+\Delta t} + \vec{f}_{ext} \right) \right) - \frac{\vec{\nu}}{\Delta t} \geq 0. \tag{4.25}$$

Rearranging yields,

$$\underbrace{\mathbf{N}^T \mathbf{C}^T \mathbf{M}^{-1} \mathbf{C}\mathbf{N}}_{\mathbf{A}} \underbrace{\Delta t \vec{f}^{t+\Delta t}}_{\vec{x}} + \underbrace{\mathbf{N}^T \mathbf{C}^T \left(\vec{u}^t + \Delta t \mathbf{M}^{-1} \vec{f}_{ext} \right) - \frac{\vec{\nu}}{\Delta t}}_{\vec{b}} \geq 0, \tag{4.26}$$

which results in a linear complementarity problem (LCP) of the form,

$$\mathbf{A}\vec{x} + \vec{b} \geq 0 \quad \text{compl. to} \quad \vec{x} \geq 0, \tag{4.27}$$

where $\mathbf{A} \in \mathbb{R}^{K \times K}$ and $\vec{x}, \vec{b} \in \mathbb{R}^K$. Entry l, k of $\mathbf{A}$ looks like

$$\begin{aligned}
\mathbf{A}_{lk} = {} & \delta_{i_l i_k} \vec{n}_l^T \left(\frac{1}{m_{i_k}} \mathbf{1} - \mathbf{r}_{li_l}^{\times} \mathbf{I}_{i_k}^{-1} \mathbf{r}_{ki_k}^{\times} \right) \vec{n}_k \\
& - \delta_{i_l j_k} \vec{n}_l^T \left(\frac{1}{m_{j_k}} \mathbf{1} - \mathbf{r}_{li_l}^{\times} \mathbf{I}_{j_k}^{-1} \mathbf{r}_{kj_k}^{\times} \right) \vec{n}_k \\
& - \delta_{j_l i_k} \vec{n}_l^T \left(\frac{1}{m_{i_k}} \mathbf{1} - \mathbf{r}_{lj_l}^{\times} \mathbf{I}_{i_k}^{-1} \mathbf{r}_{ki_k}^{\times} \right) \vec{n}_k \\
& + \delta_{j_l j_k} \vec{n}_l^T \left(\frac{1}{m_{j_k}} \mathbf{1} - \mathbf{r}_{lj_l}^{\times} \mathbf{I}_{j_k}^{-1} \mathbf{r}_{kj_k}^{\times} \right) \vec{n}_k,
\end{aligned} \tag{4.28}$$

with the Kronecker-symbol being,

$$\delta_{ij} = \begin{cases} 1 & \text{for } i = j, \\ 0 & \text{for } i \neq j. \end{cases} \tag{4.29}$$

4.5 Linearization

Sauer and Schömer [129] use a linearized contact condition in equation (4.24). In the following, we will derive the linearized contact condition. The linearization serves as a measure of when a potential contact constraint should be switched on. This allows bigger time-steps to be taken while keeping the error low. Taking the same time-step size without the linearization will imply a larger approximation error,

The k'th potential contact point may be represented by the closest points $\vec{p}_{i_k}$ and $\vec{p}_{j_k}$ between two bodies, B_{i_k} and B_{j_k}, that eventually meet and form the k'th contact point. The closest points depend on the position and orientation of the bodies. If we let the vector $\vec{s}_k \in \mathbb{R}^{14}$ be the generalized position vector of the two bodies, where the function $\vec{s}_k$'s dependency of time has been omitted for readability,

$$\vec{s}_k = [\vec{r}_{i_k}, q_{i_k}, \vec{r}_{j_k}, q_{j_k}], \tag{4.30}$$

the minimal distance between the two bodies, $d_k(\vec{s}_k)$, is

$$d_k(\vec{s}_k) = \vec{n}_k^T(\vec{s}_k)\,(\vec{p}_{j_k} - \vec{p}_{i_k}) \geq 0, \tag{4.31}$$

where $\vec{n}_k$ is a unit vector pointing from $\vec{p}_{i_k}$ to $\vec{p}_{j_k}$. A first order Taylor-expansion of $d_k(\vec{s}_k)$ at $\vec{s}_k{}'$ is,

$$d_k(\vec{s}_k) = d_k(\vec{s}_k{}') + (\nabla_{\vec{s}_k} d_k(\vec{s}_k{}'))^T\,(\vec{s}_k - \vec{s}_k{}') + O(\Delta t^2). \tag{4.32}$$

Notice that $\nabla_{\vec{s}_k}$ is the functional derivative. If we look at the backward difference of the time derivative of the generalized position vector we find,

$$\frac{d}{dt}\left(\vec{s}_k{}^{\,t+\Delta t}\right) = \frac{\vec{s}_k{}^{\,t+\Delta t} - \vec{s}_k{}^{\,t}}{\Delta t} + O(\Delta t). \tag{4.33}$$

Rearranging yields,

$$\vec{s}_k{}^{\,t+\Delta t} = \vec{s}_k{}^{\,t} + \frac{d}{dt}\left(\vec{s}_k{}^{\,t+\Delta t}\right)\Delta t + O(\Delta t^2) \tag{4.34}$$

Again we approximate $\nabla_{\vec{s}_k} d_k(\vec{s}_k{}')$ at $\vec{s}_k{}^{\,t+\Delta t}$ using Taylor's theorem by taking the zeroth order expansion to get,

$$\nabla_{\vec{s}_k} d_k(\vec{s}_k{}') = \nabla_{\vec{s}_k} d_k(\vec{s}_k{}^{\,t+\Delta t}) + O(\Delta t). \tag{4.35}$$

Substituting (4.34) for $\vec{s}_k$ in (4.32) gives,

$$\begin{aligned} d_k(\vec{s}_k{}^{\,t+\Delta t}) &\approx d_k(\vec{s}_k{}') + (\nabla_{\vec{s}_k} d_k(\vec{s}_k{}'))^T \left(\vec{s}_k{}^{\,t} + \frac{d}{dt}\left(\vec{s}_k{}^{\,t+\Delta t}\right)\Delta t - \vec{s}_k{}'\right) && (4.36)\\ &= d_k(\vec{s}_k{}') + (\nabla_{\vec{s}_k} d_k(\vec{s}_k{}'))^T \left(\vec{s}_k{}^{\,t} - \vec{s}_k{}'\right) \\ &\quad + \Delta t\,(\nabla_{\vec{s}_k} d_k(\vec{s}_k{}'))^T \frac{d}{dt}\left(\vec{s}_k{}^{\,t+\Delta t}\right). && (4.37) \end{aligned}$$

Now we insert (4.35) in the last term and get

$$\begin{aligned} d_k(\vec{s}_k{}^{\,t+\Delta t}) \approx\; & d_k(\vec{s}_k{}') + (\nabla_{\vec{s}_k} d_k(\vec{s}_k{}'))^T \left(\vec{s}_k{}^{\,t} - \vec{s}_k{}'\right) \\ & + \Delta t \left(\nabla_{\vec{s}_k} d_k(\vec{s}_k{}^{\,t+\Delta t})\right)^T \frac{d}{dt}\left(\vec{s}_k{}^{\,t+\Delta t}\right). \end{aligned} \tag{4.38}$$

Recall that the distance function is actually a function of time, $d_k(\vec{s}_k(t))$, so by the chain rule we have

$$\frac{d}{dt}\left(d_k(\vec{s}_k{}^{t+\Delta t})\right) = \left(\nabla d_k(\vec{s}_k{}^{t+\Delta t})\right)^T \frac{d}{dt}\left(\vec{s}_k{}^{t+\Delta t}\right). \tag{4.39}$$

Inserting (4.39) into (4.38) yields

$$\begin{aligned} d_k(\vec{s}_k{}^{t+\Delta t}) \approx\ & d_k(\vec{s}_k{}') + \left(\nabla_{\vec{s}_k} d_k(\vec{s}_k{}')\right)^T \left(\vec{s}_k{}^t - \vec{s}_k{}'\right) \\ & + \Delta t \frac{d}{dt}\left(d_k(\vec{s}_k{}^{t+\Delta t})\right). \end{aligned} \tag{4.40}$$

From past work such as [18, 16] we know,

$$\frac{d}{dt}\left(d_k(t)\right) = \vec{n}_k^T\left((\vec{v}_{j_k} + \vec{\omega}_{j_k} \times \vec{r}_{j_k}) - (\vec{v}_{i_k} + \vec{\omega}_{i_k} \times \vec{r}_{i_k})\right). \tag{4.41}$$

Using this in (4.40) together with (4.31) we derive

$$\begin{aligned} & d_k(\vec{s}_k{}') + \left(\nabla_{\vec{s}_k} d_k(\vec{s}_k{}')\right)^T \left(\vec{s}_k{}^t - \vec{s}_k{}'\right) \\ & \quad + \Delta t \vec{n}_k^T \left(\left(\vec{v}_{j_k}{}^{t+\Delta t} + \vec{\omega}_{j_k}{}^{t+\Delta t} \times \vec{r}_{j_k}{}^{t+\Delta t}\right)\right. \\ & \quad \left. - \left(\vec{v}_{i_k}{}^{t+\Delta t} + \vec{\omega}_{i_k}{}^{t+\Delta t} \times \vec{r}_{i_k}{}^{t+\Delta t}\right)\right) \geq 0. \end{aligned} \tag{4.42}$$

Rearranging we have,

$$\begin{aligned} & \vec{n}_k^T \left(\left(\vec{v}_{j_k}{}^{t+\Delta t} + \vec{\omega}_{j_k}{}^{t+\Delta t} \times \vec{r}_{j_k}{}^{t+\Delta t}\right) - \left(\vec{v}_{i_k}{}^{t+\Delta t} + \vec{\omega}_{i_k}{}^{t+\Delta t} \times \vec{r}_{i_k}{}^{t+\Delta t}\right)\right) \\ & \quad \geq -\frac{1}{\Delta t}\left(d_k(\vec{s}_k{}') + \left(\nabla_{\vec{s}_k} d_k(\vec{s}_k{}')\right)^T \left(\vec{s}_k{}^t - \vec{s}_k{}'\right)\right). \end{aligned} \tag{4.43}$$

Recall that the left side of this equation is in fact $\vec{n}_k^T \mathbf{P}_k^T \mathbf{C}^T \vec{u}^{\ t+\Delta t}$. It now follows that

$$\vec{n}_k^T \mathbf{P}_k^T \mathbf{C}^T \vec{u}^{\ t+\Delta t} \geq -\frac{1}{\Delta t}\left(d_k(\vec{s}_k{}') + \left(\nabla_{\vec{s}_k} d_k(\vec{s}_k{}')\right)^T \left(\vec{s}_k{}^t - \vec{s}_k{}'\right)\right). \tag{4.44}$$

Comparing with (4.23) we write,

$$\nu_k = -\left(d_k(\vec{s}_k{}') + \left(\nabla_{\vec{s}_k} d_k(\vec{s}_k{}')\right)^T \left(\vec{s}_k{}^t - \vec{s}_k{}'\right)\right). \tag{4.45}$$

All curvature information is lost with the linearized constraints which implies that a step of length $O(\Delta t)$ introduces errors of $O(\Delta t^2)$. Hence, the approach of Sauer and Schömer prevents the penetration of increasing by more than $O(\Delta t^2)$.

4.6 The Frictional Case

In this section, we will expand the formulation given in (4.27) to include friction. For each contact, we use two orthogonal unit vectors $\vec{t}_{1_k}$ and $\vec{t}_{2_k}$ which span the tangential plane at the k'th contact. Together with the normal vector $\vec{n}_k$ the three vectors form an orthogonal coordinate system. The friction cone at the k'th contact is approximated by a discretized version having η direction vectors $\vec{d}_{h_k}$ with $h = 1, \ldots, \eta$, where $\eta = 2i$ for all $i \in \mathbb{N}$ and $i \geq 2$. The direction vectors are concatenated into a matrix $\mathbf{D}_k \in \mathbb{R}^{3\times\eta}$,

$$\mathbf{D}_k = \left[\vec{d}_{1_k}, \ldots, \vec{d}_{\eta_k}\right], \tag{4.46}$$

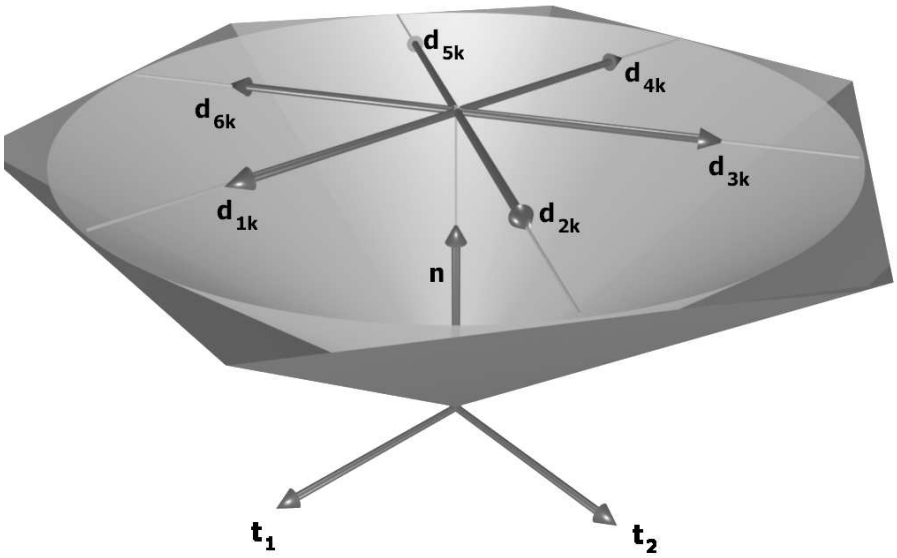


Figure 4.6: The friction pyramid approximation for $\eta = 6$. Observe that the vectors $\vec{d}_{h_k}$ positively span the friction pyramid.

where

$$\vec{d}_{h_k} = \cos\left(\frac{2(h-1)\pi}{\eta}\right)\vec{t}_{1_k} + \sin\left(\frac{2(h-1)\pi}{\eta}\right)\vec{t}_{2_k}. \tag{4.47}$$

We have transformed the spatial cone limiting the friction force due to Coulomb's friction law, called the friction cone into a friction pyramid with η facets, illustrated in Figure 4.6. For each direction vector we will use β_{h_k} for the magnitude of the component of friction force in the direction of $\vec{d}_{h_k}$. Like before we can build up a vector of all friction components $\vec{\beta}_k \in \mathbb{R}^\eta$,

$$\vec{\beta}_k = \left[\beta_{1_k}, \ldots, \beta_{\eta_k}\right]^T. \tag{4.48}$$

The modification of the equations of motion (4.7) is the definition of contact force $\vec{f}_k$ from (4.10) which we now write as,

$$\vec{f}_k = f_k\vec{n}_k + \mathbf{D}_k\vec{\beta}_k. \tag{4.49}$$

As before we use matrix notation which will allow us to write the equations of motion in a single matrix equation. The generalized acceleration is again described by the Newton-Euler equations

$$\dot{\vec{u}} = \mathbf{M}^{-1}\left(\mathbf{C}\left(\mathbf{N}\vec{f} + \mathbf{D}\beta\right) + \vec{f}_{ext}\right). \tag{4.50}$$

We need a vector, $\vec{\beta} \in \mathbb{R}^{\eta K}$

$$\vec{\beta} = \left[\vec{\beta}_1^T, \ldots, \vec{\beta}_K^T, \right]^T \tag{4.51}$$

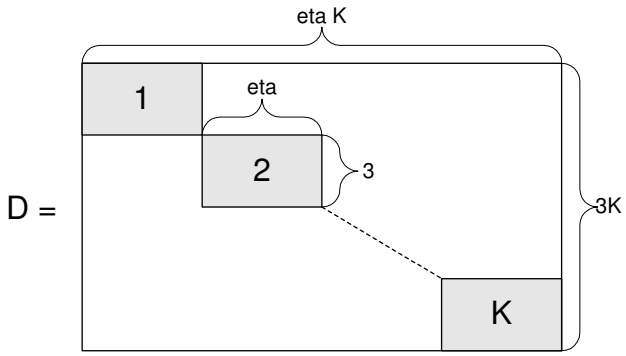


Figure 4.7: The $\mathbf{D}$ matrix layout.

That is, the concatenation of the $\vec{\beta}_k$-vectors. We also need the matrix $\mathbf{D} \in \mathbb{R}^{3K \times \eta K}$

$$\mathbf{D} = \begin{bmatrix} \mathbf{D}_1 & & \mathbf{0} \\ & \mathbf{D}_2 & \\ & \vdots & \\ \mathbf{0} & & \mathbf{D}_K \end{bmatrix} \tag{4.52}$$

Figure 4.7 illustrates the $\mathbf{D}$ matrix layout. Using an Euler-Scheme gives us the discretized approximation,

$$\vec{u}^{\,t+\Delta t} = \vec{u}^{\,t} + \Delta t \mathbf{M}^{-1} \left(\mathbf{C} \left(\mathbf{N}\vec{f} + \mathbf{D}\beta \right) + \vec{f}_{ext} \right). \tag{4.53}$$

In order to model the relationship between the normal force and the friction, known as Coulomb's Friction Law, we need to add two complementarity conditions for the friction forces in addition to the previous complementarity condition for the normal force. We have a total of three complementarity conditions for the k'th contact

$$\lambda_k \vec{e}_k + \mathbf{D}_k^T \mathbf{P}_k^T \mathbf{C}^T \vec{u}^{\,t+\Delta t} \geq 0 \qquad \text{compl. to} \quad \vec{\beta}_k \geq 0, \tag{4.54a}$$

$$\mu_k f_k - \vec{e}_k^T \vec{\beta}_k \geq 0 \qquad \text{compl. to} \quad \lambda_k \geq 0, \tag{4.54b}$$

$$\vec{n}_k^T \mathbf{P}_k^T \mathbf{C}^T \vec{u}^{\,t+\Delta t} - \frac{\nu_k}{\Delta t} \geq 0 \qquad \text{compl. to} \quad f_k \geq 0. \tag{4.54c}$$

Where μ_k is the friction coefficient at the k'th contact point, and $\vec{e}_k = [1, \ldots, 1]^T \in \mathbb{R}^\eta$. The symbol λ_k is a Lagrange multiplier with no real physical meaning, but it is an approximation to the magnitude of the relative tangential contact velocity. We will now list the possible contact states modeled by (4.54).

Separation: In this case $\vec{n}_k^T \mathbf{P}_k^T \mathbf{C}^T \vec{u}^{\,t+\Delta t} - \frac{\nu_k}{\Delta t} > 0$, and (4.54c) implies that $f_k = 0$. Substitution of this into (4.54b), implies that $\vec{\beta}_k = 0$, i.e. there is no friction force. From (4.54a) we see that λ_k can take on any value without violating the conditions.

Sliding: For sliding, $\mathbf{D}_k^T \mathbf{P}_k^T \mathbf{C}^T \vec{u}^{\,t+\Delta t}$ is non-zero, since the columns of $\mathbf{D}_k$ positively span the entire contact plane. There must be at least one direction vector such that $\vec{d}_{h_k}^T \mathbf{P}_k^T \mathbf{C}^T \vec{u}^{\,t+\Delta t} < 0$, and since the corresponding $\beta_{h_k} > 0$, we must have $\lambda_k > 0$ for (4.54a) to hold, and (4.54b) implies that $\beta_{h_k} = \mu_k f_k$.

Rolling: In this case $\mathbf{D}_k^T\mathbf{P}_k^T\mathbf{C}^T\vec{u}^{\,t+\Delta t}$ is zero, and (4.54a) implies that $\lambda_k \geq 0$. There are two interesting cases:

Case 1: Choosing $\lambda_k = 0$ (4.54a) implies that $\vec{\beta}_k \geq 0$. This means that the contact impulse can range over the interior and the surface of the discretized friction cone.

Case 2: Choosing $\lambda_k > 0$ (4.54a) implies that $\vec{\beta}_k = 0$. Then (4.54b) will only be fulfilled if $\mu_k f_k = 0$. This is a non-generic case that occurs by chance in the absence of a frictional impulse that is when $\mu_k = 0$.

We can now proceed analogous to the frictionless case and try to insert (4.53) into (4.54a) and (4.54c):

$$\lambda_k\vec{e}_k + \mathbf{D}_k^T\mathbf{P}_k^T\mathbf{C}^T\left(\vec{u}^{\,t} + \Delta t\mathbf{M}^{-1}\left(\mathbf{C}\left(\mathbf{N}\vec{f} + \mathbf{D}\beta\right) + \vec{f}_{\text{ext}}\right)\right) \geq 0$$
$$\text{compl. to} \quad \vec{\beta}_k \geq 0, \tag{4.55a}$$
$$\vec{n}_k^T\mathbf{P}_k^T\mathbf{C}^T\left(\vec{u}^{\,t} + \Delta t\mathbf{M}^{-1}\left(\mathbf{C}\left(\mathbf{N}\vec{f} + \mathbf{D}\beta\right) + \vec{f}_{\text{ext}}\right)\right) - \frac{\nu_k}{\Delta t} \geq 0$$
$$\text{compl. to} \quad \vec{f}_k \geq 0. \tag{4.55b}$$

Rearranging provides us with two new complementarity conditions which replace those in (4.54a) and (4.54c):

$$\Delta t\mathbf{D}_k^T\mathbf{P}_k^T\mathbf{C}^T\mathbf{M}^{-1}\mathbf{C}\mathbf{N}\vec{f} + \Delta t\mathbf{D}_k^T\mathbf{P}_k^T\mathbf{C}^T\mathbf{M}^{-1}\mathbf{C}\mathbf{D}\beta$$
$$+\lambda_k\vec{e}_k + \mathbf{D}_k^T\mathbf{P}_k^T\mathbf{C}^T\vec{u}^{\,t} + \Delta t\mathbf{D}_k^T\mathbf{P}_k^T\mathbf{C}^T\mathbf{M}^{-1}\vec{f}_{\text{ext}} \geq 0$$
$$\text{compl. to} \quad \vec{\beta}_k \geq 0, \tag{4.56a}$$
$$\Delta t\vec{n}_k^T\mathbf{P}_k^T\mathbf{C}^T\mathbf{M}^{-1}\mathbf{C}\mathbf{N}\vec{f} + \Delta t\vec{n}_k^T\mathbf{P}_k^T\mathbf{C}^T\mathbf{M}^{-1}\mathbf{C}\mathbf{D}\beta$$
$$+\vec{n}_k^T\mathbf{P}_k^T\mathbf{C}^T\vec{u}^{\,t} + \Delta t\vec{n}_k^T\mathbf{P}_k^T\mathbf{C}^T\mathbf{M}^{-1}\vec{f}_{\text{ext}} - \frac{\nu_k}{\Delta t} \geq 0$$
$$\text{compl. to} \quad \vec{f}_k \geq 0. \tag{4.56b}$$

By rearranging the complementarity conditions (4.54b), (4.56a), and (4.56b) we can formulate the LCP-formulation in matrix from as,

$$\begin{bmatrix} \mathbf{D}^T\mathbf{C}^T\mathbf{M}^{-1}\mathbf{C}\mathbf{D} & \mathbf{D}^T\mathbf{C}^T\mathbf{M}^{-1}\mathbf{C}\mathbf{N} & \mathbf{E} \\ \mathbf{N}^T\mathbf{C}^T\mathbf{M}^{-1}\mathbf{C}\mathbf{D} & \mathbf{N}^T\mathbf{C}^T\mathbf{M}^{-1}\mathbf{C}\mathbf{N} & \mathbf{0} \\ -\mathbf{E}^T & \mu & \mathbf{0} \end{bmatrix} \cdot \begin{bmatrix} \Delta t\vec{\beta} \\ \Delta t\vec{f} \\ \vec{\lambda}_{\text{aux}} \end{bmatrix}$$
$$+ \begin{bmatrix} \mathbf{D}^T\mathbf{C}^T\left(\vec{u}^{\,t} + \Delta t\mathbf{M}^{-1}\vec{f}_{\text{ext}}\right) \\ \mathbf{N}^T\mathbf{C}^T\left(\vec{u}^{\,t} + \Delta t\mathbf{M}^{-1}\vec{f}_{\text{ext}}\right) - \frac{\vec{\nu}}{\Delta t} \\ 0 \end{bmatrix} \geq 0$$
$$\text{compl. to} \quad \begin{bmatrix} \Delta t\vec{\beta} \\ \Delta t\vec{f} \\ \vec{\lambda}_{\text{aux}} \end{bmatrix} \geq 0, \tag{4.57a}$$

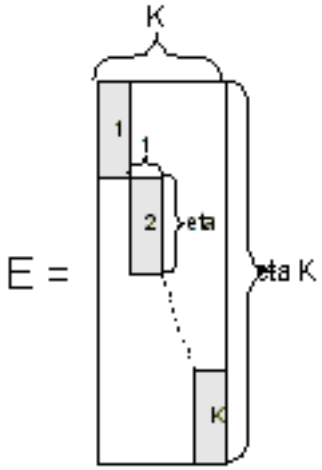


Figure 4.8: The **E** matrix layout.

where the diagonal matrix $\mu \in \mathbb{R}^{K\times K}$ is given as,

$$\mu = \begin{bmatrix} \mu_1 & & 0 \\ & \mu_2 & \\ & & \ddots \\ 0 & & \mu_K \end{bmatrix}, \tag{4.58}$$

and the matrix $\mathbf{e} \in \mathbb{R}^{\eta K\times K}$ is given by,

$$\mathbf{E} = \begin{bmatrix} \vec{e}_1 & & 0 \\ & \vec{e}_2 & \\ & & \ddots \\ 0 & & \vec{e}_K \end{bmatrix}. \tag{4.59}$$

That is, $\mathbf{E}$ consists of $\eta \times 1$ dimensional sub matrices, and all sub matrices on the diagonal consist of ones and off diagonal sub matrices are 0, see Figure 4.8. Finally, the vector $\vec{\lambda}_{\text{aux}} \in \mathbb{R}^K$ is given as,

$$\vec{\lambda}_{\text{aux}} = [\lambda_1, \ldots, \lambda_K]^T. \tag{4.60}$$

Let the matrix $\mathbf{A} \in \mathbb{R}^{(\eta+2)K\times(\eta+2)K}$ be defined as,

$$\mathbf{A} = \begin{bmatrix} \mathbf{D}^T\mathbf{C}^T\mathbf{M}^{-1}\mathbf{C}\mathbf{D} & \mathbf{D}^T\mathbf{C}^T\mathbf{M}^{-1}\mathbf{C}\mathbf{N} & \mathbf{E} \\ \mathbf{N}^T\mathbf{C}^T\mathbf{M}^{-1}\mathbf{C}\mathbf{D} & \mathbf{N}^T\mathbf{C}^T\mathbf{M}^{-1}\mathbf{C}\mathbf{N} & \mathbf{0} \\ -\mathbf{E}^T & \mu & \mathbf{0} \end{bmatrix}, \tag{4.61}$$

and the vector $\vec{x} \in \mathbb{R}^{(\eta+2)K}$ as,

$$\vec{x} = \begin{bmatrix} \Delta t\vec{\beta} \\ \Delta t\vec{f} \\ \vec{\lambda}_{\text{aux}} \end{bmatrix}, \tag{4.62}$$

and the vector $b \in \mathbb{R}^{(\eta+2)K}$ as,

$$\vec{b} = \begin{bmatrix} \mathbf{D}^T\mathbf{C}^T\left(\vec{u}^{\,t} + \Delta t\mathbf{M}^{-1}\vec{f}_{\text{ext}}\right) \\ \mathbf{N}^T\mathbf{C}^T\left(\vec{u}^{\,t} + \Delta t\mathbf{M}^{-1}\vec{f}_{\text{ext}}\right) - \frac{\vec{\nu}}{\Delta t} \\ 0 \end{bmatrix} \tag{4.63}$$

then we see that we have a typical LCP formulation of the form

$$\mathbf{A}\vec{x} + \vec{b} \geq 0 \quad \text{compl. to} \quad \vec{x} \geq 0. \tag{4.64}$$

The above formulation can be further extended to include torsional friction [146].

Because of real-time demands, a scalable friction model for time-critical computing is of importance. The constraint-based method is easily adopted to a scalable friction model by controlling the number of facets, η, used in the friction pyramid approximation.

Several methods could be used for setting the value of η. A global control could be used based on the amount of computation time or the total number of variables in the LCP problem. If either of these exceed some given limits, η is decreased correspondingly.

However, local control could also be used. Often, only visualization is important, therefore accurate friction is only needed for objects seen by a user. In such cases it is reasonable to use a low η for all objects outside the view-frustum, and for those objects inside the view-frustum, a higher η-value is used.

4.7 Joints

In the previous sections we have treated the problem of contact mechanics using classical mechanics taught in first year undergraduate physics and linear algebra. The approach is straightforward and easy to understand even though there is a lot of symbols and notation. Until now, we have treated what is known as unilateral contacts, where unilateral refers to the "$\geq$"-constraints on the contact forces. In this section, we will try to generalize our formulation and include bilateral-constraints. Here bilateral refers to a "$=$"-constraint on the constraint forces. Bilateral constraints are used for modeling joints between the bodies such as hinges and ball-in-socket connections.

In this section, we will show how we can go from the formulation based on "classical mechanics" to a formulation based on "analytical mechanics". To achieve a more abstract and general formulation, we need to introduce concepts of: holonomic and non-holonomic constraints, and Jacobians.

There exists many papers which takes the analytical mechanics approach for formulating their complementarity problems, e.g. [117, 8, 4, 5]. A useful reference for further reading on analytical mechanics is [65].

4.7.1 Holonomic Constraints

Working with constraints, we are particularly interested in the number of degrees of freedom (DOF) that is the minimum set of parameters needed to describe the motion of our system. For instance, a free moving body has 6 DOFs, because we need at least three parameters to describe its position, and at least three parameters to describe its

orientation. For two free floating rigid bodies we have 12 DOFs, from which we conclude that the smallest possible generalized position vector we can find will have 12 entries.

Following the conventions from previous sections, the spatial position vector $\vec{s}_l =\in \mathbb{R}^{14}$ for the l'th joint between the two bodies B_{i_l} and B_{j_l} can be written as,

$$\vec{s}_l = [\vec{r}_{i_l}, q_{i_l}, \vec{r}_{j_l}, q_{j_l}]^T . \tag{4.65}$$

To ease notation we will not bother with writing the subscript indicating the joint number or the contact number in the following sections, so we simply write,

$$\vec{s} = [\vec{r}_i, q_i, \vec{r}_j, q_j]^T . \tag{4.66}$$

The position vector $\vec{s}$ is not the minimum set of parameters, since we use quaternions for the representation of the orientations, and thus use four parameters instead of the minimal three for each orientation. For describing velocities and accelerations we could use time derivatives of the quaternions, but this is tedious, since the laws of physics use angular velocities $\vec{\omega}$. Instead, we need a transformation like the one we introduced in Section 4.3,

$$\dot{\vec{s}} = \mathbf{S}\vec{u}, \tag{4.67}$$

where

$$\vec{u} = [\vec{v}_i, \vec{\omega}_i, \vec{v}_j, \vec{\omega}_j]^T , \tag{4.68}$$

and

$$\mathbf{S} = \begin{bmatrix} \mathbf{1} & & & \mathbf{0} \\ & \mathbf{Q}_i & & \\ & & \mathbf{1} & \\ \mathbf{0} & & & \mathbf{Q}_j \end{bmatrix} . \tag{4.69}$$

We write the position vector $\vec{r} \in \mathbb{R}^{12}$ associated with the integrals of $\vec{u}$ as

$$\vec{r} = \left[\vec{r}_i, \vec{\theta}_i, \vec{r}_j, \vec{\theta}_j\right]^T , \tag{4.70a}$$

$$= [x_i, y_i, z_i, \alpha_i, \beta_i, \gamma_i, x_j, y_j, z_j, \alpha_j, \beta_j, \gamma_j]^T . \tag{4.70b}$$

Here $\vec{\theta}_i$ is the integral quantities of $\vec{\omega}_i$, i.e.,

$$\vec{u} = \frac{d}{dt}\vec{r}. \tag{4.71}$$

In general, the quantities $\vec{\theta}_i$ and $\vec{\theta}_j$ in $\vec{r}$ do not give meaning as finite quantities, and in plain computer graphics language you can not use them like Euler angles to make a rotation matrix. Nevertheless, $\vec{r}$ is a minimum spatial position vector.

When we link two rigid bodies together by a joint, we are removing DOFs from the system, and we can therefore find an even smaller generalized position vector. For instance, if we make a rigid connection between two free floating bodies then we can remove 6 DOFs, because we only need to describe the movement of one of the bodies, and the movement of the other body will follow immediately from the movement of the first body. This means that the smallest possible generalized position vector has 6 entries. From the example, we see that we can at most remove 6 DOFs from any joint.

By definition, a holonomic constraint[1] between two bodies B_i and B_j can be written as a function Φ of time and a spatial position vector $\vec{s} =\in \mathbb{R}^{14}$, such that we always have,

$$\Phi(t, \vec{s}) = 0. \tag{4.72}$$

All joint types presented in this dissertation can be modeled by time-independent holonomic constraints, meaning that for the l'th joint we have m holonomic constraints

$$\Phi_1(\vec{s}) = 0, \tag{4.73a}$$
$$\Phi_2(\vec{s}) = 0, \tag{4.73b}$$
$$\vdots$$
$$\Phi_m(\vec{s}) = 0, \tag{4.73c}$$

where m is the number of degrees of freedom removed by the constraints. This type of holonomic constraint is called a scleronomous constraint.

Assume that the l'th joint is a ball-in-socket joint between the two bodies B_i and B_j. A ball-in-socket joint is characterized by the fact that two points, one from each body, is always connected to each other, meaning that we have one constraint saying the x coordinates of the two points must be equal, another constraint requiring equality of the y-coordinates and a third one for equality of the z-coordinates. That is if we let the two points be specified by two fixed vectors $\vec{r}^{\,i}_{\text{anc}}$ and $\vec{r}^{\,j}_{\text{anc}}$ in the respective body frames of the two bodies as shown in Figure 4.9, then we can formulate the geometric constraint as follows

$$\underbrace{\left[\left(\vec{r}_i + \mathbf{R}(q_i)\vec{r}^{\,i}_{\text{anc}}\right) - \left(\vec{r}_j + \mathbf{R}(q_j)\vec{r}^{\,j}_{\text{anc}}\right)\right]_x}_{\Phi_1} = 0, \tag{4.74a}$$
$$\underbrace{\left[\left(\vec{r}_i + \mathbf{R}(q_i)\vec{r}^{\,i}_{\text{anc}}\right) - \left(\vec{r}_j + \mathbf{R}(q_j)\vec{r}^{\,j}_{\text{anc}}\right)\right]_y}_{\Phi_2} = 0, \tag{4.74b}$$
$$\underbrace{\left[\left(\vec{r}_i + \mathbf{R}(q_i)\vec{r}^{\,i}_{\text{anc}}\right) - \left(\vec{r}_j + \mathbf{R}(q_j)\vec{r}^{\,j}_{\text{anc}}\right)\right]_z}_{\Phi_3} = 0, \tag{4.74c}$$

where $\mathbf{R}(q)$ is the corresponding rotation matrix of the quaternion q. From the equations above, it is clear that the geometric constraint characterizing the ball-in-socket joint can be expressed as three holonomic constraints on vector form as,

$$\vec{\Phi}(\vec{s}) = \begin{bmatrix} \Phi_1(\vec{s}) \\ \Phi_2(\vec{s}) \\ \Phi_3(\vec{s}) \end{bmatrix} = 0. \tag{4.75}$$

Notice that the small example is not only illustrative. It actually provides us with a recipe for deriving different joint types. In conclusion, a holonomic constraint is equivalent to removing a degree of freedom from the system, which means that we can find a generalized position vector with one entry less than the spatial position vector.

[1] Holonomic (holos in Greek = integer in Latin = integrable) is synonymous with completely integrable. A holonomic constraint is exact and often understood as being independent of the generalized velocities.

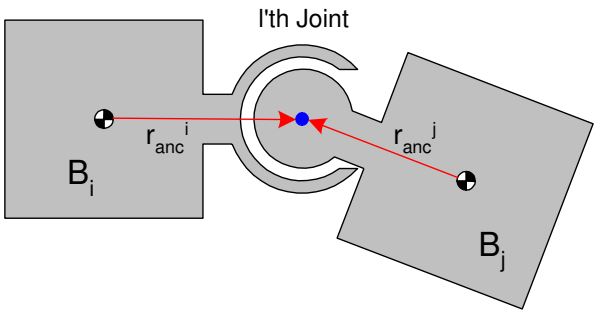


Figure 4.9: A 2D illustration of a ball in a socket joint.

By differentiation with respect to time, we can derive a kinematic constraint from each holonomic constraint,

$$\frac{d}{dt}\vec{\Phi}_l(\vec{s}) = \frac{\partial\vec{\Phi}}{\partial\vec{s}}\frac{d\vec{s}}{dt} \tag{4.76a}$$

$$= \underbrace{\frac{\partial\vec{\Phi}}{\partial\vec{s}}\mathbf{S}}_{J_\Phi}\vec{u} \tag{4.76b}$$

$$= 0. \tag{4.76c}$$

The matrices $\frac{\partial\vec{\Phi}}{\partial\vec{s}} \in \mathbb{R}^{m\times 14}$ and $J_\Phi \in \mathbb{R}^{m\times 12}$ are called Jacobians, and they describe relations between velocities in different coordinate representations. Finally, we have the kinematic constraint,

$$J_\Phi\vec{u}_l = 0. \tag{4.77}$$

Performing another differentiation wrt. time leads to the acceleration constraint,

$$\frac{d^2}{dt^2}\vec{\Phi}(\vec{s}(t)) = \frac{d}{dt}\left(J_\Phi\vec{u}\right) \tag{4.78a}$$

$$= \frac{d}{dt}\left(J_\Phi\right)\vec{u} + J_\Phi\frac{d}{dt}\left(\vec{u}\right) \tag{4.78b}$$

$$= 0, \tag{4.78c}$$

from which we conclude that

$$J_\Phi\dot{\vec{u}} = -\dot{J}_\Phi\vec{u}. \tag{4.79}$$

For our velocity-based formulation we have no use of the acceleration constraint. However, if we were to setup an acceleration-based formulation, we would need to augment the Newton-Euler equations with these acceleration constraints.

From classical mechanics it is well known that the generalized constraint force exerted by a holonomic constraint can be written as,

$$\vec{F}_\Phi = J_\Phi^T\vec{\lambda}_\Phi. \tag{4.80}$$

This follows from the principle of virtual work. The $\vec{\lambda}_\Phi \in \mathbb{R}^m$ is a vector of Lagrange multipliers. They account for the reaction forces coming from the joint bearings, and the Lagrange multipliers can take any real value, both positive and negative. Observe that the dimension of $\vec{\lambda}_\Phi$ depends on the number of constraints on the joint. Thus we conclude that we have as many independent reaction forces as there are constraints.

4.7.2 Non-holonomic Constraints

A non-holonomic constraint[2] is a differential constraint that cannot be integrated, however in this context we define a non-holonomic constraint as constraint that cannot be put into the form of a holonomic constraint (4.72). There are many different kinds of non-holonomic constraints, and we will restrict ourselves to a certain kind, namely those called unilateral constraints.

A non-holonomic constraint between two bodies B_i and B_j can by definition be written as a function Ψ of time and generalized position vector $\vec{s} \in \mathbb{R}^{14}$, such that we always have,

$$\Psi(t, \vec{s}) \geq 0. \tag{4.81}$$

The condition for a contact can be modeled by $(1 + \eta)$ time-independent non-holonomic constraints. That is

$$\vec{\Psi}(\vec{s}) \geq 0. \tag{4.82}$$

This looks very different from the contact conditions we have seen previously, but do not despair. The connection with our previous derivations will be made clear later on. Taking the time derivative gives us a kinematic contact constraint,

$$\begin{aligned}
\frac{d}{dt}\vec{\Psi}(\vec{s}) &= \frac{\partial \vec{\Psi}}{\partial \vec{s}}\frac{d\vec{s}}{dt} && \text{(4.83a)}\\
&= \underbrace{\frac{\partial \vec{\Psi}}{\partial \vec{s}}}_{J_\Psi}\mathbf{S}\,\vec{u} && \text{(4.83b)}\\
&= J_\Psi \vec{u} && \text{(4.83c)}\\
&\geq 0, && \text{(4.83d)}
\end{aligned}$$

where $J_\Psi \in \mathbb{R}^{(1+\eta)\times 12}$ is the Jacobian of the contact constraint. Taking the time derivative one more time yields an acceleration constraint,

$$\begin{aligned}
\frac{d^2}{dt^2}\vec{\Psi}(\vec{s}(t)) &= \frac{d}{dt}\left(J_\Psi \vec{u}\right) && \text{(4.84a)}\\
&= \dot{J}_\Psi \vec{u} + J_\Psi \dot{\vec{u}} && \text{(4.84b)}\\
&\geq 0. && \text{(4.84c)}
\end{aligned}$$

The generalized constraint force exerted by the contact constraint can be written as,

$$\vec{F}_\Psi = J_\Psi^T \vec{\lambda}_\Psi, \tag{4.85}$$

where $\vec{\lambda}_\Psi \in \mathbb{R}^{1+\eta}$ is the vector of Lagrange multipliers. Unlike the Lagrange multipliers used for the joint reaction forces, the Lagrange multipliers for the contacts can only take nonnegative values, i.e.

$$\vec{\lambda}_\Psi \geq 0. \tag{4.86}$$

[2]Nonintegrable is synonymous with non-holonomic. A non-holonomic constraint can not be integrated to become a strictly algebraic expression. In the field of robotic manipulation this is sometimes understood as a constraint that is dependent on the generealized velocities.

As pointed out in Anitescu and Potra [5], one should be careful about what the constraint $\frac{d}{dt}\vec{\Psi}(\vec{s}) \geq 0$ implies. Because if we have

$$\vec{\Psi}(\vec{s}) > 0 \tag{4.87}$$

indicating a potential future contact, this does not imply that

$$\frac{d}{dt}\vec{\Psi}(\vec{s}) > 0. \tag{4.88}$$

Only so-called "active" contacts, where

$$\vec{\Psi}(\vec{s}) = 0, \tag{4.89}$$

requires this. This boils down to that separated contacts are allowed to move towards each other, until they become touching contacts, and a touching contact can either continue being a touching contact, or it can become a separated contact.

Momentarily re-introducing the subscript for the contact ordering, we can write all the kinematic constraints and force contributions as

$$J_{\Psi_k}\vec{u}_k \geq 0, \quad \text{where} \quad J_{\Psi_k} \in \mathbb{R}^{(1+\eta)\times 12}, \tag{4.90a}$$

$$\vec{f}_{\Psi_k} = J_{\Psi_k}^T \vec{\lambda}_{\Psi_k}, \quad \text{where} \quad \vec{\lambda}_{\Psi_k} \in \mathbb{R}^{1+\eta}. \tag{4.90b}$$

Concatenating them into a matrix equation we have,

$$J_{\text{contact}}\vec{u} = 0, \tag{4.91a}$$

$$\vec{f}_{\text{contact}} = J_{\text{contact}}^T \vec{\lambda}_{\text{contact}}, \tag{4.91b}$$

where $\vec{u} \in \mathbb{R}^{6n}$ is the generalized velocity vector introduced in Section 4.3, $\vec{u} = [\vec{v}_1, \vec{\omega}_1, \cdots, \vec{v}_n, \vec{\omega}_n]^T$, and $\vec{\lambda}_{\text{contact}} \in \mathbb{R}^{K(1+\eta)}$ is the concatenated vector of all the Lagrange multipliers,

$$\vec{\lambda}_{\text{contact}} = \left[\vec{\lambda}_{\Psi_1}^1, \ldots, \vec{\lambda}_{\Psi_1}^{(\eta+1)}, \ldots, \vec{\lambda}_{\Psi_K}^1, \ldots, \vec{\lambda}_{\Psi_K}^{(\eta+1)}\right]^T. \tag{4.92}$$

The $J_{\text{contact}} \in \mathbb{R}^{K(1+\eta)\times 6n}$ is the system Jacobian for all the contacts, and it is given by,

$$J_{\text{contact}} = \begin{bmatrix} J_{\Psi_1}^1 & \cdots\cdots\cdots\cdots\cdots & J_{\Psi_1}^n \\ \vdots & & \vdots \\ J_{\Psi_k}^1 & \cdots \; J_{\Psi_k}^i \; \cdots \; J_{\Psi_k}^j \; \cdots & J_{\Psi_k}^n \\ \vdots & & \vdots \\ J_{\Psi_K}^1 & \cdots\cdots\cdots\cdots\cdots & J_{\Psi_K}^n \end{bmatrix}. \tag{4.93}$$

This Jacobian is inherently extremely sparse, since the k'th contact only involves two bodies i and j, meaning that the only non-zero entries in the k'th row of J_{contact} is the columns corresponding to the bodies i and j,

$$\begin{bmatrix} J_{\Psi_k}^i & J_{\Psi_k}^j \end{bmatrix} = J_{\Psi_k}. \tag{4.94}$$

We will now prove that

$$\mathbf{C}\left(\mathbf{N}\vec{f} + \mathbf{D}\vec{\beta}\right) \equiv J_{\text{contact}}^T \vec{\lambda}_{\text{contact}}. \tag{4.95}$$

The above equation follows from straightforward computations and permutations of the left hand side of

$$\mathbf{C}\left(\mathbf{N}\vec{f}+\mathbf{D}\vec{\beta}\right) = \mathbf{CN}\vec{f}+\mathbf{CD}\vec{\beta} \tag{4.96a}$$

$$= \underbrace{\begin{bmatrix}\mathbf{CN} & \mathbf{CD}\end{bmatrix}}_{\pi\left(J^T_{\text{contact}}\right)} \underbrace{\begin{bmatrix}\vec{f}\\ \vec{\beta}\end{bmatrix}}_{\pi\left(\vec{\lambda}_{\text{contact}}\right)} \tag{4.96b}$$

$$= \pi\left(J^T_{\text{contact}}\vec{\lambda}_{\text{contact}}\right), \tag{4.96c}$$

where $\pi(\cdot)$ is a permutation. Now simply swap rows and columns, such that

$$\vec{\lambda}^1_{\Psi_k} = f_k, \tag{4.97a}$$

$$\vec{\lambda}^2_{\Psi_k} = \beta_{1_k}, \tag{4.97b}$$

$$\vdots \quad \vdots$$

$$\vec{\lambda}^{(\eta+1)}_{\Psi_k} = \beta_{\eta_k}, \tag{4.97c}$$

and the relation between the Jacobian, J_{contact}, and the matrices $\mathbf{C}$, $\mathbf{N}$, and $\mathbf{D}$ is clear.

4.7.3 A Unified Notation for Unilateral and Bilateral Constraints

We now want to show that both bilateral and unilateral constraints can be added to the governing system of equations of motion through the same notion of Jacobians and Lagrange multipliers.

Momentarily re-introducing the subscripts on both the joint and contact ordering, we have K contacts and L joints, and we can write all their kinematic constraints and force contributions as,

$$J_{\Phi_l}\vec{u}_l = 0, \quad \text{where} \quad J_{\Phi_l} \in \mathbb{R}^{m_l\times 12}, \tag{4.98a}$$

$$J_{\Psi_k}\vec{u}_k \geq 0, \quad \text{where} \quad J_{\Psi_k} \in \mathbb{R}^{(1+\eta)\times 12}, \tag{4.98b}$$

$$\vec{f}_{\Phi_l} = J^T_{\Phi_l}\vec{\lambda}_{\Phi_l}, \quad \text{where} \quad \vec{\lambda}_{\Phi_l} \in \mathbb{R}^{m_l}, \tag{4.98c}$$

$$\vec{f}_{\Psi_k} = J^T_{\Psi_k}\vec{\lambda}_{\Psi_k}, \quad \text{where} \quad \vec{\lambda}_{\Psi_k} \in \mathbb{R}^{1+\eta}. \tag{4.98d}$$

Following the same recipe as in Section 4.7.2 for concatenating these into matrix notation, we get

$$J_{\text{joint}}\vec{u} = 0, \quad \text{where} \quad J_{\text{joint}} \in \mathbb{R}^{\left(\sum_l^L m_l\right)\times 6n}, \tag{4.99a}$$

$$J_{\text{contact}}\vec{u} \geq 0, \quad \text{where} \quad J_{\text{contact}} \in \mathbb{R}^{K(1+\eta)\times 6n}, \tag{4.99b}$$

$$\vec{f}_{\text{joint}} = J^T_{\text{joint}}\vec{\lambda}_{\text{joint}}, \quad \text{where} \quad \vec{\lambda}_{\text{joint}} \in \mathbb{R}^{\sum_l^L m_l}, \tag{4.99c}$$

$$\vec{f}_{\text{contact}} = J^T_{\text{contact}}\vec{\lambda}_{\text{contact}}, \quad \text{where} \quad \vec{\lambda}_{\text{contact}} \in \mathbb{R}^{K(1+\eta)}. \tag{4.99d}$$

The Jacobian J_{contact} and the Lagrange multiplier vector $\vec{\lambda}_{\text{contact}}$ was given in (4.93) and (4.92). The system joint Jacobian J_{joint} and the joint Lagrange multiplier vector $\vec{\lambda}_{\text{joint}}$

follows the same pattern and is given as,

$$J_{\text{joint}} = \begin{bmatrix} J^1_{\Phi_1} & \cdots & & & & & J^n_{\Phi_1} \\ \vdots & & & & & & \vdots \\ J^1_{\Phi_l} & \cdots & J^i_{\Phi_l} & \cdots & J^j_{\Phi_l} & \cdots & J^n_{\Phi_l} \\ \vdots & & & & & & \vdots \\ J^1_{\Phi_L} & \cdots & & & & & J^n_{\Phi_L} \end{bmatrix}. \tag{4.100}$$

This Jacobian is inherently extremely sparse, since the l'th joint only involves two bodies i and j, meaning that the only non-zero entries in the l'th row of J_{joint} is the columns corresponding to the bodies i and j,

$$\begin{bmatrix} J^i_{\Phi_l} & J^j_{\Phi_l} \end{bmatrix} = J_{\Phi_l}, \tag{4.101}$$

and

$$\vec{\lambda}_{\text{joint}} = \left[\vec{\lambda}^1_{\Phi_1}, \ldots, \vec{\lambda}^{m_1}_{\Phi_1}, \ldots, \vec{\lambda}^1_{\Phi_K}, \ldots, \vec{\lambda}^{m_K}_{\Phi_K} \right]^T. \tag{4.102}$$

Using the matrix notation to write constraint forces of both bilateral and unilateral constraint, the generalized acceleration vector can be written as,

$$\dot{\vec{u}} = \mathbf{M}^{-1} \left(\vec{f}_{\text{contact}} + \vec{f}_{\text{joint}} + \vec{f}_{\text{ext}} \right) \tag{4.103a}$$

$$= \mathbf{M}^{-1} \left(J^T_{\text{contact}} \vec{\lambda}_{\text{contact}} + J^T_{\text{joint}} \vec{\lambda}_{\text{joint}} + \vec{f}_{\text{ext}} \right). \tag{4.103b}$$

This is a completely general way to add constraints, and it will be further explored in the remainder of this chapter, and in the end it will also lead to a general and efficient implementation framework.

4.8 Joint Modeling

In this section, we will derive the machinery for modeling joints and later joint limits as well as joint motors. We will start by introducing a sub matrix pattern of the Jacobian matrix. Hereafter, we will describe joint error, connectivity and error reduction.

For the $l'th$ joint constraint we can write the kinematic constraint, as

$$J_l \vec{u}_l = 0. \tag{4.104}$$

Since we will focus on joint types, we will omit writing the subscript indicating the joint "ordering". That is, for a given joint type we simply write the kinematic constraint as, $J\vec{u} = 0$. There is a remarkable sub matrix pattern of the Jacobians which we will make extensive use of, because later on it will make the assembly of the system matrix easier. Writing the generalized velocity vector with its sub vectors as,

$$J\vec{u} = 0, \tag{4.105a}$$

$$\begin{bmatrix} \mathbf{J}^i_{\text{lin}} & \mathbf{J}^i_{\text{ang}} & \mathbf{J}^j_{\text{lin}} & \mathbf{J}^j_{\text{ang}} \end{bmatrix} \begin{bmatrix} \vec{v}_i \\ \vec{\omega}_i \\ \vec{v}_j \\ \vec{\omega}_j \end{bmatrix} = 0. \tag{4.105b}$$

Observe that there is a part of the Jacobian matrix that is only multiplied with the linear velocity of body i which is denoted $\mathbf{J}^i_{\text{lin}}$, a part that is only multiplied by the angular part of body i, $\mathbf{J}^i_{\text{ang}}$, and so on. In fact, we can interpret

$$\mathbf{J}^i_{\text{lin}}\vec{v}_i + \mathbf{J}^i_{\text{ang}}\vec{\omega}_i, \tag{4.106}$$

as the velocity of the joint bearings on body i, and

$$\mathbf{J}^j_{\text{lin}}\vec{v}_j + \mathbf{J}^j_{\text{ang}}\vec{\omega}_j, \tag{4.107}$$

as the velocity of the joint bearing on body j. It is now obvious, that in order to keep the joint bearings together, the bearings must move with the same velocity, and the sum must therefore be zero. This observation provides us with a strategy for designing the Jacobians: given the body velocities, set up a matrix equation, such that the relative velocity in the direction of the joint bearings is always zero.

4.8.1 Joint Error

By now, it should be clear that the kinematic constraints are constraints on the velocities, not the positions. This means that both numerical errors and errors stemming from internal approximations can sneak into the computations of the positions as the simulation proceeds. Here, we will outline the general idea of using stabilization for error correction and in Section 4.16 we will go into details of a projection based error correction method. Imagine that some positional error has occurred, such that there is a positional displacement of the joint bearings and/or a misalignment of the joint bearings. This error could be reduced by adjusting the velocities of the joint bearings, such that the error is smaller in the next simulation step. Therefore, we augment our kinematic constraints with a velocity error correction term, $\vec{b}$,

$$J\vec{u} = \vec{b}. \tag{4.108}$$

To illustrate this, we will present a simple one-dimensional example: Imagine two particles that can move along a line, where the particles are jointed together, such that their positions always should be equal. Their kinematic constraint will then be,

$$\vec{v}_i - \vec{v}_j = 0. \tag{4.109}$$

Now imagine that some error is present,

$$\vec{r}_{\text{err}} = \vec{r}_j - \vec{r}_i, \tag{4.110}$$

with $\mid \vec{r}_{\text{err}} \mid > 0$. To adjust the velocities such that this error is eliminated within some time Δt, we require that

$$\underbrace{\vec{v}_i - \vec{v}_j}_{J\vec{u}} = \underbrace{\frac{\vec{r}_{\text{err}}}{\Delta t}}_{\vec{b}}, \tag{4.111a}$$

$$J\vec{u} = \vec{b}. \tag{4.111b}$$

If joints or limits are subject to an initial error, and incident links are at rest, then error terms will accelerate the links. So not only will the error be corrected, but bodies will

continue to move afterward. This is obviously an unwanted effect! The error correction should not add kinetic energy to the system, otherwise the links connected by the joint will seem to start to accelerate unexpectedly. In fact, the error correction has the same effect as using Newton's collision law for solving simultaneous collisions [11]. An acceptable and practical workaround is to use an error reduction parameter to control the rate of error correction, which will be discussed in Section 4.8.3.

4.8.2 Connectivity

We will describe the connectivity and movements of all joints by using anchor points and joint axes. An anchor point is a point in space where two points, one from each incident body, are always perfectly aligned. The placement of an anchor point relative to a body, i, is given by a body frame vector, $\vec{r}_{\text{anc}}^{\,i}$. The position of the anchor point in the world coordinate system (WCS) wrt. to body i is given by,

$$\vec{r}_{\text{anc}}^{\,\text{wcs}} = \vec{r}_i + \mathbf{R}(q_i)\vec{r}_{\text{anc}}^{\,i}. \tag{4.112}$$

A joint axis describes an allowed direction of movement, such as an axis of rotation or a direction of sliding. The joint axis is given by a 3-dimensional unit vector, $\vec{s}_{\text{axis}}^{\,\text{wcs}}$. In Section 4.9 we will explain the details in describing different joint types using the notation of anchor points and joint axes.

This way of describing the connectivity is very similar to the "paired joint coordinate frames" described in Featherstone [59]. In comparison, anchor points correspond to the placement of the origin of the joint frames, and joint axes correspond to the orientation of the joint frames such as the z-axis of the joint coordinate frame. Alternative notations for describing the connectivity of jointed mechanisms is used in some literature [59, 39].

4.8.3 Error Reduction Parameter

The kind of approach for simulating joints that we outline in this dissertation belongs to a class of algorithms referred to as Full-Coordinate methods, because every body in a jointed mechanism is described by the full set of rigid body motion coordinates.

An alternative approach are the Reduced Coordinate methods, where a good example is Featherstone's algorithm [59]. The central idea is that only the relative motion of bodies between joints needs to be described. Therefore only the relative coordinates of the joints is needed.

The main difference between the two approaches is that Reduced Coordinate methods explicitly work with joint parameters, and that the position and placement of the links are derived from these joint parameters. On the other hand, with a Full-Coordinate method, we work explicitly on the links, and instead, we need to derive joint parameters if needed. There are some benefits and disadvantages of these methods which we will describe shortly.

The Reduced Coordinate methods are often computationally faster, since they have fewer variables to work on, and since they are often implemented by recursive algorithms like Armstrong and Featherstone [10, 59]. These recursive algorithms are often limited to tree-like mechanisms. Only by very difficult and computationally intractable extensions can these recursive algorithms handle closed loops and contacts.

The Full Coordinate methods are not limited by any kind of topology, but they are often more computationally demanding because they must describe all the constraints on each links rigid body motion. The Reduced Coordinate methods only need to describe the free movement which is often of smaller dimension.

Many people prefer the Full Coordinate methods, because they think the notation is easier to read and work with, while Reduced Coordinate methods appear to have long and difficult terms representing Coriolis and centripetal accelerations.

From a computer animation viewpoint, numerical errors in a Full Coordinate method seem to be much more noticeable than in a Reduced Coordinate method. The reason being that errors in the body coordinates, will split joints apart and introduce an effect called drifting, because links that are supposed to be jointed together drift apart. Reduced Coordinate methods do not suffer from the drifting problem, since no matter how big numerical errors one will obtain, the simulation will always show bodies connected properly.

In conclusion, with Full Coordinate methods we can expect drifting problems, and there are two ways these can arise in a working simulator:

- The user interacts with a mechanism, and forgets to set the correct position or orientation of all the links in a mechanism.
- During the simulation, errors can creep in that result in the links drifting away from their joints.

In Section 4.9 we describe the kinematic constraints of different joint types, and we will introduce some error correcting terms. These are all multiplied by a coefficient, k_{cor} which denotes a measure of the rate of error correction. The idea is as follows: for each joint we will have an error reduction parameter, k_{erp},

$$0 \leq k_{\text{erp}} \leq 1. \tag{4.113}$$

Its value is a measure of how much error reduction that should occur in the next simulation step. A value of zero means that there is no error correction at all, and a value of 1 means that the error should be totally eliminated.

If we let the duration of time in the next simulation step be denoted by a characteristic time-step, Δt, then the following constant is a measure of rate of change,

$$k_{\text{fps}} = \frac{1}{\Delta t}. \tag{4.114}$$

The coefficient k_{cor} can now be determined as,

$$k_{\text{cor}} = k_{\text{erp}} k_{\text{fps}}. \tag{4.115}$$

Setting $k_{\text{erp}} = 1$ is not recommended, since various internal approximations can cause the errors not to be completely fixed. The Open Dynamics Engine [112] uses the same approach for correcting errors, and they recommend to use a value around 0.8.

In the following, we are only concerned with error correction based on stabilization. Projection based error correction are discussed in Section 4.12 and Section 4.16.

4.9 Joint Types

In this section we will derive the Jacobians for several different kinds of joint types, needed for the kinematic constraints explained in the previous sections.

4.9.1 Ball-in-Socket Joint

A ball-in-socket joint allows arbitrary rotation between two bodies as illustrated in Figure 4.10. We already know that a ball-in-socket joint removes 3 DOFs, so we conclude

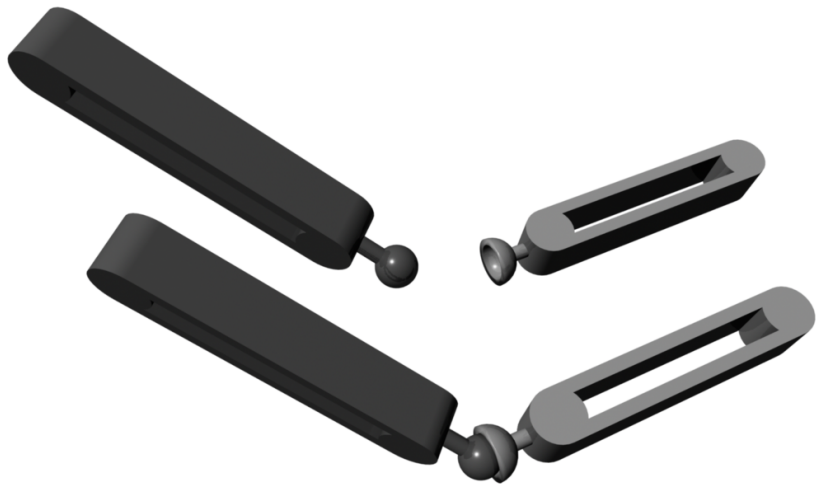


Figure 4.10: A ball-in-socket joint example.

that the Jacobian, J_{ball}, for the ball is a 3-by-12 matrix. From our previous example in Section 4.7.1 it should not come as a surprise that the sub matrix of the Jacobian is given by,

$$J_{\text{ball}} = \left[\mathbf{J}^i_{\text{lin}}, \mathbf{J}^j_{\text{lin}}, \mathbf{J}^i_{\text{ang}}, \mathbf{J}^j_{\text{ang}}\right], \tag{4.116}$$

where

$$\mathbf{J}^i_{\text{lin}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{4.117a}$$

$$\mathbf{J}^j_{\text{lin}} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \tag{4.117b}$$

$$\mathbf{J}^i_{\text{ang}} = -\left(\mathbf{R}(\mathbf{q_i})\tilde{\mathbf{r}}^{\mathbf{i}}_{\text{anc}}\right)^{\times}, \tag{4.117c}$$

$$\mathbf{J}^j_{\text{ang}} = \left(\mathbf{R}(\mathbf{q_j})\tilde{\mathbf{r}}^{\mathbf{j}}_{\text{anc}}\right)^{\times}, \tag{4.117d}$$

and where $\mathbf{J}^i_{\text{lin}} \in \mathbb{R}^{3\times3}$, $\mathbf{J}^i_{\text{ang}} \in \mathbb{R}^{3\times3}$, $\mathbf{J}^j_{\text{lin}} \in \mathbb{R}^{3\times3}$, $\mathbf{J}^j_{\text{ang}} \in \mathbb{R}^{3\times3}$, and the velocity error correcting term, $\vec{b}_{\text{ball}} \in \mathbb{R}^3$, is given by,

$$\vec{b}_{\text{ball}} = k_{\text{cor}} \left(\vec{r}_j + \mathbf{R}(q_j)\vec{r}^j_{\text{anc}} - \vec{r}_i - \mathbf{R}(q_i)\vec{r}^i_{\text{anc}} \right). \tag{4.118}$$

4.9.2 Hinge Joint

A hinge joint, also called a revolute joint, only allows relative rotation around a specified joint axis as illustrated in Figure 4.11. We describe the joint by an anchor point placed on

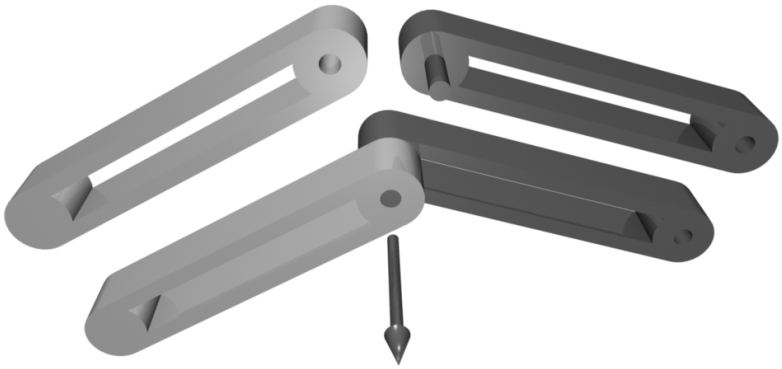


Figure 4.11: A hinge joint example.

the axis of rotation and a joint axis, $\vec{s}^{\text{wcs}}_{\text{axis}}$, given by a unit vector in the world coordinate system. We only have 1 DOF, meaning that the hinge-joint places 5 constraints on the relative movement and hence the Jacobian, J_{hinge}, is a 5-by-12 matrix,

$$J_{\text{hinge}} = \begin{bmatrix} \mathbf{J}^i_{\text{lin}} & \mathbf{J}^i_{\text{ang}} & \mathbf{J}^j_{\text{lin}} & \mathbf{J}^j_{\text{ang}} \end{bmatrix}, \tag{4.119}$$

where $\mathbf{J}^i_{\text{lin}} \in \mathbb{R}^{5\times3}$, $\mathbf{J}^i_{\text{ang}} \in \mathbb{R}^{5\times3}$, $\mathbf{J}^j_{\text{lin}} \in \mathbb{R}^{5\times3}$, $\mathbf{J}^j_{\text{ang}} \in \mathbb{R}^{5\times3}$, and $\vec{b}_{\text{hinge}} \in \mathbb{R}^5$. A hinge joint has the same kind of positional constraints as the ball-in-socket joint, so we can immediately borrow the 3 first rows of the ball-in-socket Jacobian and the error measure, and we only have to extend the hinge Jacobian with two more rows which will constrain the rotational freedom from the ball-in-socket joint to only one axis of rotation.

The strategy for adding the two rotational constraints is as follows: since we only want to allow rotations around the joint axis, only the relative angular velocity of the two bodies with respect to the joint axis is allowed to be nonzero, that is,

$$\vec{s}_{\text{axis}} \cdot (\vec{\omega}_i - \vec{\omega}_j) \neq 0. \tag{4.120}$$

The relative angular velocity with any other axis orthogonal to $\vec{s}_{\text{axis}}$ must be zero.

In particular, if we let the two vectors $\vec{t}_1{}^{\text{wcs}}, \vec{t}_2{}^{\text{wcs}} \in \mathbb{R}^3$ be two orthogonal unit vectors, and require them to be orthogonal to the joint axis $\vec{s}_{\text{axis}}^{\text{wcs}}$, then

$$\vec{t}_1{}^{\text{wcs}} \cdot (\vec{\omega}_i - \vec{\omega}_j) = 0, \tag{4.121a}$$
$$\vec{t}_2{}^{\text{wcs}} \cdot (\vec{\omega}_i - \vec{\omega}_j) = 0. \tag{4.121b}$$

From these two equations we have the two needed kinematic constraints, and we can write the hinge Jacobian as follows,

$$\mathbf{J}^i_{\text{lin}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \tag{4.122a}$$

$$\mathbf{J}^j_{\text{lin}} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \tag{4.122b}$$

$$\mathbf{J}^i_{\text{ang}} = \begin{bmatrix} -\left(\mathbf{R}(\mathbf{q_i})\tilde{\mathbf{r}}^{\mathbf{i}}_{\text{anc}}\right)^{\times} \\ \left(\vec{t}_1{}^{\text{wcs}}\right)^T \\ \left(\vec{t}_2{}^{\text{wcs}}\right)^T \end{bmatrix}, \tag{4.122c}$$

$$\mathbf{J}^j_{\text{ang}} = \begin{bmatrix} \left(\mathbf{R}(\mathbf{q_j})\tilde{\mathbf{r}}^{\mathbf{j}}_{\text{anc}}\right)^{\times} \\ -\left(\vec{t}_1{}^{\text{wcs}}\right)^T \\ -\left(\vec{t}_2{}^{\text{wcs}}\right)^T \end{bmatrix}. \tag{4.122d}$$

For the error-measure term, we already have the first three error measures from the ball-in-socket joint taking care of positional errors. Two further error measures are needed for rotational misalignment around any non-joint axes.

If we store the joint axis with respect to both body frames, $\vec{s}_{\text{axis}}^{\,i}$ and $\vec{s}_{\text{axis}}^{\,j}$, and then compute the joint axis directions in the world coordinate system with respect to each of the incident bodies, we get,

$$\vec{s}_i{}^{\text{wcs}} = \mathbf{R}(q_i)\vec{s}_{\text{axis}}^{\,i}, \tag{4.123a}$$
$$\vec{s}_j{}^{\text{wcs}} = \mathbf{R}(q_j)\vec{s}_{\text{axis}}^{\,j}. \tag{4.123b}$$

If $\vec{s}_i{}^{\text{wcs}} = \vec{s}_j{}^{\text{wcs}}$, then there is obviously no error in the relative hinge orientation between the bodies. If there is an error, then the bodies must be rotated such that $\vec{s}_i{}^{\text{wcs}}$ and $\vec{s}_j{}^{\text{wcs}}$ are equal. This can be done as follows: imagine the angle between the two vectors is θ_{err}. Then we can fix the relative error by rotation of θ_{err} radians around the axis,

$$\vec{u} = \vec{s}_i{}^{\text{wcs}} \times \vec{s}_j{}^{\text{wcs}}. \tag{4.124}$$

Let us say that we want to correct the error by the angle θ_{cor} within the time Δt which could be the size of the time-step in some time-stepping algorithm. Then we would need

a relative angular velocity of magnitude,

$$||\vec{\omega}_{\text{cor}}|| = \frac{\theta_{\text{cor}}}{\Delta t} \tag{4.125a}$$

$$= \frac{k_{\text{erp}}\theta_{\text{err}}}{\Delta t} \tag{4.125b}$$

$$= k_{\text{erp}}\frac{1}{\Delta t}\theta_{\text{err}} \tag{4.125c}$$

$$= k_{\text{erp}}k_{\text{fps}}\theta_{\text{err}} \tag{4.125d}$$

$$= k_{\text{cor}}\theta_{\text{err}}. \tag{4.125e}$$

The direction of this correcting angular velocity is dictated by the $\vec{u}$-vector, since

$$\vec{\omega}_{\text{cor}} = ||\vec{\omega}_{\text{cor}}||\frac{\vec{u}}{||\vec{u}||} \tag{4.126a}$$

$$= k_{\text{cor}}\theta_{\text{err}}\frac{\vec{u}}{||\vec{u}||} \tag{4.126b}$$

$$= k_{\text{cor}}\theta_{\text{err}}\frac{\vec{u}}{\sin\theta_{\text{err}}}. \tag{4.126c}$$

In the last step we used that $\vec{s}_i^{\,\text{wcs}}$ and $\vec{s}_j^{\,\text{wcs}}$ are unit vectors, such that

$$||\vec{u}|| = ||\vec{s}_i^{\,\text{wcs}} \times \vec{s}_j^{\,\text{wcs}}|| = \sin\theta_{\text{err}}. \tag{4.127}$$

We expect the error to be small, so it is reasonable to use the small angle approximation, where $\theta_{\text{err}} \approx \sin\theta_{\text{err}}$, i.e.

$$\vec{\omega}_{\text{cor}} = k_{\text{cor}}\vec{u}. \tag{4.128}$$

We know that $\vec{u}$ is orthogonal to $\vec{s}_{\text{axis}}^{\,\text{wcs}}$ so we project it onto the vectors $\vec{t}_1^{\,\text{wcs}}$ and $\vec{t}_2^{\,\text{wcs}}$, and we end up with the error measure,

$$\vec{b}_{\text{hinge}} = k_{\text{cor}}\begin{bmatrix}(\vec{r}_j + \mathbf{R}(q_j)\vec{r}^{\,j}_{\text{anc}} - \vec{r}_i - \mathbf{R}(q_i)\vec{r}^{\,i}_{\text{anc}}) \\ \vec{t}_1^{\,\text{wcs}} \cdot \vec{u} \\ \vec{t}_2^{\,\text{wcs}} \cdot \vec{u}\end{bmatrix}. \tag{4.129}$$

4.9.3 Slider Joint

The slider joint only allows translation in a single direction as shown in Figure 4.12. Hence, there is only 1 DOF, and the Jacobian of the slider joint, J_{slider}, must be a 5-by-12 matrix,

$$J_{\text{slider}} = \begin{bmatrix}\mathbf{J}^i_{\text{lin}} & \mathbf{J}^i_{\text{ang}} & \mathbf{J}^j_{\text{lin}} & \mathbf{J}^j_{\text{ang}}\end{bmatrix}, \tag{4.130}$$

where $\mathbf{J}^i_{\text{lin}} \in \mathbb{R}^{5\times3}$, $\mathbf{J}^i_{\text{ang}} \in \mathbb{R}^{5\times3}$, $\mathbf{J}^j_{\text{lin}} \in \mathbb{R}^{5\times3}$, and $\mathbf{J}^j_{\text{ang}} \in \mathbb{R}^{5\times3}$. We will use the first three rows of the Jacobian to ensure that the two bodies connected by the slider joint do not rotate relative to each other. Hence, we require that they have identical angular velocity.

The last two rows of the Jacobian are used to make sure that the bodies only move relatively in the direction of the joint axis, $\vec{s}_{\text{axis}}^{\,\text{wcs}}$. This is done as follows: first we note the following relation between the bodies' linear velocities

$$\vec{v}_j = \vec{v}_i + \vec{\omega}_i \times \vec{c} + \vec{v}_{\text{slider}}, \tag{4.131}$$

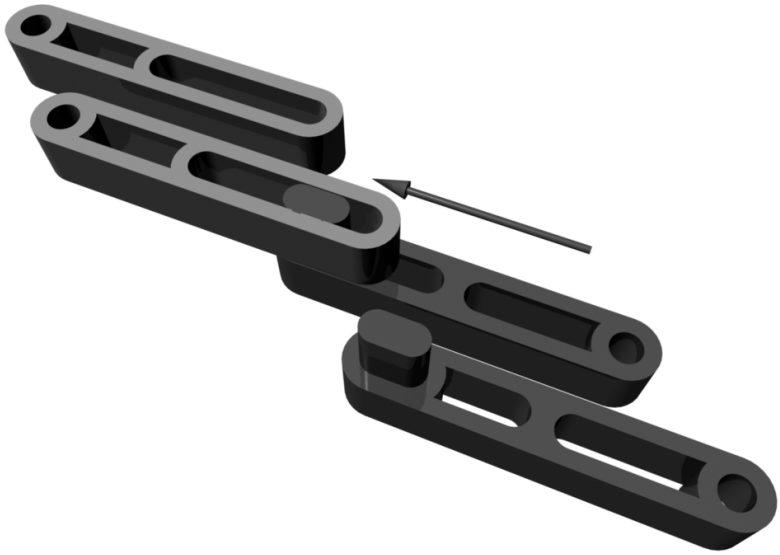


Figure 4.12: A slider joint example.

where $\vec{c} = \vec{r}_j - \vec{r}_i$, and $\vec{v}_{\text{slider}}$ is the joint velocity along the slider axis. Recalling that $\vec{\omega}_i = \vec{\omega}_j$, we can rewrite the velocity relation as follows,

$$\vec{v}_j = \vec{v}_i + \vec{\omega}_i \times \vec{c} + \vec{v}_{\text{slider}}, \tag{4.132a}$$

$$-\vec{v}_{\text{slider}} = \vec{v}_i - \vec{v}_j + \vec{\omega}_i \times \vec{c}, \tag{4.132b}$$

$$-\vec{v}_{\text{slider}} = \vec{v}_i - \vec{v}_j + \frac{\vec{\omega}_i + \vec{\omega}_j}{2} \times \vec{c}. \tag{4.132c}$$

From the joint axis, $\vec{s}_{\text{axis}}^{\text{wcs}}$, we can compute two orthogonal vectors $\vec{t}_1^{\text{wcs}}$, and $\vec{t}_2^{\text{wcs}}$. By the workings of a slider joint, we know that we may never have any relative velocities in the directions of the two vectors $\vec{t}_1^{\text{wcs}}$ and $\vec{t}_2^{\text{wcs}}$. That is,

$$0 = \vec{t}_1^{\text{wcs}} \cdot (-\vec{v}_{\text{slider}}), \tag{4.133a}$$

$$0 = \vec{t}_2^{\text{wcs}} \cdot (-\vec{v}_{\text{slider}}). \tag{4.133b}$$

From these two equations we can derive the remaining two rows in the Jacobian slider

matrix, and we find,

$$\mathbf{J}^i_{\text{lin}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \left(\vec{t}_1{}^{\text{wcs}}\right)^T \\ \left(\vec{t}_2{}^{\text{wcs}}\right)^T \end{bmatrix}, \tag{4.134a}$$

$$\mathbf{J}^j_{\text{lin}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ -\left(\vec{t}_1{}^{\text{wcs}}\right)^T \\ -\left(\vec{t}_2{}^{\text{wcs}}\right)^T \end{bmatrix}, \tag{4.134b}$$

$$\mathbf{J}^i_{\text{ang}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \frac{1}{2}\vec{c} \times \vec{t}_1{}^{\text{wcs}} \\ \frac{1}{2}\vec{c} \times \vec{t}_2{}^{\text{wcs}} \end{bmatrix}, \tag{4.134c}$$

$$\mathbf{J}^j_{\text{ang}} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ \frac{1}{2}\vec{c} \times \vec{t}_1{}^{\text{wcs}} \\ \frac{1}{2}\vec{c} \times \vec{t}_2{}^{\text{wcs}} \end{bmatrix}. \tag{4.134d}$$

Now we will look at the error term, $\vec{b}_{\text{slider}} \in \mathbb{R}^5$. The first three entries are used for rotational misalignment between the two links, such as sliding along a bend axis. The last two will be used for fixing parallel positional displacement of the joint axis.

As for the hinge-joint, we derived an angular velocity to correct the misalignment error of θ_{err} radians. The magnitude of this correcting angular velocity is as before,

$$\begin{aligned} ||\vec{\omega}_{\text{cor}}|| &= \frac{\theta_{\text{cor}}}{\Delta t} && (4.135a) \\ &= \frac{k_{\text{erp}}\theta_{\text{err}}}{\Delta t} && (4.135b) \\ &= k_{\text{erp}}\frac{1}{\Delta t}\theta_{\text{err}} && (4.135c) \\ &= k_{\text{erp}}k_{\text{fps}}\theta_{\text{err}} && (4.135d) \\ &= k_{\text{cor}}\theta_{\text{err}}. && (4.135e) \end{aligned}$$

As before, the direction of this correcting angular velocity is dictated by a rotation axis given by some unit $\vec{u}$-vector,

$$\begin{aligned} \vec{\omega}_{\text{cor}} &= ||\vec{\omega}_{\text{cor}}||\vec{u} && (4.136a) \\ &= k_{\text{cor}}\theta_{\text{err}}\vec{u}. && (4.136b) \end{aligned}$$

However, unlike previously, the correcting angular velocity will be derived as follows: let

the rotational misalignment be given by the quaternion, q_{err}, then we have

$$q_{\text{err}} = [s, \vec{v}]\,, \tag{4.137a}$$

$$q_{\text{err}} = \left[\cos\left(\frac{\theta_{\text{err}}}{2}\right), \sin\left(\frac{\theta_{\text{err}}}{2}\right)\vec{u}\right]. \tag{4.137b}$$

The error is expected to be small, so the small angle approximation is reasonable, and we find,

$$\frac{\theta_{\text{err}}}{2}\vec{u} \approx \sin\left(\frac{\theta_{\text{err}}}{2}\right)\vec{u} = \vec{v}. \tag{4.138}$$

Using this in our formula for the correcting angular velocity, we get

$$\vec{\omega}_{\text{cor}} = k_{\text{cor}} 2\vec{v}. \tag{4.139}$$

This will be the first three entries in the $\vec{b}_{slider}$-vector.

We can describe the current joint position by an offset vector, $\vec{r}_{\text{off}}^{\,\text{wcs}}$ which indicates the initial difference between the body centers that is

$$\vec{r}_{\text{off}}^{\,\text{j}} = \mathbf{R}(q_j)^T\left(\vec{r}_j - \vec{r}_i\right). \tag{4.140}$$

Observe that this offset vector is computed when the joint was set up initially, and it is constant. The corresponding offset in the world coordinate system is then simply found as,

$$\vec{r}_{\text{off}}^{\,\text{wcs}} = R(q_j)\vec{r}_{\text{off}}^{\,\text{j}}. \tag{4.141}$$

If there is no parallel displacement of the joint axis, then the vector, $\vec{c} - \vec{r}_{\text{off}}^{\,\text{wcs}}$, will have no components orthogonal to the joint axis. From this observation we have the last two entries in the vector $\vec{b}_{\text{slider}}$,

$$\vec{b}_{\text{slider}} = k_{\text{cor}}\begin{bmatrix} 2\vec{v} \\ \vec{t}_1^{\,\text{wcs}} \cdot (\vec{c} - \vec{r}_{\text{off}}^{\,\text{wcs}}) \\ \vec{t}_2^{\,\text{wcs}} \cdot (\vec{c} - \vec{r}_{\text{off}}^{\,\text{wcs}}) \end{bmatrix}. \tag{4.142}$$

4.9.4 Hinge-2 Joint

This kind of joint is also called a wheel-joint, because its motion resembles that of a turning front wheel on a car. We will therefore explain the workings of this joint type by the example of a car-wheel as shown in Figure 4.13.

The wheel-joint is the same as a series of two hinge joints. Its motion is described by a rotation axis, $\vec{s}_{\text{axis}_1}^{\,i}$, given by a unit vector in the body frame of body i, and another rotation axis, $\vec{s}_{\text{axis}_2}^{\,j}$, given as a unit vector in the body frame of body j.

In the following, we will implicitly assume that body i is the car and body j is the wheel. Using this convention, the axes are referred to as the steering axis or suspension axis and the motor axis.

We will also use an anchor point like before, where both axes go through this anchor point. We will assume that the axes do not lie along the same line, and that they are always separated by the initial angle, θ, between them.

Figure 4.13: A car wheel joint example.

From the description it is clear that the joint has 2 DOFs, from which we know that the wheel-joint Jacobian must have dimension 4-by-12,

$$J_{\text{wheel}} = \begin{bmatrix} \mathbf{J}^i_{\text{lin}} & \mathbf{J}^i_{\text{ang}} & \mathbf{J}^j_{\text{lin}} & \mathbf{J}^j_{\text{ang}} \end{bmatrix}, \tag{4.143}$$

where $\mathbf{J}^i_{\text{lin}} \in \mathbb{R}^{4\times 3}$, $\mathbf{J}^i_{\text{ang}} \in \mathbb{R}^{4\times 3}$, $\mathbf{J}^j_{\text{lin}} \in \mathbb{R}^{4\times 3}$, and $\mathbf{J}^j_{\text{ang}} \in \mathbb{R}^{4\times 3}$.

Following the same recipe as previously, we re-use the ball-in-socket joint for the positional constraints, and we are now only left with the fourth row in the Jacobian matrix.

Let us compute the joint axis in the world coordinate system,

$$\vec{s}_i^{\,\text{wcs}} = \mathbf{R}(q_i)\vec{s}^{\,i}_{\text{axis}}, \tag{4.144a}$$

$$\vec{s}_j^{\,\text{wcs}} = \mathbf{R}(q_j)\vec{s}^{\,j}_{\text{axis}}, \tag{4.144b}$$

then the constrained rotational DOF is dictated by a rotational axis orthogonal to the two rotation axes. That is,

$$\vec{u} = \vec{s}_i^{\,\text{wcs}} \times \vec{s}_j^{\,\text{wcs}}. \tag{4.145}$$

For the hinge to keep its alignment, we must ensure that there is no relative rotation around this axis,

$$\vec{u} \cdot \vec{\omega}_i - \vec{u} \cdot \vec{\omega}_j = 0. \tag{4.146}$$

This gives us the missing fourth row of the Jacobian matrix,

$$\mathbf{J}^i_{\text{lin}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \tag{4.147a}$$

$$\mathbf{J}^j_{\text{lin}} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix}, \tag{4.147b}$$

$$\mathbf{J}^i_{\text{ang}} = \begin{bmatrix} \left(\mathbf{R}(\mathbf{q_i})\tilde{\mathbf{r}}^{\mathbf{i}}_{\text{anc}}\right)^{\times} \\ \vec{u}^T \end{bmatrix}, \tag{4.147c}$$

$$\mathbf{J}^j_{\text{ang}} = -\begin{bmatrix} \left(\mathbf{R}(\mathbf{q_j})\tilde{\mathbf{r}}^{\mathbf{j}}_{\text{anc}}\right)^{\times} \\ \vec{u}^T \end{bmatrix}. \tag{4.147d}$$

From the ball-in-socket joint we also have the first three entries of the error term, $\vec{b}_{\text{wheel}} \in \mathbb{R}^4$. Therefore, we only need to come up with the fourth entry to re-establish the angle θ between the two joint axes. Let us say that the current angle is given by ϕ. Then we need a correcting angular velocity of magnitude,

$$||\vec{\omega}_{\text{cor}}|| = \frac{\theta_{\text{cor}}}{\Delta t} \tag{4.148a}$$

$$= \frac{k_{\text{erp}}\,(\theta - \phi)}{\Delta t} \tag{4.148b}$$

$$= k_{\text{erp}} \frac{1}{\Delta t}\,(\theta - \phi) \tag{4.148c}$$

$$= k_{\text{erp}} k_{\text{fps}}\,(\theta - \phi) \tag{4.148d}$$

$$= k_{\text{cor}}\,(\theta - \phi)\,. \tag{4.148e}$$

We can now write the error-term vector as,

$$\vec{b}_{\text{wheel}} = k_{\text{cor}} \begin{bmatrix} \vec{b}_{\text{ball}} \\ (\theta - \phi) \end{bmatrix}. \tag{4.149}$$

Finally, two more tricks are possible: firstly, one rotates the axes of the ball-in-socket joints, such that the first constraining axis is along the suspension axis. This allows one to model suspension by modulating the translational error in the ball-in-socket joint along its first axis. Secondly, a small angle approximation for the fourth entry in the error term vector may be used. We refer to the source code of the Open Dynamics Engine [112] for further details.

4.9.5 Universal Joint

The universal joint is in some sense similar to the wheel-joint, and is described similarly by two joint axes,

$$\vec{s}^{\,i}_{\text{axis}_1}, \tag{4.150a}$$

$$\vec{s}^{\,j}_{\text{axis}_2}, \tag{4.150b}$$

and an anchor point which the two axes run through. The difference from the wheel joint is that it is further required that axis 2 makes an angle of $\pi/2$ with axis 1. An example of an universal joint is shown in Figure 4.14.

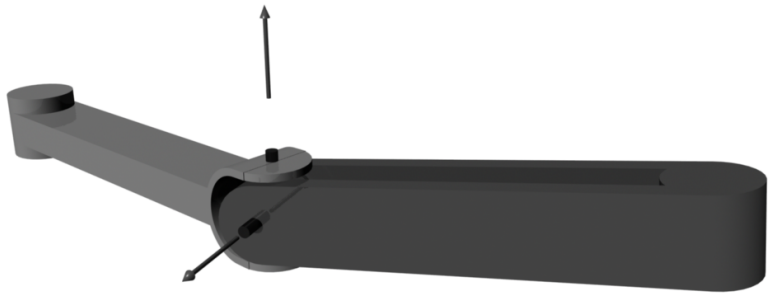


Figure 4.14: A universal joint example.

We notice that we have 2 DOFs, and the Jacobian matrix of the universal joint, $J_{\text{universal}}$, must be a 4-by-12 matrix,

$$J_{\text{universal}} = \begin{bmatrix} \mathbf{J}^i_{\text{lin}} & \mathbf{J}^i_{\text{ang}} & \mathbf{J}^j_{\text{lin}} & \mathbf{J}^j_{\text{ang}} \end{bmatrix}. \tag{4.151}$$

Since this joint type has derivations almost identical to previous types, we will ease on notation and go through the steps faster. We start out by reusing the ball-in-socket joint for the positional constraints, and then we compute the constrained rotation axis,

$$\vec{u} = \vec{s}_i^{\,\text{wcs}} \times \vec{s}_j^{\,\text{wcs}}, \tag{4.152}$$

along which we know there must be no relative angular velocity. We can now write the

Jacobian matrix for the universal joint as,

$$\mathbf{J}^i_{\text{lin}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \tag{4.153a}$$

$$\mathbf{J}^j_{\text{lin}} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix}, \tag{4.153b}$$

$$\mathbf{J}^i_{\text{ang}} = \begin{bmatrix} \left(\mathbf{R}(\mathbf{q_i})\tilde{\mathbf{r}}^{\mathbf{i}}_{\text{anc}}\right)^{\times} \\ \vec{u}^T \end{bmatrix}, \tag{4.153c}$$

$$\mathbf{J}^j_{\text{ang}} = -\begin{bmatrix} \left(\mathbf{R}(\mathbf{q_j})\tilde{\mathbf{r}}^{\mathbf{j}}_{\text{anc}}\right)^{\times} \\ \vec{u}^T \end{bmatrix}. \tag{4.153d}$$

We already have the first three entries of the vector, $\vec{b}_{\text{universal}}$, and we must find the fourth. We do this by first looking at the magnitude of the correcting angular velocity,

$$||\vec{\omega}_{\text{cor}}|| = \frac{\theta_{\text{cor}}}{\Delta t} \tag{4.154a}$$

$$= \frac{k_{\text{erp}}\left(\phi - \frac{\pi}{2}\right)}{\Delta t} \tag{4.154b}$$

$$= k_{\text{cor}}\left(\phi - \frac{\pi}{2}\right), \tag{4.154c}$$

where ϕ denotes the current angle between the two joint axes. If ϕ is close to $\pi/2$, then

$$\phi - \frac{\pi}{2} \approx \cos(\phi) \tag{4.155a}$$

$$= \vec{s}_i{}^{\text{ wcs}} \cdot \vec{s}_j{}^{\text{ wcs}}. \tag{4.155b}$$

We can now write the error term vector as,

$$\vec{b}_{\text{universal}} = k_{\text{cor}} \begin{bmatrix} \vec{b}_{\text{ball}} \\ -\vec{s}_i{}^{\text{ wcs}} \cdot \vec{s}_j{}^{\text{ wcs}} \end{bmatrix}. \tag{4.156}$$

4.9.6 Fixed Joint

For fixed joints, we know that it constrains two bodies completely from any relative movement, and therefore it has 0 DOFs, from which we know that the Jacobian matrix, $J_{\text{fixed}} \in \mathbb{R}^{6\times 12}$,

$$J_{\text{universal}} = \begin{bmatrix} \mathbf{J}^i_{\text{lin}} & \mathbf{J}^i_{\text{ang}} & \mathbf{J}^j_{\text{lin}} & \mathbf{J}^j_{\text{ang}} \end{bmatrix}. \tag{4.157}$$

The fixed joint is described by an anchor point, and initially we compute an offset vector, and store it in the body frame of body i,

$$\vec{r}_{\text{off}}{}^{\text{ i}} = \vec{r}_i - \vec{r}_j. \tag{4.158}$$

Observe that this offset vector is computed when the joint was set up initially, and it is a constant. The corresponding offset in the world coordinate system is then found by,

$$\vec{r}_{\text{off}}^{\,\text{wcs}} = R(q_i)\vec{r}_{\text{off}}^{\,\text{i}}. \tag{4.159}$$

Since we have a fixed joint, both incident bodies must be rotating with the same angular velocity, and the linear velocities must obey the relation,

$$\vec{v}_j = \vec{v}_i + \vec{\omega}_i \times \vec{r}_{\text{off}}^{\,\text{wcs}}. \tag{4.160}$$

From all this we can now set up the Jacobian matrix as,

$$\mathbf{J}^i_{\text{lin}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \tag{4.161a}$$

$$\mathbf{J}^j_{\text{lin}} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \tag{4.161b}$$

$$\mathbf{J}^i_{\text{ang}} = \begin{bmatrix} & -(\tilde{\mathbf{r}}_{\text{off}}^{\,\text{wcs}})^{\times} & \\ & & \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{4.161c}$$

$$\mathbf{J}^j_{\text{ang}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}. \tag{4.161d}$$

Similarly, the error term is straightforward. To correct the positional error we use the first three entries from the ball-in-socket joint, $\vec{b}_{\text{ball}}$. From the slider joint, $\vec{b}_{\text{slider}}$, we may re-use the first three entries to take care of any rotational errors. We can now write the $\vec{b}_{\text{fixed}}$ vector as,

$$\vec{b}_{\text{fixed}} = k_{\text{cor}} \begin{bmatrix} (\vec{r}_i + \mathbf{R}(q_i)\vec{r}_{\text{anc}}^{\,i} - \vec{r}_j - \mathbf{R}(q_j)\vec{r}_{\text{anc}}^{\,j}) \\ 2\vec{v} \end{bmatrix}, \tag{4.162}$$

where $\vec{v}$ comes from the quaternion representing the rotational error,

$$q_{\text{err}} = [s, \vec{v}]\,. \tag{4.163}$$

4.9.7 Contact Point

As we have explained, contact constraints are completely different from joint constraints, but they too are described by a Jacobian matrix. We will show that this Jacobian matrix, J_{contact}, can be expressed in the same sub matrix pattern as the joint Jacobians, and one can even construct an error correcting term.

From previously we know that the contact Jacobian has $1+\eta$ constraints, so it is a $(1+\eta)$-by-12 dimensional matrix,

$$J_{\text{contact}} = \begin{bmatrix} \mathbf{J}^i_{\text{lin}} & \mathbf{J}^i_{\text{ang}} & \mathbf{J}^j_{\text{lin}} & \mathbf{J}^j_{\text{ang}} \end{bmatrix}. \tag{4.164}$$

The first row corresponds to the normal force constraints and the remaining η rows correspond to the tangential friction constraints, i.e.

$$\mathbf{J}^i_{\text{lin}} = \begin{bmatrix} -\vec{n}^t \\ -\mathbf{D}_k^T \end{bmatrix}, \tag{4.165a}$$

$$\mathbf{J}^j_{\text{lin}} = \begin{bmatrix} \vec{n}^t \\ \mathbf{D}_k^T \end{bmatrix}, \tag{4.165b}$$

$$\mathbf{J}^i_{\text{ang}} = \begin{bmatrix} -\left(\tilde{\mathbf{r}}^\times_{\mathbf{i}}\vec{n}\right)^T \\ -\left(\tilde{\mathbf{r}}^\times_{\mathbf{i}}\mathbf{D}_k\right)^T \end{bmatrix}, \tag{4.165c}$$

$$\mathbf{J}^j_{\text{ang}} = \begin{bmatrix} \left(\tilde{\mathbf{r}}^\times_{\mathbf{j}}\vec{n}\right)^T \\ \left(\tilde{\mathbf{r}}^\times_{\mathbf{j}}\mathbf{D}_k\right)^T \end{bmatrix}. \tag{4.165d}$$

If the penetration constraints are violated then an error correcting vector, $\vec{b}_{\text{contact}} \in \mathbb{R}^{1+\eta}$, can be used as,

$$\vec{b}_{\text{contact}} = k_{\text{cor}} \begin{bmatrix} d_{\text{penetration}} \\ \vec{0} \end{bmatrix}. \tag{4.166}$$

where $d_{\text{penetration}}$ is the penetration depth. These observations regarding the sub matrix patterns of the Jacobians of both the contact and joint constraints, allow us to implement these kind of constraints using almost the same kind of data structure.

4.10 Joint Limits

It is not always enough just to setup a joint: even though the relative motion is constrained to only move in a consistent manner wrt. the joint, other constraints need our attention. For instance, in the real world we can not find a jointed mechanism with a sliding joint that has an infinitely long joint axis. In other words, we need some way to model the extent of a sliding joint. We will do this modeling by setting up joint limits. To be specific, we will treat joint limits on a sliding joint and a hinge joint.

The general approach we will take here for introducing joint limits is very similar to the way we use unilateral constraints for enforcing normal non-penetration constraints at the contact points. In a sense, setting joint limits this way is nothing more than a slightly exotic way of computing contact points for normal force constraints disguised as joint limits.

4.10.1 Slider Joint Limits

Recall that in specifying the slider joint we used an offset vector, which was the initial difference between the origin of the body frames. That is, at time $t = 0$, we compute

$$\vec{r}_{\text{off}}^{\,\text{j}} = \mathbf{R}(q_j)^T \left(\vec{r}_j - \vec{r}_i\right). \tag{4.167}$$

As before, the initial offset wrt. the bodies' current location in the world coordinate system is computed as,

$$\vec{r}_{\text{off}}^{\,\text{wcs}} = R(q_j)\vec{r}_{\text{off}}^{\,\text{j}}. \tag{4.168}$$

Now letting $\vec{c} = \vec{r}_j - \vec{r}_i$, we can compute the current displacement, $\vec{r}_{\text{dis}}$, along the joint axis as,

$$\vec{r}_{\text{dis}} = \vec{c} - \vec{r}_{\text{off}}^{\,\text{wcs}}. \tag{4.169}$$

Taking the dot-product with the joint axis, $\vec{s}_{\text{axis}}^{\,\text{wcs}}$, gives a signed distance measure of the displacement along the joint axis,

$$d_{\text{dis}} = \vec{s}_{\text{axis}}^{\,\text{wcs}} \cdot \vec{r}_{\text{dis}}. \tag{4.170}$$

When the joint limits are imposed on the slider joint, we want to be able to specify a lower distance limit, d_{lo}, and and upper distance limit, d_{hi}. If one of these are violated, for instance

$$d_{\text{dis}} \leq d_{\text{lo}}, \tag{4.171}$$

then we will add a new unilateral constraint. This constraint specifies that the relative velocity of the joint along the joint axis must be such that the displacement does not move beyond the limit. For a slider joint that means we require,

$$\vec{s}_{\text{axis}}^{\,\text{wcs}} \cdot \left(\vec{v}_j - \vec{v}_i\right) \geq 0. \tag{4.172}$$

We immediately see that this is equivalent to a 1-by-12 dimensional Jacobian matrix, $J_{\text{slider}}^{\text{lo}}$, where

$$\mathbf{J}_{\text{lin}}^{i} = -\left(\vec{s}_{\text{axis}}^{\,\text{wcs}}\right)^T, \tag{4.173a}$$

$$\mathbf{J}_{\text{lin}}^{j} = \left(\vec{s}_{\text{axis}}^{\,\text{wcs}}\right)^T, \tag{4.173b}$$

$$\mathbf{J}_{\text{ang}}^{i} = \vec{0}, \tag{4.173c}$$

$$\mathbf{J}_{\text{ang}}^{j} = \vec{0}. \tag{4.173d}$$

However, this would be the wrong Jacobian to use, because when we look at the reaction forces from the joint limit, this Jacobian will not model the torques coming from the limiting force because only the linear contribution is included.

We will now remodel the Jacobian to include all the force and torque contributions. Let us say that the position of the joint limit is given by the vectors $\vec{r}_{\text{lim}_i}^{\,\text{wcs}}$ and $\vec{r}_{\text{lim}_j}^{\,\text{wcs}}$. These vectors are specified in the world coordinate system and go from their respective body centers to the position of the joint limit. Now, say that some force, $\vec{F}$, is acting on body i at the joint limit. Then the force is required to be parallel with the joint axis $\vec{s}_{\text{axis}}^{\,\text{wcs}}$. The force contribution to body j is $-\vec{F}$ according to Newton's third law of motion. The limit force also results in a torque on body i,

$$\vec{\tau}_{\text{lim}_i} = \vec{r}_{\text{lim}_i}^{\,\text{wcs}} \times \vec{F}, \tag{4.174}$$

and a torque on body j,

$$\vec{\tau}_{\mathrm{lim}_j} = -\vec{r}_{\mathrm{lim}_j}^{\,\mathrm{wcs}} \times \vec{F}. \tag{4.175}$$

For a slider joint, we must also require that these torques do not induce a relative angular velocity of the two bodies, meaning that angular momentum should remain unchanged. From Euler's equation we get,

$$\vec{\tau}_{\mathrm{lim}_i} + \vec{\tau}_{\mathrm{lim}_j} = 0. \tag{4.176}$$

Recalling that the corresponding reaction force is given by,

$$\vec{F}_{\mathrm{slider}}^{\,\mathrm{lo}} = \left(J_{\mathrm{slider}}^{\,\mathrm{lo}}\right)^T \lambda_{\mathrm{lo}}, \tag{4.177}$$

where λ_{lo} is a non-negative Lagrange multiplier. This suggests that the Jacobian should look like,

$$\mathbf{J}_{\mathrm{lin}}^i = -\left(\vec{s}_{\mathrm{axis}}^{\,\mathrm{wcs}}\right)^T, \tag{4.178a}$$

$$\mathbf{J}_{\mathrm{lin}}^j = \left(\vec{s}_{\mathrm{axis}}^{\,\mathrm{wcs}}\right)^T, \tag{4.178b}$$

$$\mathbf{J}_{\mathrm{ang}}^i = \left(\vec{r}_{\mathrm{lim}_i}^{\,\mathrm{wcs}} \times \vec{s}_{\mathrm{axis}}^{\,\mathrm{wcs}}\right)^T, \tag{4.178c}$$

$$\mathbf{J}_{\mathrm{ang}}^j = -\left(\vec{r}_{\mathrm{lim}_j}^{\,\mathrm{wcs}} \times \vec{s}_{\mathrm{axis}}^{\,\mathrm{wcs}}\right)^T. \tag{4.178d}$$

However, it is not obvious that $\mathbf{J}_{\mathrm{ang}}^i\vec{\omega}_i + \mathbf{J}_{\mathrm{ang}}^j\vec{\omega}_j = 0$. Further, to avoid computing the vectors $\vec{r}_{\mathrm{lim}_i}^{\,\mathrm{wcs}}$ and $\vec{r}_{\mathrm{lim}_j}^{\,\mathrm{wcs}}$ during simulation, it would be nice to remove them from the expressions.

Observe that the vector, $\vec{c} = \vec{r}_j - \vec{r}_i$, can be written as, $\vec{c} = \vec{r}_{\mathrm{lim}_i}^{\,\mathrm{wcs}} - \vec{r}_{\mathrm{lim}_j}^{\,\mathrm{wcs}}$. We will now show that with the $\vec{c}$-vector we can obtain:

$$\vec{r}_{\mathrm{lim}_i}^{\,\mathrm{wcs}} \times \vec{F} = \frac{1}{2}\vec{c} \times \vec{F} = \frac{1}{2}\left(\vec{r}_{\mathrm{lim}_i}^{\,\mathrm{wcs}} - \vec{r}_{\mathrm{lim}_j}^{\,\mathrm{wcs}}\right) \times \vec{F}, \tag{4.179a}$$

$$\vec{r}_{\mathrm{lim}_j}^{\,\mathrm{wcs}} \times \vec{F} = -\frac{1}{2}\vec{c} \times \vec{F} = -\frac{1}{2}\left(\vec{r}_{\mathrm{lim}_i}^{\,\mathrm{wcs}} - \vec{r}_{\mathrm{lim}_j}^{\,\mathrm{wcs}}\right) \times \vec{F}. \tag{4.179b}$$

From the second equation we have

$$\vec{r}_{\mathrm{lim}_j}^{\,\mathrm{wcs}} \times \vec{F} = -\frac{1}{2}\vec{r}_{\mathrm{lim}_i}^{\,\mathrm{wcs}} \times \vec{F} + \frac{1}{2}\vec{r}_{\mathrm{lim}_j}^{\,\mathrm{wcs}} \times \vec{F}, \tag{4.180}$$

which yields

$$\frac{1}{2}\vec{r}_{\mathrm{lim}_j}^{\,\mathrm{wcs}} \times \vec{F} = -\frac{1}{2}\vec{r}_{\mathrm{lim}_i}^{\,\mathrm{wcs}} \times \vec{F}, \tag{4.181}$$

and substituting this into the first equation yields

$$\vec{r}_{\mathrm{lim}_i}^{\,\mathrm{wcs}} \times \vec{F} = \frac{1}{2}\left(\vec{r}_{\mathrm{lim}_i}^{\,\mathrm{wcs}} - \vec{r}_{\mathrm{lim}_j}^{\,\mathrm{wcs}}\right) \times \vec{F} \tag{4.182a}$$

$$= \frac{1}{2}\vec{r}_{\mathrm{lim}_i}^{\,\mathrm{wcs}} \times \vec{F} - \frac{1}{2}\vec{r}_{\mathrm{lim}_j}^{\,\mathrm{wcs}} \times \vec{F} \tag{4.182b}$$

$$= \frac{1}{2}\vec{r}_{\mathrm{lim}_i}^{\,\mathrm{wcs}} \times \vec{F} - \left(-\frac{1}{2}\vec{r}_{\mathrm{lim}_i}^{\,\mathrm{wcs}} \times \vec{F}\right), \tag{4.182c}$$

$$\vec{r}_{\mathrm{lim}_i}^{\,\mathrm{wcs}} \times \vec{F} = \vec{r}_{\mathrm{lim}_i}^{\,\mathrm{wcs}} \times \vec{F}. \tag{4.182d}$$

This proves (4.179a). Repeating the steps but interchanging equations easily derives (4.179b). Observe also that the sum of the two equations is equal to zero as required by Euler's equation. We can now rewrite the angular parts of the Jacobian as,

$$\mathbf{J}^i_{\text{lin}} = -\left(\vec{s}^{\,\text{wcs}}_{\text{axis}}\right)^T, \tag{4.183a}$$

$$\mathbf{J}^j_{\text{lin}} = \left(\vec{s}^{\,\text{wcs}}_{\text{axis}}\right)^T, \tag{4.183b}$$

$$\mathbf{J}^i_{\text{ang}} = \frac{1}{2}\left(\vec{c}\times\vec{s}^{\,\text{wcs}}_{\text{axis}}\right)^T, \tag{4.183c}$$

$$\mathbf{J}^j_{\text{ang}} = -\frac{1}{2}\left(\vec{c}\times\vec{s}^{\,\text{wcs}}_{\text{axis}}\right)^T. \tag{4.183d}$$

To verify that $\mathbf{J}^i_{\text{ang}}\vec{\omega}_i + \mathbf{J}^j_{\text{ang}}\vec{\omega}_j = 0$, we insert (4.183c) and (4.183d) and find,

$$\frac{1}{2}\left(\vec{c}\times\vec{s}^{\,\text{wcs}}_{\text{axis}}\right)^T\vec{\omega}_i - \frac{1}{2}\left(\vec{c}\times\vec{s}^{\,\text{wcs}}_{\text{axis}}\right)^T\vec{\omega}_j = 0, \tag{4.184a}$$

$$\Rightarrow \quad \frac{1}{2}\left(\vec{c}\times\vec{s}^{\,\text{wcs}}_{\text{axis}}\right)^T\left(\vec{\omega}_i - \vec{\omega}_j\right) = 0, \tag{4.184b}$$

and because we have a slider joint $\vec{\omega}_i = \vec{\omega}_j$. In conclusion, we see that with the Jacobian in (4.184) both the kinematic constraint are satisfied, and the reaction forces are proper.

Putting it all together we have the complementarity constraint,

$$J^{\text{lo}}_{\text{slider}}\vec{u} \geq 0, \quad \text{compl. to} \quad \lambda_{\text{lo}} \geq 0. \tag{4.185}$$

An error correcting term, $\vec{b}^{\,\text{lo}}_{\text{slider}}$ is easily added to the right side of the kinematic constraint as,

$$\vec{b}^{\,\text{lo}}_{\text{slider}} = k_{\text{erp}}\frac{d_{\text{lo}} - d_{\text{dis}}}{\Delta t} \tag{4.186a}$$

$$= k_{\text{cor}}d_{\text{err}}, \tag{4.186b}$$

where we have set $d_{\text{err}} = d_{\text{lo}} - d_{\text{dis}}$.

In conclusion, we have derived a single linear complementarity constraint for the lower joint limit of a slider joint. The same approach can be used to derive a single linear complementarity constraint for the upper limit. It should be apparent that all that is really needed is to negate the Jacobian, i.e.

$$J^{\text{hi}}_{\text{slider}} = -J^{\text{lo}}_{\text{slider}}, \tag{4.187}$$

and for the error term

$$\vec{b}^{\,\text{hi}}_{\text{slider}} = k_{\text{erp}}\frac{d_{\text{hi}} - d_{\text{dis}}}{\Delta t} \tag{4.188a}$$

$$= k_{\text{cor}}d_{\text{err}}. \tag{4.188b}$$

4.10.2 Hinge Joint Limits

There is not much difference between setting limits on a slider joint and a hinge joint. The major difference is that the joint axis now describes a rotation axis, and instead of distances we use angle measures.

If we store the initial relative rotation of two bodies in the quaternion, q_{ini}, then we can compute the current relative rotation of the two bodies as,

$$q_{\text{rel}} = q_j^* q_i q_{\text{ini}}^*. \tag{4.189}$$

This quaternion corresponds to a rotation of θ radians around the unit axis, $\vec{v}$, i.e. the angle measured from i to j around the joint axis,

$$q_{\text{rel}} = \left[\cos\left(\frac{\theta}{2}\right), \sin\left(\frac{\theta}{2}\right)\vec{v}\right]. \tag{4.190}$$

Using standard trigonometry and taking care of the double representation of rotations by picking the smallest angle solution, one can extract the angle from the quaternion of relative rotation. Specifically taking the dot product of the vector part of the quaternion with itself, $\sin\frac{\theta}{2}$, is obtained, since $\vec{v}\cdot\vec{v} = 1$, and arctan can be used to obtain $\frac{\theta}{2}$.

As we did in the case of the slider joint, we want to impose a low and high joint limit, θ_{lo} and θ_{hi}. If, for instance, the lower limit is violated as,

$$\theta \leq \theta_{\text{lo}}, \tag{4.191}$$

then we will add a new unilateral constraint which specifies that the relative angular velocity of the joint around the joint axis must be such that the angle does not move beyond the limit. For a hinge joint this means that we must require,

$$\vec{s}_{\text{axis}}^{\text{ wcs}} \cdot (\vec{\omega}_j - \vec{\omega}_i) \geq 0. \tag{4.192}$$

We see immediately that this is equivalent to a 1-by-12 dimensional Jacobian matrix, $J_{\text{hinge}}^{\text{ lo}}$, where

$$\mathbf{J}_{\text{lin}}^i = \vec{0}, \tag{4.193a}$$
$$\mathbf{J}_{\text{lin}}^j = \vec{0}, \tag{4.193b}$$
$$\mathbf{J}_{\text{ang}}^i = -\left(\vec{s}_{\text{axis}}^{\text{ wcs}}\right)^T, \tag{4.193c}$$
$$\mathbf{J}_{\text{ang}}^j = \left(\vec{s}_{\text{axis}}^{\text{ wcs}}\right)^T. \tag{4.193d}$$

The corresponding reaction force from the joint limit is given by,

$$\vec{F}_{\text{hinge}}^{\text{ lo}} = \left(J_{\text{hinge}}^{\text{ lo}}\right)^T \lambda_{\text{lo}}, \tag{4.194}$$

where λ_{lo} is a non-negative Lagrange multiplier. We see that we have the complementarity constraint,

$$J_{\text{hinge}}^{\text{ lo}}\vec{u} \geq 0, \quad \text{compl. to} \quad \lambda_{\text{lo}} \geq 0. \tag{4.195}$$

An error correcting term, $\vec{b}_{\text{hinge}}^{\text{ lo}}$, is added to the right side of the kinematic constraint as,

$$\vec{b}_{\text{hinge}}^{\text{ lo}} = k_{\text{erp}}\frac{\theta_{\text{lo}} - \theta}{\Delta t} \tag{4.196a}$$
$$= k_{\text{cor}}\theta_{\text{err}}, \tag{4.196b}$$

where $\theta_{\text{err}} = \theta_{\text{lo}} - \theta$.

The constraints for the high limit of the hinge joint is obtained by negating the Jacobian,

$$J_{\text{hinge}}^{\text{hi}} = -J_{\text{hinge}}^{\text{lo}}, \tag{4.197}$$

and the error term is given by,

$$\vec{b}_{\text{hinge}}^{\text{hi}} = k_{\text{erp}} \frac{\theta_{\text{hi}} - \theta}{\Delta t} \tag{4.198a}$$

$$= k_{\text{cor}} \theta_{\text{err}}. \tag{4.198b}$$

4.10.3 Generalization of Joint Limits

In the previous sections we derived constraint equations specific for low and high joint limits on slider and hinge joints. Fortunately, it is possible to extend these ideas to a more general framework. For instance, the concept of a reach cone, i.e. multiple angular joint limits, is often used in biomechanics to describe the limited movement of the shoulder or hip joints in the human skeleton [152].

In fact, we could formulate the allowable configuration space or the reachable region for a joint by an implicit function, $C(\ldots) \in \mathbb{R}$, of the joint parameters. For the slider and hinge joints, the joint parameters are the displacement and the angle which we will specify with the generalized joint parameter vector, $\vec{q}$, as a function of the generalized position vector, $\vec{s}$. Furthermore, the implicit function has the following characteristics,

$$C(\vec{q}(\vec{s})) < 0 \quad \text{Outside}, \tag{4.199a}$$

$$C(\vec{q}(\vec{s})) = 0 \quad \text{On the boundary}, \tag{4.199b}$$

$$C(\vec{q}(\vec{s})) > 0 \quad \text{Inside}. \tag{4.199c}$$

We can now reformulate positional constraints as,

$$C(\vec{q}(\vec{s})) \geq 0. \tag{4.200}$$

Differentiation wrt. time leads to the kinematic constraint,

$$\frac{d}{dt} C(\vec{q}(\vec{s})) = \underbrace{\frac{dC(\vec{q}(\vec{s}))}{d\vec{q}} \frac{d\vec{q}}{d\vec{s}}}_{J_C} \underbrace{\frac{d\vec{s}}{dt}}_{\vec{u}} \tag{4.201a}$$

$$= J_C \vec{u} \tag{4.201b}$$

$$\geq 0, \tag{4.201c}$$

which could be augmented with an error reduction term,

$$J_C \vec{u} \geq \vec{b}_C. \tag{4.202}$$

The reaction forces are determined by,

$$\vec{F}_{\text{reaction}}^{C} = J_C^T \vec{\lambda}_C, \tag{4.203}$$

where $\vec{\lambda}_C$ is a vector of non-negative Lagrange multipliers. Finally, we have the complementarity constraints,

$$J_C \vec{u} - \vec{b}_C \geq 0, \quad \text{compl. to} \quad \vec{\lambda}_C \geq 0. \tag{4.204}$$

The constraints in terms of the Jacobian and the error term should be added to the system equations whenever a joint limit has been violated or its boundary has been reached. This is completely analogous to collision detection, and enforcing joint limits in this manner is therefore not much different from finding contacts and computing normal forces.

Observe that in an acceleration-based formulation, the kinematic joint limit constraints should be differentiated wrt. time to get the acceleration constraints needed for augmenting the system equations.

4.11 Joint Motors

With joints and joint limits we are capable of modeling the range of relative motion between two bodies, and we will now look at one way to control the motion that is taking place.

A joint motor applies torque or force to a joint's degrees of freedom to induce movement. The joint motor model uses two parameters for this: A desired speed, v_{desired}, and the maximum torque or force, $\lambda_{\max}$, that can be applied to reach the desired speed.

From past sections we have seen that the error correcting term can be used to adjust velocities. The same principle can be used to drive a joint towards a desired speed by,

$$J_{\text{motor}}\vec{u} \geq \vec{b}_{\text{motor}}, \tag{4.205a}$$

$$\begin{bmatrix}\mathbf{J}^i_{\text{lin}} & \mathbf{J}^i_{\text{ang}} & \mathbf{J}^j_{\text{lin}} & \mathbf{J}^j_{\text{ang}}\end{bmatrix}\begin{bmatrix}\vec{v}_i \\ \vec{\omega}_i \\ \vec{v}_j \\ \vec{\omega}_j\end{bmatrix} \geq \vec{b}_{\text{motor}}. \tag{4.205b}$$

For a 1 DOF joint like a slider and a hinge joint, the motor Jacobian will have dimension 1-by-12, and the right hand side will be a scalar. In fact:

$$\vec{b}_{\text{motor}} = v_{\text{desired}}. \tag{4.206}$$

The Jacobian is also easy to derive for these two cases

$$J_{\text{motor}}^{\text{slider}} = \left[\vec{s}_{\text{axis}}^{\text{wcs}}, \vec{0}, -\vec{s}_{\text{axis}}^{\text{wcs}}, \vec{0}\right], \tag{4.207a}$$

$$J_{\text{motor}}^{\text{hinge}} = \left[\vec{0}, -\vec{s}_{\text{axis}}^{\text{wcs}}, \vec{0}, \vec{s}_{\text{axis}}^{\text{wcs}}\right]. \tag{4.207b}$$

The motor force is given by the relation,

$$\vec{F}_{\text{motor}} = J^T_{\text{motor}}\lambda_{\text{motor}}, \tag{4.208}$$

where λ_{motor} is a Lagrange multiplier that can be interpreted as a measure of the magnitude of the joint force along the degrees of freedom in the joint. By setting upper and lower limits on λ_{motor} we can model the aspect of the maximum available force, that is,

$$-\lambda_{\max} \leq \lambda_{\text{motor}} \leq \lambda_{\max}. \tag{4.209}$$

Finally, we require the force and the desired velocity term to be complementary to each other, i.e.

$$J_{\text{motor}}\vec{u} \geq \vec{b}_{\text{motor}}, \quad \text{compl. to} \quad |\lambda_{\text{motor}}| \leq \lambda_{\max}. \tag{4.210}$$

The basic idea behind this is that if the velocity exceeds the desired velocity then the motor force should work at its maximum to bring the velocity back to the desired speed.

On the other hand, if the desired speed has been reached, then the motor force can assume any value between 0 and $|\lambda_{\max}|$ to keep the desired speed.

In conclusion, we have developed linear complementarity constraints with both upper and lower limits for joint motors. The theory we have outlined can be extended to more complex joint types in a straightforward manner, simply by formulating Jacobians for their degree of freedom.

Another aspect of joint motors which comes in handy, is that they can be used to model friction in joints. This is done by setting the desired velocity to zero and the maximum force to some constant value. Then all joint motion will be slowed down by the "frictional" motor force.

To drive the joints to a specified position, a positional joint motor would simply be a hybrid of the joint limit model and the joint motor models we have outlined. The trick lies in setting the lower and upper joint limits equal to the wanted position, and then limiting the motor force as we did in this section.

In contrast to real life motors and engines, the presented motor controls do capture the essential idea of limited power and the need for controlling the speed. Higher level controllers could be added to a simulator (motor programs) for manipulating the joint motors, but this is out of the scope of this dissertation.

4.12 Time-Stepping Methods

In order to calculate the movement of rigid bodies, the simulation loop needs to advance the simulation time. This process is called a time-stepping method or time control. Knowing how to compute contact and constraint forces or impulses at collisions, the time-stepping method sets up a scheme that integrates the forces in order to obtain the motion of the rigid bodies. In the following, we will mainly discuss fixed time-stepping methods.

The matrices $\mathbf{M}$, $\mathbf{C}$, $\mathbf{N}$, and $\mathbf{D}$ depend on the generalized position vector $\vec{s}$ which itself is dependent on time.

In some simulators the time dependency is ignored and the simulator uses a fixed time-stepping routine such as the Open Dynamics Engine does [112]. The general idea is illustrated in Figure 4.15, and the main advantage is that only a single LCP problem is solved per time-step. Unfortunately, it also leads to penetrations and drifting problems. In the Open Dynamics Engine several heuristics are used to overcome these problems: Constraint Force Mixing, Error Reduction Parameter, and a specialized position update. These heuristics will be discussed later.

We will now discuss the detection of contacts. There are several ways to approach this for the the pseudo-code in Figure 4.15. One way is to invoke the collision detection at time t and hope for the best, but new contacts might develop during the time from t to $t + \Delta t$. The future contacts that are overlooked at time t could potentially end up as

```
Algorithm fixed-time-step(s^t, u^t, Δt)
  s' = s^t + Δt S u^t
  λ = LCP(s', u^t)
  u^{t+Δt} = u^t + M^{-1} ( J^T λ + Δt f_ext )
  s^{t+Δt} = s^t + Δt S u^{t+Δt}
  return s^{t+Δt}
End algorithm
```

Figure 4.15: Fixed time-stepping.

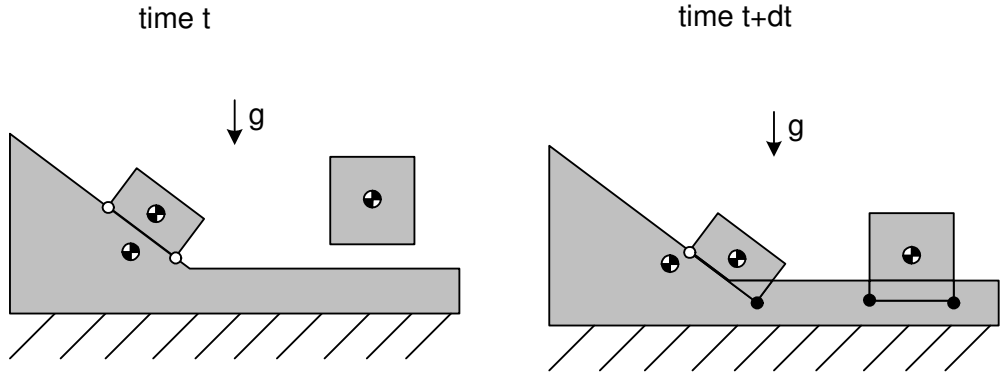


Figure 4.16: 2D illustration of the problem with violated contacts in a fixed time stepping due to overlooking potential furture contact. Small white circles show valid contacts, small black circles show violated contacts. Notice that contacts not detected at time t will be detected as violated contacts at time $t + dt$.

violated contacts at time $t + \Delta t$. That is, deeply penetrating contacts. This is illustrated in Figure 4.16.

Stewart and Trinkle [140] propose a retroactive detection approach to detecting overlooked future contacts. First, all contacts are detected at time t then a fixed time-step is taken, and finally all contacts at time $t+\Delta t$ are detected. If any new and violated contacts are found, then these contacts are added to the set of contacts found at time t, and the simulation is then rewound to time t, and a new fixed time-step is taken. These steps are repeated until all the necessary contacts have been found. Pseudo-code can be found in Figure 4.17. Although it is obvious that only a finite number of contacts exist, and that the algorithm sooner or later will have detected all contacts necessary for preventing penetration, it is however not obvious how many iterations the algorithm will take, and it is therefore difficult to say anything meaningful about the time-complexity. The retroactive detection of contacts is similar to the contact tracking algorithm described in [101].

Detecting a future contact at an early stage can have a undesirable effect. In essence, a contact that appears at time $t + \Delta t$ is resolved at time t. This would make objects appear to have a force field or envelope surrounding them preventing other objects from actually touching them. Most noticeably, this could cause object hanging in the air as shown in Figure 4.18. If we are simulating fast moving and bouncing objects, it is unlikely that an observer would ever notice the visual artifacts, but if the time-step is big enough the artifacts illustrated in Figure 4.18 may annoy an observer. To remedy this effect one

```
Algorithm retroactive-1(s^t, u^t, Δt)
  repeat
    S_contacts = collision detection(s^t)
    s^{t+Δt} = fixed-time-step(s^t, u^t, Δt)
    S'_contacts = collision detection(s^{t+Δt})
    if violated contacts in S'_contacts then
      for all contacts c ∈ S'_contacts do
        if c violated and c ∉ S_contacts then
          add c to S_contacts
      next c
    else
      return s^{t+Δt}
    end if
  until forever
End algorithm
```

Figure 4.17: Retroactive detection of contacts in a fixed time-stepping method.

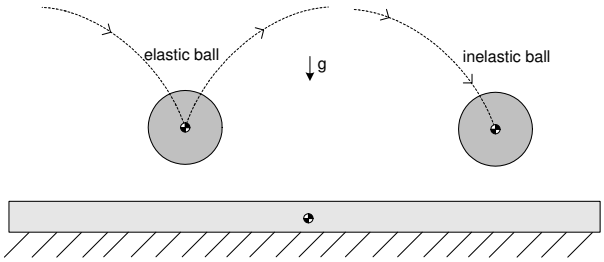


Figure 4.18: 2D example showing visual artifacts of resolving future contacts at time t. The elastic ball never touches the fixed box before it bounces, while the inelastic ball is hanging in the air after its impact.

could either lower the size of the time-step, which causes a performance degradation, or one could remodel the error terms to take into account that a contact does not exist at time t. This approach was elaborated on in Section 4.5.

Another issue with time-stepping methods is that penetrations might occur when the system moves along concave boundaries of the admissible region in configuration space [4, 138]. Configuration space is the vector space consisting of the concatenation of the generalized position and velocity vectors. The admissible region is the subspace of states that make physical sense, like states that do not cause a penetration. A simple example is illustrated in Figure 4.19. As it can be seen in Figure 4.19, the dotted line indicates the normal constraint, and the box is therefore allowed to move along this line. Furthermore, since the lower fixed object is concave at the touching contact point, then a penetration will always occur no matter how small a step the box moves along the dotted line. This indicates that situations will arise where all necessary contacts have been detected. But still, the algorithm will find violated contacts and enter an infinite loop since the *if*-statement in the pseudo-code of Figure 4.17 will always be true. This seems to be an unsolved problem in the literature.

Stewart and Trinkle [138] suggest that simple projection can be used to eliminate the

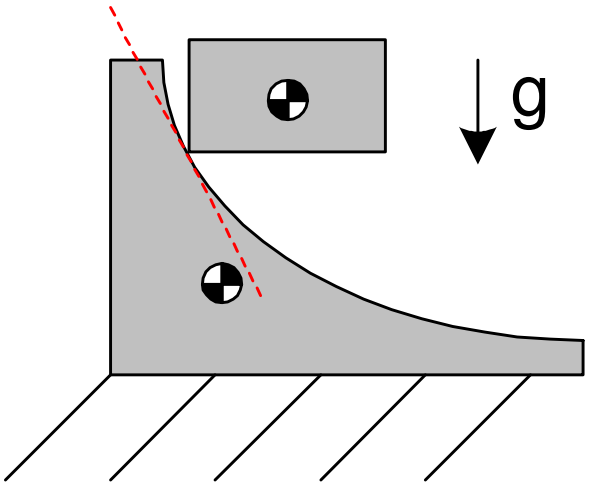


Figure 4.19: A system moving along a concave boundary of the admissible region in configuration space.

problem of penetrations as a result of moving along concave boundaries in configuration space. However they state that care must be taken to avoid losing energy in the process. In [17] a displacement is computed by using a first order system $\mathbf{A}\vec{f} = \vec{b}$, where $\mathbf{A}$ only contains normal and bilateral constraints. In our notation this corresponds to,

$$\mathbf{A} = \begin{bmatrix} \mathbf{N}^T\mathbf{C}^T\mathbf{M}^{-1}\mathbf{C}\mathbf{N} & \mathbf{N}^T\mathbf{C}^T\mathbf{M}^{-1}J_\Phi \\ J_\Phi^T\mathbf{M}^{-1}\mathbf{C}\mathbf{N} & J_\Phi^T\mathbf{M}^{-1}J_\Phi \end{bmatrix}. \tag{4.211}$$

The $\vec{b}$-vector contains the signed constraint errors. From the solution, $\vec{f}$, and the derivative, $\dot{\vec{s}}$, can be computed as described in Section 4.16. Now using $\Delta s = \dot{\vec{s}}$, a displacement can be obtained as $\vec{s}^{t+\Delta t} = \vec{s}^{t+\Delta t} + \Delta s$, see [17] for more details.

Solving penetration errors by projection can change the potential energy of the system, thus changing the total energy in the system.

In some cases energy may be added to the system, which is a problem if energy conservation is important. During subsequent simulation, the potential energy may be transformed into kinetic energy. Thus during simulation it will be observed that objects gain speed unexplainedly. In conclusion, these artifacts are either unwanted from a physical viewpoint or an animation viewpoint. Figure 4.20 illustrates the energy problem with the projection method. In the figure, three possible projections of the box are shown as dashed boxes. Due to the gravitational field $\vec{g}$, both the vertical and the inclined projections will increase the energy of the system, while only the horizontal projection will maintain the same energy level. If the geometries are mirrored in the vertical direction, then the vertical and inclined projections will result in a loss of energy. In other words, projection must be done in such a way that the potential energy remains unchanged. It is not obvious to us how a general method of projection could be designed to fulfill this.

Another approach for handling penetration errors is to use constraint stabilization. Intuitively, this can be explained as inserting small damped virtual springs in between objects at those places where penetration errors are detected. In Section 4.8 and 4.9 the constraint based method was extended with error correcting terms that stabilized the constraints. An error reduction parameter was used to control the amount of error

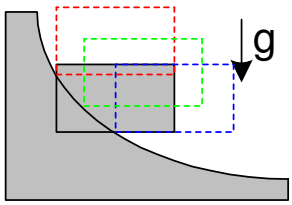


Figure 4.20: Different ways to project a violated contact back into a valid contact. The horizontal projection does not change the potential energy of the box, while both the vertical and the inclined projections increase the potential energy. In the following time-step, the added potential energy is transformed into kinetic energy, resulting in a more violent collision between the two objects.

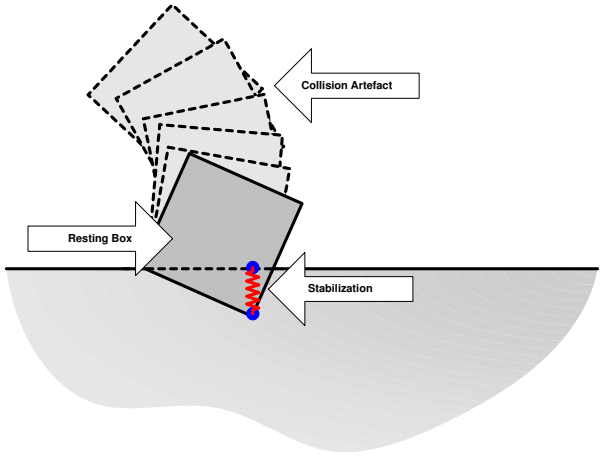


Figure 4.21: An example of constraint stabilization on a violated contact of a resting box causing the box to bounce off the resting surface.

correction. In case of large errors or over-eager error reduction, stabilization can cause severe alterations of the simulated motion because the stabilization accelerates objects apart in order to fix the error. Thus, the stabilization adds kinetic energy to the system. In severe cases, the stabilization can cause a shock-like effect on constraints with large errors. The artifact of stabilization is illustrated in Figure 4.21. Several approaches exist for minimizing the problem, but none of them completely remove it. By lowering the time-step size, the magnitude of constraint errors would also be lowered, thus decreasing the chance of large constraint errors, all with a performance hit. Lowering the error reduction parameter minimizes the chance of stabilization being too eager. The drawback is that error correction will take longer, thus increasing the chance of an observer noticing deep penetration of objects. Finally, constraints can be made softer by using Constraint Force Mixing as described in Section 4.15. This have the same drawbacks as lowering the error reduction parameter. Furthermore, it removes kinetic energy from the system. Thus, if used too aggressively, objects appear more lazy than they should be.

The matrices $\mathbf{M}$, $\mathbf{C}$, $\mathbf{N}$, and $\mathbf{D}$ depend on time and the constraint based method should be formulated as a nonlinear complementarity problem (NCP). An NCP can be solved iteratively by using a fix-point algorithm as e.g. shown in Figure 4.22.

In [4], an explicit time-stepping method is used together with retroactive detection

```
Algorithm fixpoint(s⃗^t, u⃗^t, Δt)
  s⃗' = s⃗^t + ΔtS u⃗^t
  repeat
    λ⃗ = LCP(s⃗', u⃗^t)
    u⃗' = u⃗^t + M^{-1} ( J^T λ⃗ + Δt f⃗_ext )
    s⃗'' = s⃗'
    s⃗' = s⃗^t + ΔtS u⃗'
  until |s⃗' − s⃗''| < ε_fix
  s⃗^{t+Δt} = s⃗'
  return s⃗^{t+Δt}
End algorithm
```

Figure 4.22: Fix-point-iteration algorithm, Typical values according to [129] are $\Delta t \leq 10^{-3}$ and $\epsilon_{fix} \leq 10^{-4}$.

```
Algorithm retroactive-2(s⃗^t, u⃗^t, Δt)
  s⃗' = s⃗^t
  u⃗' = u⃗^t
  h = Δt/n_steps
  t = 0
  while t < Δt do
    s⃗'' = s⃗' + hS u⃗'
    λ⃗ = LCP(s⃗'', u⃗')
    if no collision then
      t = t + h
      s⃗' = s⃗''
      u⃗' = u⃗^t + M^{-1} ( (J^T λ⃗ + Δt f⃗_ext )
    else
      let t_toi be time of impact
      apply collision model
      ...
      t = t_toi
    end if
  end while
  s⃗^{t+Δt} = s⃗'
  return s⃗^{t+Δt}
End algorithm
```

Figure 4.23: Explicit time-stepping with retroactive detection of colliding contacts.

of collisions. We have outlined the general control flow in Figure 4.23. The main idea is to halt the simulation at the time of an impact, and then handle the impact before proceeding with simulation. In [4] two LCP problems are set up and used to solve for post-velocities of the impact. This is done using Poisson's Hypotheses for handling the compression and decompression phases of an impact. In [11], a simpler approach is used based on Newton's Impact Law. This leads to nearly the same kind of LCP problem we have outlined. Later, in [5], an even simpler collision model is used, which only supports completely inelastic collisions. The methods in [4, 11, 5] are termed simultaneous collision

methods. We will not treat these further. However, another possibility is to use a so-called sequential collision method.

In [5], the same control flow is used in an implicit-time-stepping method. The major difference there being the way the LCP is formed. We have chosen to omit the implicit version, since it is out of scope for the kind of applications we have in mind.

4.12.1 Numerical Issues with Retroactive Time Control

Due to numerical truncation and roundoff errors, thresholding is a necessary evil in order to determine the type of contact between two objects A and B. Using a threshold value ε, an often used rule of thumb for determining the contact state is as follows:

- if A and B are separated by ε, there is no contact.
- if the distance between A and B is less than ε, there is contact.
- if penetration between A and B is no more than ε, there is contact.
- if penetration between A and B is more than ε, A and B are intersecting.

Now let us review a typical retroactive advancement of the simulation. That is, if time control is handled in a root searching manner, the general idea is to watch out for penetrations, and then back track the simulation to the time of impact. This is illustrated in Figure 4.24. Of course, more elaborate and intelligent root search schemes could be used,

```
dt = T_target - T
do
  simulate-forward(dt)
  if intersection
    rewind-simulation()
    dt = dt/2
  else if contact
    collision-resolving()
    T = T + dt
    dt = T_target - T
  end if
while T<T_target
```

Figure 4.24: Example of retroactive advancement based on a bisection root search algorithm.

but the bisection scheme in Figure 4.24 suffice for our discussion in this section.

Suppose A and B are in resting contact and penetrate each other with a penetration of ε, and imagine that during the forward simulation a small numerical drift causes A and B to inter-penetrate with a depth greater than ε. As can be seen from the pseudo-code, the root searching algorithm will be fooled to think that a collision has occurred, even though no collision occurred. Consequently, the root searching algorithm will begin to backtrack in order to search for the time of impact. The result will be a never ending search for a non-existent root. Even if a root is determined within a threshold value, the

subsequent attempt to advance the simulation is doomed to repeat a new root search for the same non-existent root. In other words, the simulation will either go into an infinite loop, never advancing the simulation, or, alternatively, the system will end up in a faulty state with penetrations greater than ε.

One possible strategy to avoid the above mentioned problems would be to adapt a more advanced thresholding method, using one threshold for the root searching algorithm, and another threshold value for determining the type of contact [17]. However, it is extremely difficult to get such an approach to work efficiently and robustly in practice. Another way out would be to penalize penetrations by a force-feedback term. However, what you get is all the trouble of penalty methods, and there is no guarantee that the "error" will be within a certain limit, since a penalty force, like a spring, does not impose a hard constraint.

If your simulator is already based on a penalty method, you have nothing to worry about. However, retroactive advancement is really a bad choice for a penalty based simulator, and instead, fixed time-stepping is often the preferred choice. On the other hand, if your simulator is a constraint based simulator, it seems daunting to add penalty forces to it, since it essentially will turn your simulator into a hybrid simulator. Not only do you get the advantages from both types of simulator, but you could also inherit all the problems.

In our opinion, a far better approach for resolving the problem is to displace objects when their penetration depth is greater than a certain tolerance correction value. E.g. set the tolerance correction to,

$$\delta = \frac{3}{4}\varepsilon, \tag{4.212}$$

whenever the penetration depth increases beyond δ. Then objects are displaced to reduce or remove penetrations. This resolution technique will guarantee that you never will get into problems with your main simulation loop. However, there are some drawbacks. A displacement could change the potential energy of an object. If the energy is increased, the subsequent simulation might transform the potential energy into kinetic energy, causing objects to begin jittering and jumping around. In our opinion, it is more pleasing to look at a simulation where potential energy vanishes. Our main point is that displacements should occur such that the potential energy is non-increasing.

4.12.2 Unavoidable Penetrations

In some cases, penetrations are simply unavoidable due to the discrete time-stepping. This can be hard to grasp, so we will present a small example originally presented in [93], illustrating how penetrations can occur. The example uses the fixed semi-implicit time-stepping method given in Figure 4.15. In the initial state, the rod is moving to the right and the box is fixed as seen in Figure 4.25. The time-stepping method first tries to guess the next position by doing a fake position update, $t' = t + \Delta t$. This fake position is first used to determine contact points shown as small solid circles, and then the velocities are updated at the fake position. As shown in the figure, the velocity update predicts a tipping movement of the rod at the fake position. Now the time-stepping method will use the velocities computed at the fake position to update the real position. However, the tipping movement will now cause an unwanted penetration of the rod and the box.

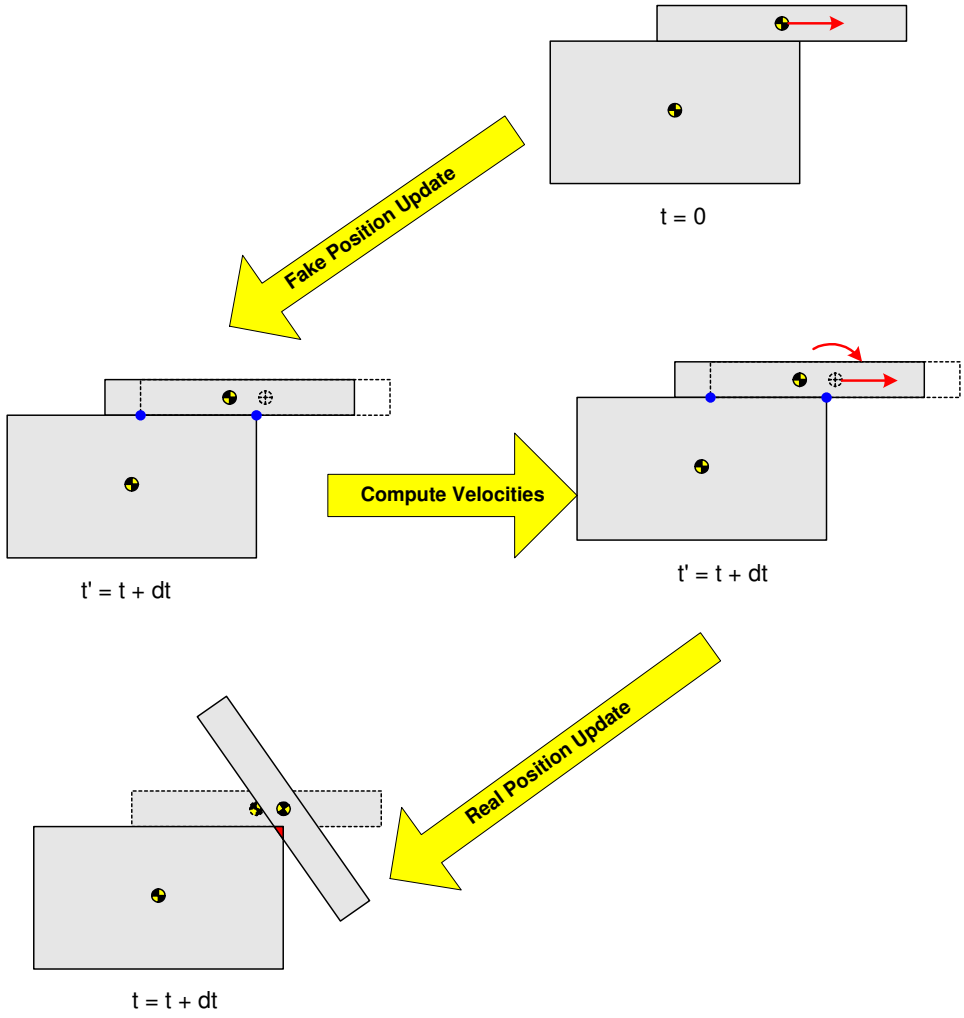


Figure 4.25: Illustration of how penetrations can occur with fixed semi-implicit time-stepping.

A rule of thumb is that penetrations can occur of the same order as the order of the time stepping method used. To be specific the fixed semi-implicit time stepping method introduced in Section 4.12 is of first order, i.e. $O(h)$, where h denotes the step-size. The errors accumulate linearly with the number of steps n, such that the error after n steps is proportional to nh. This might appear to be bad. However, from a convergence theory point of view, lowering h will guarantee better accuracy, and in the limit $h \mapsto 0$ a perfect solution will be found. In practice, this is not feasible, since the limiting case can not be reached on a computer with finite precision arithmetic, and even if the limiting cases were possible, it would take far to long to compute. Therefore, a better practical approach is to pick a step size such that errors never grow large enough for an end-user to notice them.

Another example of unavoidable penetration is the configuration shown in Figure 4.19. Here, the problem is that we have a linear discretization but a higher order surface. The "numerics" can not "see" the higher order and do not care about it.

In cloth simulation, there exists a fixed-time-stepping scheme [28] which guarantees

non-penetration. It is, however not obvious to us how this can migrated to rigid body dynamics. Besides, it relies on continuous collision detection to detect proximities, making it a less attractive choice for interactive and real-time applications.

4.13 A unified Object Oriented Constraint Design

From previous sections we have learned that four types of constraints could be expressed by use of Jacobians. These were joint constraints, contacts, joint limits, and joint motors. For these types of constraints we derived the following kinematic constraints,

$$J_{\text{joint}}\vec{u} = \vec{b}_{\text{joint}}, \tag{4.213a}$$
$$J_{\text{contact}}\vec{u} \geq \vec{b}_{\text{contact}}, \tag{4.213b}$$
$$J_{\text{limit}}\vec{u} \geq \vec{b}_{\text{limit}}, \tag{4.213c}$$
$$J_{\text{motor}}\vec{u} \geq \vec{b}_{\text{motor}}. \tag{4.213d}$$

Adding the "reaction" forces to the equations of motion result in the generalized acceleration vector,

$$\dot{\vec{u}} = \mathbf{M}^{-1}\left(J_{\text{joint}}^T\vec{\lambda}_{\text{joint}} + J_{\text{contact}}^T\vec{\lambda}_{\text{contact}} + J_{\text{limit}}^T\vec{\lambda}_{\text{limit}} + J_{\text{motor}}^T\vec{\lambda}_{\text{motor}} + \vec{f}_{\text{ext}}\right), \tag{4.214}$$

with the following limits on the Lagrange multipliers,

$$-\infty \leq \vec{\lambda}_{\text{joint}} \leq \infty, \tag{4.215a}$$
$$0 \leq \vec{\lambda}_{\text{contact}} \leq \infty, \tag{4.215b}$$
$$0 \leq \vec{\lambda}_{\text{limit}} \leq \infty, \tag{4.215c}$$
$$-\vec{\lambda}_{\max} \leq \vec{\lambda}_{\text{motor}} \leq \vec{\lambda}_{\max}. \tag{4.215d}$$

Performing the usual discretization steps and substitutions we derive the following complementarity formulation,

$$\begin{bmatrix} \begin{bmatrix} J_{\text{joint}}\mathbf{M}^{-1}J_{\text{joint}}^T & J_{\text{joint}}\mathbf{M}^{-1}J_{\text{contact}}^T & J_{\text{joint}}\mathbf{M}^{-1}J_{\text{limit}}^T & J_{\text{joint}}\mathbf{M}^{-1}J_{\text{motor}}^T \\ J_{\text{contact}}\mathbf{M}^{-1}J_{\text{joint}}^T & J_{\text{contact}}\mathbf{M}^{-1}J_{\text{contact}}^T & J_{\text{contact}}\mathbf{M}^{-1}J_{\text{limit}}^T & J_{\text{contact}}\mathbf{M}^{-1}J_{\text{motor}}^T \\ J_{\text{limit}}\mathbf{M}^{-1}J_{\text{joint}}^T & J_{\text{limit}}\mathbf{M}^{-1}J_{\text{contact}}^T & J_{\text{limit}}\mathbf{M}^{-1}J_{\text{limit}}^T & J_{\text{limit}}\mathbf{M}^{-1}J_{\text{motor}}^T \\ J_{\text{motor}}\mathbf{M}^{-1}J_{\text{joint}}^T & J_{\text{motor}}\mathbf{M}^{-1}J_{\text{contact}}^T & J_{\text{motor}}\mathbf{M}^{-1}J_{\text{limit}}^T & J_{\text{motor}}\mathbf{M}^{-1}J_{\text{motor}}^T \end{bmatrix} & \mathbf{A}_{\text{aux}} \\ \mathbf{A}_{\text{aux}} & \mathbf{A}_{\text{aux}} \end{bmatrix} \begin{bmatrix} \vec{\lambda}_{\text{joint}} \\ \vec{\lambda}_{\text{contact}} \\ \vec{\lambda}_{\text{limit}} \\ \vec{\lambda}_{\text{motor}} \\ \vec{\lambda}_{\text{aux}} \end{bmatrix} + \begin{bmatrix} J_{\text{joint}}\left(\vec{u} + \Delta t\mathbf{M}^{-1}\vec{f}_{\text{ext}}\right) - \vec{b}_{\text{joint}} \\ J_{\text{contact}}\left(\vec{u} + \Delta t\mathbf{M}^{-1}\vec{f}_{\text{ext}}\right) - \vec{b}_{\text{contact}} \\ J_{\text{limit}}\left(\vec{u} + \Delta t\mathbf{M}^{-1}\vec{f}_{\text{ext}}\right) - \vec{b}_{\text{limit}} \\ J_{\text{motor}}\left(\vec{u} + \Delta t\mathbf{M}^{-1}\vec{f}_{\text{ext}}\right) - \vec{b}_{\text{motor}} \\ \vec{b}_{\text{aux}} \end{bmatrix} \geq \vec{0}, \tag{4.216}$$

which is complementary to,

$$\begin{bmatrix} -\infty \\ 0 \\ 0 \\ -\vec{\lambda}_{\max} \\ 0 \end{bmatrix} \leq \begin{bmatrix} \vec{\lambda}_{\text{joint}} \\ \vec{\lambda}_{\text{contact}} \\ \vec{\lambda}_{\text{limit}} \\ \vec{\lambda}_{\text{motor}} \\ \vec{\lambda}_{\text{aux}} \end{bmatrix} \leq \begin{bmatrix} \infty \\ \infty \\ \infty \\ \vec{\lambda}_{\max} \\ \infty \end{bmatrix}. \tag{4.217}$$

Here $\mathbf{A}_{\text{aux}}$ and $\vec{b}_{\text{aux}}$ correspond to a permutation of the third row and column in (4.61), which are the auxiliary constraints needed to model the frictional force.

We may concatenate all the Jacobians into a single matrix,

$$J = \begin{bmatrix} J_{\text{joint}} \\ J_{\text{contact}} \\ J_{\text{limit}} \\ J_{\text{motor}} \end{bmatrix}. \tag{4.218}$$

Similarly, we may concatenate all the error correcting terms,

$$\vec{b}_{\text{error}} = \begin{bmatrix} \vec{b}_{\text{joint}} \\ \vec{b}_{\text{contact}} \\ \vec{b}_{\text{limit}} \\ \vec{b}_{\text{motor}} \end{bmatrix}, \tag{4.219}$$

and the Lagrange multipliers,

$$\vec{\lambda} = \begin{bmatrix} \vec{\lambda}_{\text{joint}} \\ \vec{\lambda}_{\text{contact}} \\ \vec{\lambda}_{\text{limit}} \\ \vec{\lambda}_{\text{motor}} \\ \vec{\lambda}_{\text{aux}} \end{bmatrix}, \tag{4.220}$$

and we may thus write,

$$\underbrace{\begin{bmatrix} J\mathbf{M}^{-1}J^T & \mathbf{A}_{\text{aux}} \\ \mathbf{A}_{\text{aux}} & \mathbf{A}_{\text{aux}} \end{bmatrix}}_{\mathbf{A}} \vec{\lambda} + \underbrace{\begin{bmatrix} J\left(\vec{u} + \Delta t \mathbf{M}^{-1}\vec{f}_{\text{ext}}\right) - \vec{b}_{\text{error}} \\ \vec{b}_{\text{aux}} \end{bmatrix}}_{\vec{b}} \geq \vec{0} \tag{4.221}$$

complementary to

$$\underbrace{\begin{bmatrix} -\infty \\ 0 \\ 0 \\ -\vec{\lambda}_{\max} \\ 0 \end{bmatrix}}_{\vec{\lambda}_{\text{low}}} \leq \vec{\lambda} \leq \underbrace{\begin{bmatrix} \infty \\ \infty \\ \infty \\ \vec{\lambda}_{\max} \\ \infty \end{bmatrix}}_{\vec{\lambda}_{\text{high}}}. \tag{4.222}$$

Rewriting the complementarity formulation into this form will allow us to compute the system matrix very efficiently. Notice also how easy we can write the generalized acceleration vector with the new notation,

$$\dot{\vec{u}} = \mathbf{M}^{-1}\left(J^T\vec{\lambda} + \vec{f}_{\text{ext}}\right). \tag{4.223}$$

When assembling the system matrix, we must first allocate space for all matrices involved, and then setup the sub-block structure for each individual constraint. For each constraint we evaluate it, i.e. compute joint axes, joint positions, joint errors and so on. That is all the information needed in the Jacobian matrix and the corresponding error term for each constraint, including any auxiliary constraints.

In the evaluation, care must be taken since joint limits and motors depend on the joints they are attached to, and joint motors and limits must therefore not be evaluated before the joints they are attached to are.

Having performed the evaluation, we can determine which constraints are currently active and which are non-active. The non-active constraints can simply be dropped from further consideration in the current assembly. As an example, the joint limits is non-active whenever the joint has not reached its "outer" limit.

Knowing how many active constraints there are, we can compute the dimensions of the matrices involved. For each constraint we will query how many rows its Jacobian matrix contains, and we will query how many auxiliary variables there are. Summing these up, we are able to determine the total dimensions of the matrices and vectors needed in the assembly.

During these summations, it is also possible to assign indices for the sub-blocks for each and every constraint, i.e. where its Jacobian matrix, error term and the auxiliary variable data should be mapped to. Figure 4.26 shows the pseudo-code. Observe that substantial memory savings can be achieved by using a simple sparse matrix representation of the matrices $\mathbf{M}^{-1}$ and J, as shown in the pseudo-code as well. We can now start filling in data in the matrices $\mathbf{M}^{-1}$ and J, and the vector $\mathbf{b}_{\text{error}}$, together with the parts of the system matrix $\mathbf{A}$ and the right hand side $\vec{b}$ containing the auxiliary data. Also, external forces, limits and the generalized velocity vector are easily dealt with as shown in Figure 4.27. Using these matrices, it is quite easy to compute (4.221) in a straightforward manner. The only real difficulty is that one must be careful about the sparse representation of the matrices $\mathbf{M}^{-1}$ and J^{-1}.

We have completed what we set out to do in this section. A unified framework for handling both contact point constraints with friction and joint constraints with both motors and limit has been derived.

The framework allows a modular and object oriented design of all the constraints in a simulator. This object oriented design is outlined in Figure 4.28. In the figure we have omitted details regarding assembly of the jointed mechanism (linking bodies together with joints, and adding limits and motors).

4.14 Modified Position Update

When the new generalized position vector is computed as,

$$\vec{s}^{\,t+\Delta t} = \vec{s}^{\,t} + \Delta t \mathbf{S} \vec{u}^{\,t+\Delta t}, \tag{4.224}$$

then we call it a position update. In the position update written above, an "infinitesimal" orientation update is used. This is fast to compute, but can occasionally cause inaccuracies for bodies that are rotating at high speed, and especially when they are joined to other bodies.

```
Algorithm allocate(...)
  C = set of all constraints
  n_jacobian = 0
  for each constaint c ∈ C do
    c.evaluate()
    if not c.active() then
      remove c from C
    end if
    c.setJacobianIndex(n_jacobian)
    n_jacobian+ = c.getNumberOfJacobianRows()
  next c

  n_auxiliary = 0
  for each constaint c ∈ C do
    c.setAuxiliaryIndex(n_jacobian + n_auxiliary)
    n_auxiliary+ = c.getNumberOfAuxiliaryVars()
  next c

  J = matrix(n_jacobian, 12)
  b_error = vector(n_jacobian)
  n_total = n_jacobian + n_auxiliary
  A = matrix(n_total, n_total)
  b = vector(n_total)
  λ = vector(n_total)
  λ_low = vector(n_total)
  λ_high = vector(n_total)

  B = set of all bodies
  n_body = B.size()
  M_inv = matrix(3, 6n_body)
  u = vector(6n_body)
  f_ext = vector(6n_body)
End algorithm
```

Figure 4.26: Allocate system matrix and setup sub-block structure for constraints.

For instance, in a car simulation, four wheels might be attached to a chassis with a wheel joint. When driving the car, the wheels may rotate in incorrect directions, as though the joints were somehow becoming ineffective. The problem is observed, when the car is moving fast and turning. The wheels appear to rotate off their proper constraints as though the wheel axes have been bent. If the wheels are rotating slowly, or the turn is made slowly, the problem is less apparent. The problem is that the high rotation speed of the wheels is causing numerical errors. A "finite" orientation update can be used to reduce such inaccuracies. This is more costly to compute, but will be more accurate for high speed rotations. High speed rotations can result in many types of error in a simulation, and a "finite" orientation update will only fix one of those sources of error.

We will now outline a modified position update like the one used in the Open Dynamics Engine [112]. For each body B_i we want to be able to set a kind of bit flag indicating whether the body should be updated using the infinitesimal or finite update method, and

```
Algorithm fillIn(...)
  C = set of all active constraints
  for each constaint c ∈ C do
    i = c.getJacobianIndex()
    j = i + c.getNumberOfJacobianRows()
    J^i_lin = J_(i..j),(0..2)
    J^j_lin = J_(i..j),(3..5)
    J^i_ang = J_(i..j),(6..8)
    J^j_ang = J_(i..j),(9..11)
    c.getLinearJacobian_i(J^i_lin)
    c.getLinearJacobian_j(J^j_lin)
    c.getAngularJacobian_i(J^i_ang)
    c.getAngularJacobian_j(J^j_ang)
    c.getErrorTerm(b⃗_error,(i..j))
    c.getJacobianLowLimit(λ⃗_low,(i..j))
    c.getJacobianHighLimit(λ⃗_high,(i..j))
  next c
  for each constaint c ∈ C do
    i = c.getAuxiliaryIndex()
    j = i + c.getNumberOfAuxiliaryVars()
    A_rows = A_(i..j),(0..n_total−1)
    A_cols = A_(0..n_total−1),(i..j)
    c.getAuxiliaryRowsAndColumns(A_rows, A_cols)
    c.getAuxiliaryLowLimit(λ⃗_low,(i..j))
    c.getAuxiliaryHighLimit(λ⃗_high,(i..j))
    c.getAuxiliaryRightHandSide(b⃗_(i..j))
  next c
  B = set of all bodies
  for each body b ∈ B do
    i = 6 b.getIndex()
    j = i + 6
    b.getMassInvMatrix(M_inv,(i..i+3,0..2))
    b.getInertiaInvMatrix(M_inv,(i+3..i+5,3..5))
    b.getLinearVelocity(u⃗ i..i+2)
    b.getAngularVelocity(u⃗ i+3..i+5)
    b.getExternalForce(f⃗_ext,(i..i+2))
    b.getExternalTorque(f⃗_ext,(i+3..i+5))
    b.getInertiaMatrix(I)
    b.getAngularVelocity(ω⃗)
    f_ext,(i+3..i+5) −= ω⃗ × Iω⃗
  next b
End algorithm
```

Figure 4.27: Fill-in data in sub-block structure for constraints and bodies.

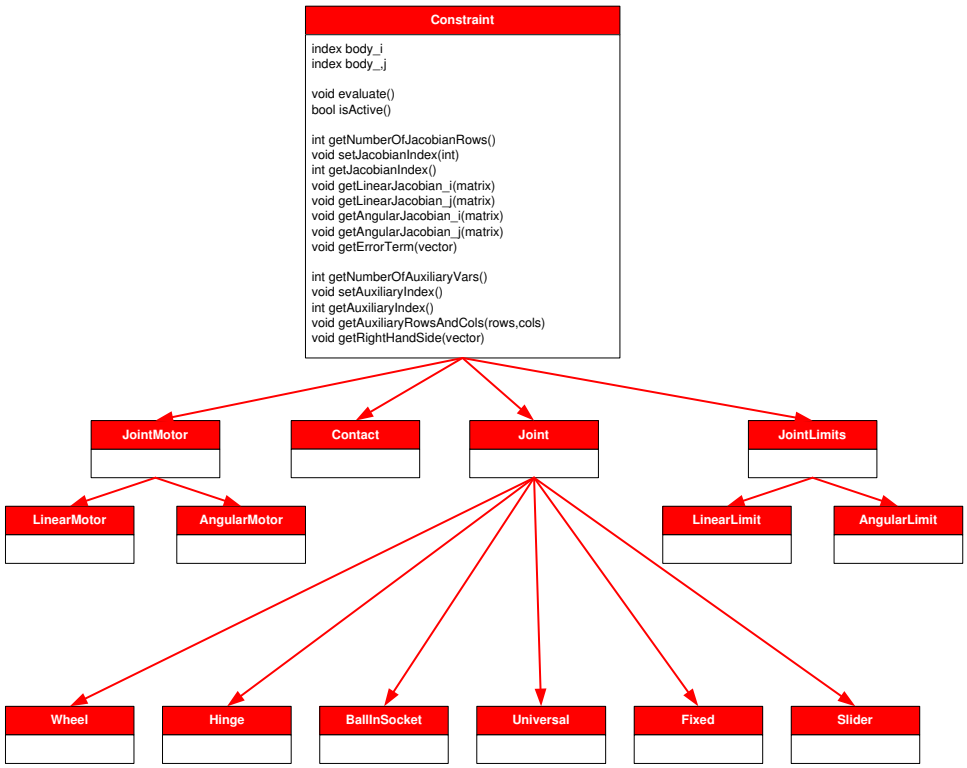


Figure 4.28: The constraints design. Observe the unification of contacts and joints.

that should be done at configuration design time. In case of the finite update we want to compute a rotation, q, corresponding to the radians traveled around the rotation axis in the time interval Δt. That is, the rotation angle θ is given by,

$$\theta = \Delta t ||\vec{\omega}_i||, \tag{4.225}$$

where the rotation axis is,

$$\vec{u} = \frac{\vec{\omega}_i}{||\vec{\omega}_i||}. \tag{4.226}$$

The corresponding quaternion q is then given as,

$$q = \left[\cos\left(\frac{\theta}{2}\right), \vec{u} \sin\left(\frac{\theta}{2}\right) \right]. \tag{4.227}$$

This can be rewritten as follows,

$$q = \left[\cos\left(\frac{\theta}{2}\right), \vec{u}\sin\left(\frac{\theta}{2}\right)\right] \tag{4.228a}$$

$$= \left[\cos\left(\frac{\theta}{2}\right), \frac{\vec{\omega}_i}{||\vec{\omega}_i||}\sin\left(\frac{\Delta t||\vec{\omega}_i||}{2}\right)\right] \tag{4.228b}$$

$$= \left[\cos\left(\frac{\theta}{2}\right), \vec{\omega}_i\frac{\sin\left(\frac{\Delta t||\vec{\omega}_i||}{2}\right)}{||\vec{\omega}_i||}\right] \tag{4.228c}$$

$$= \left[\cos\left(\frac{\theta}{2}\right), \frac{\Delta t}{2}\vec{\omega}_i\frac{\sin\left(\frac{\Delta t||\vec{\omega}_i||}{2}\right)}{\frac{\Delta t||\vec{\omega}_i||}{2}}\right] \tag{4.228d}$$

$$= \left[\cos\left(\frac{\Delta t||\vec{\omega}_i||}{2}\right), \frac{\Delta t}{2}\vec{\omega}_i\text{sinc}\left(\frac{\Delta t||\vec{\omega}_i||}{2}\right)\right]. \tag{4.228e}$$

Introducing the notation $h = \Delta t/2$, and $\theta = ||\vec{\omega}_i||h$, we end up with,

$$q = \left[\cos(\theta), \vec{\omega}_i\text{sinc}(\theta)\, h\right], \tag{4.229}$$

where

$$\text{sinc}(x) = \begin{cases} 1 - \frac{x^2}{6} & \text{if } |x| < \varepsilon, \\ \frac{sin(x)}{x} & \text{otherwise.} \end{cases} \tag{4.230}$$

In order to avoid division by zero, we have patched the sinc-function around 0 by using a Taylor expansion for small values $\varepsilon = 10^{-4}$. Furthermore, we want to be able to do both a full finite orientation update and a partial finite orientation update. We have already taken care of the full finite orientation update, and for the partial finite orientation update we split the angular velocity vector into a component $\vec{\omega}_{\text{finite}}$ along a specified finite rotation axis $\vec{r}_{\text{axis}}$, and a component $\vec{\omega}_{\text{infinitesimal}}$ orthogonal to it, i.e.,

$$\vec{\omega}_{\text{finite}} = (\vec{r}_{\text{axis}} \cdot \vec{\omega}_i)\,\vec{r}_{\text{axis}}, \tag{4.231a}$$

$$\vec{\omega}_{\text{infinitesimal}} = \vec{\omega}_i - \vec{\omega}_{\text{finite}}. \tag{4.231b}$$

First, a finite update is done with $\vec{\omega}_{\text{finite}}$ followed by an infinitesimal update done with $\vec{\omega}_{\text{infinitesimal}}$,

$$q_i = qq_i, \tag{4.232a}$$

$$q_i = q_i + \Delta t Q_i \vec{\omega}_{\text{infinitesimal}}. \tag{4.232b}$$

A partial finite orientation update can be useful in a situation like the previously mentioned wheel problem. Simply set the finite rotation axis equal to the hinge axis. Figure 4.29 shows the pseudo-code for the position update on body B_i.

4.15 Constraint Force Mixing

The constraint equation for a joint has the form,

$$J\vec{u} = \vec{b}, \tag{4.233}$$

```
Algorithm position-update (i, Δt)
  r_i = r_i + Δt v_i
  if finite rotation on i then
    if finite rotation axis on i then
      ω_finite = (r_axis · ω_i) r_axis
      ω_infinite = ω_i − ω_finite
      Δt = Δt/2
      θ = (r_axis · ω_i) Δt
      q_s = cos(θ)
      q_v = (sinc(θ)Δt) ω_finite
    else
      Δt = Δt/2
      θ = ||ω_i|| * Δt
      q_s = cos(θ)
      q_v = (sinc(θ)Δt) ω_i
    end if
    q = [q_s, q_v]
    q_i = q q_i
    if finite rotation axis on i then
      q_i = q_i + Δt Q_i ω_infinite
    end if
  else
    q_i = q_i + Δt Q_i ω_i
  end if
  normlize(q_i)
End Algorithm
```

Figure 4.29: Position update on i'th body.

where $\vec{u}$ is a velocity vector for the bodies involved, J is the Jacobian matrix with one row for every degree of freedom the joint removes from the system, and $\vec{b}$ is the error correcting term.

The constraint forces, i.e. the reaction forces, from the joint bearings are computed by,

$$\vec{F} = J^T \vec{\lambda}, \tag{4.234}$$

where $\vec{\lambda}$ is a vector of Lagrange multipliers and has the same dimension as $\vec{b}$. The Open Dynamics Engine [112] adds a new twist to these equations by reformulating the constraint equation as,

$$J\vec{u} = \vec{b} - \mathbf{K}_{\text{cmf}}\vec{\lambda}, \tag{4.235}$$

Where $\mathbf{K}_{\text{cmf}}$ is a square diagonal matrix. The matrix $\mathbf{K}_{\text{cmf}}$ mixes the resulting constraint force in with the constraint. A nonzero (positive) value of a diagonal entry of $\mathbf{K}_{\text{cmf}}$ allows the original constraint equation to be violated by an amount proportional to $\mathbf{K}_{\text{cmf}}$ times the restoring force $\vec{\lambda}$. The equations of motion give,

$$M\ddot{\vec{u}} = M\frac{\vec{u}^{\,\Delta t+t} - \vec{u}}{\Delta t} = J^T\vec{\lambda}, \tag{4.236}$$

from which we isolate $\vec{u}^{\,\Delta t+t}$ as,

$$\vec{u}^{\,\Delta t+t} = \vec{u} + \Delta t M^{-1} J^T \vec{\lambda}. \tag{4.237}$$

Assuming that the constraint equation holds at time $t+\Delta t$ and substituting the expression for $\vec{u}^{\,\Delta t+t}$ into the constraint equation we get,

$$J\vec{u}^{\,\Delta t+t} = \vec{b} - \mathbf{K}_{\text{cmf}}\vec{\lambda}, \tag{4.238a}$$

$$J\vec{u} + \Delta t J M^{-1} J^T \vec{\lambda} = \vec{b} - \mathbf{K}_{\text{cmf}}\vec{\lambda}. \tag{4.238b}$$

Collecting $\vec{\lambda}$ terms on the left side we get

$$\left(J\mathbf{M}^{-1}J^T + \frac{1}{\Delta t}\mathbf{K}_{\text{cmf}} \right) \vec{\lambda} = \frac{1}{\Delta t}\left(\vec{b} - J\vec{u} \right), \tag{4.239}$$

from which it is clear that the $\mathbf{K}_{\text{cmf}}$ matrix is added to the diagonal of the original system matrix.

According to [112], using only positive values in the diagonal of $\mathbf{K}_{\text{cmf}}$ has the benefit of moving the system away from singularities and improve factorization accuracy.

This twist of the constraint equations is known as constraint force mixing, and the diagonal entries of the matrix $\mathbf{K}_{\text{cmf}}$ controls the amount of mixing that is done.

4.16 First Order World

First order world simulation is greatly inspired by the way Aristotle saw and described the world in his time. These misconceptions were later rectified by Newton in his 3 laws of motion. Regardless, a first order world is useful in animation for error correction and precise positioning of objects. Aristotle's basic views were,

1. Heavier objects fall faster.

2. To keep an object in motion at constant velocity, a constant force is needed.

The problem with this second statement was in not realizing that, in addition to the pulling or pushing force, there are other forces involved, typically friction and air or water resistance. Fortunately, these misconceptions are of great practical use in computer animation and also in more serious applications such as virtual prototyping e.g. [124].

A first order world is useful due to its controllability. In Newtonian mechanics, objects keep moving or sliding after a user interaction, making it hard to accurately position objects interactively in a virtual world. Contrary, in a first order world an object would stop immediately after a user stops pushing it.

Ease of interaction can be obtained by merely translating objects, but such an approach will not take care of the rotational movement induced by collision et cetera. In a first order world, misalignments between objects are taken care of through simulation.

For instance, if a light misaligned object is pushed against a heavy perfectly aligned object, the simulation will automatically align the two objects wrt. each other. Furthermore, the light object will be aligned more than the heavier object. Figure 4.30 and 4.31 show a few frames of a first order world simulation where two objects are automatically aligned. As seen in Figure 4.30 and 4.31 a first order world simulation is well suited for aligning objects and it respects their mass and inertia. This smooth alignment is very

Figure 4.30: A sequence in a first order world simulation where two identical objects are aligned. The left box is pulled to the right. Observe that both the left and right boxes are equally affected.

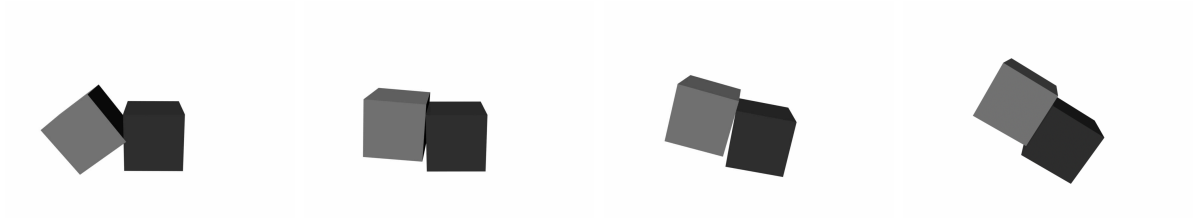


Figure 4.31: A sequence in a first order world simulation where two objects of different mass are aligned. The left box has less mass than the right box. The left box is pulled to the right. Observe that the heavier box on the right is less affected than the light box on the left in comparison with Figure 4.30.

attractive in many virtual environments, and this also makes a first order world simulation an ideal tool for correcting simulation errors [17]. An example of this is shown in Figure 4.32.

In a second order system we have a state function, $\vec{s}(t)$, for a rigid body,

$$\vec{s}(t) = \begin{bmatrix} \vec{r} \\ q \\ \vec{P} \\ \vec{L} \end{bmatrix}, \quad \text{and} \quad \dot{\vec{s}}(t) = \begin{bmatrix} \vec{v} \\ \frac{1}{2}\vec{\omega}q \\ \vec{F} \\ \vec{\tau} \end{bmatrix}, \tag{4.240}$$

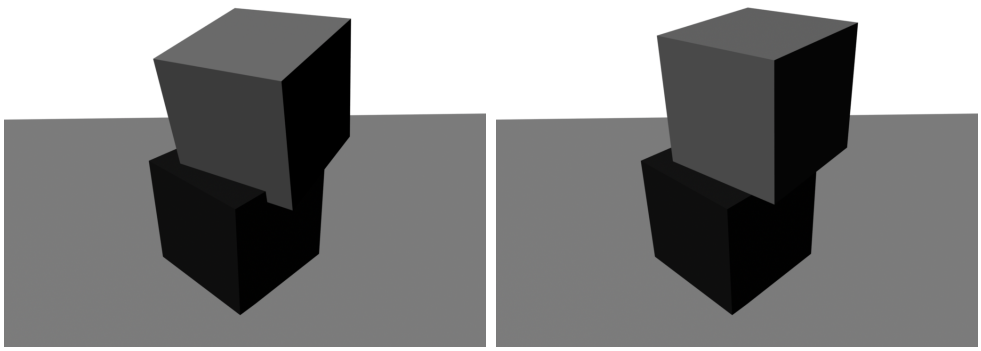


Figure 4.32: First order world simulation used to correct penetration error. The left figure shows initial state, while the right shows the corrected state. Observe that when corrected, the upper box is both translated and rotated.

where

$$\vec{v} = \frac{\vec{P}}{m}, \quad \text{and} \quad \vec{\omega} = \mathbf{I}^{-1}\vec{L}, \tag{4.241}$$

or equivalently we have,

$$\vec{s}(t) = \begin{bmatrix} \vec{r} \\ q \\ \vec{v} \\ \vec{\omega} \end{bmatrix}, \quad \text{and} \quad \dot{\vec{s}}(t) = \begin{bmatrix} \vec{v} \\ \frac{1}{2}\vec{\omega}q \\ \frac{\vec{F}}{m} \\ \mathbf{I}^{-1}\left(\vec{\tau} - \vec{\omega} \times L\right) \end{bmatrix}. \tag{4.242}$$

In a first order system the state function for a rigid body simplifies to,

$$\vec{s}(t) = \begin{bmatrix} \vec{r} \\ q \end{bmatrix}, \quad \text{and} \quad \dot{\vec{s}}(t) = \begin{bmatrix} \frac{\vec{F}}{m} \\ \frac{1}{2}\mathbf{I}^{-1}\vec{\tau}q \end{bmatrix}. \tag{4.243}$$

Observe that the difference is that force relates directly to velocity, i.e.,

$$\vec{F} = m\vec{v}, \tag{4.244}$$

$$\vec{\tau} = \mathbf{I}\vec{\omega}. \tag{4.245}$$

These are the first order world equivalents to the Newton-Euler equations of motion. Thus, in a first order world, there is velocity but no acceleration, and first order worlds are therefore very useful for evaluating system kinematics. In a first order system the dynamics equation $\vec{F} = m\vec{v}$ dictates that objects have no intrinsic velocity of their own. Equivalently, the velocity at any given instant depends completely on the forces acting at that instant. This means that velocity based damping and friction are non-existent in a first order world, and that inertial forces due to velocity are absent. As we will see later, these consequences greatly simplifies the contact modeling compared to a second order world obeying the Newton-Euler equations of motion.

4.16.1 Single Point of Contact

Now let us study a single point of contact, $\vec{p}_k$, between two bodies i and j. Let $\vec{r}_i$ and $\vec{r}_j$ be the position of the center of mass of the bodies. Then the two vectors from the center of mass to the point of contact are found as

$$\vec{r}_{ki} = \vec{p}_k - \vec{r}_i, \tag{4.246}$$

$$\vec{r}_{kj} = \vec{p}_k - \vec{r}_j. \tag{4.247}$$

The change in velocity of the contact point with respect to each body are,

$$\Delta\vec{u}_i = \Delta\vec{v}_i + \Delta\vec{\omega}_i \times \vec{r}_{ki}, \tag{4.248}$$

$$\Delta\vec{u}_j = \Delta\vec{v}_j + \Delta\vec{\omega}_j \times \vec{r}_{kj}. \tag{4.249}$$

In a first order world, a change in velocity corresponds to a change in force. For the time being, we will ignore all other forces in the system, except for the forces acting at

the contact point. This means that the change in force is simply the force acting at the contact point, and for body j we have

$$\Delta\vec{v}_j = \frac{\vec{F}}{m_j}, \tag{4.250}$$

$$\Delta\vec{\omega}_j = \mathbf{I}_j^{-1}\left(\vec{r}_{kj} \times \vec{F}\right), \tag{4.251}$$

where $\vec{F}$ is the force acting on body j. Assuming the law of reaction equals action, we must have that the force on body i is given by $-\vec{F}$, yielding the following equations for body i,

$$\Delta\vec{v}_i = -\frac{\vec{F}}{m_i}, \tag{4.252}$$

$$\Delta\vec{\omega}_i = -\mathbf{I}_i^{-1}\left(\vec{r}_{ki} \times \vec{F}\right). \tag{4.253}$$

Now, let us compute the relative change in contact velocity as,

$$\vec{u} = \left(\Delta\vec{u}_j - \Delta\vec{u}_i\right). \tag{4.254}$$

Substituting in previous results yield

$$\vec{u} = \left(\left(\Delta\vec{v}_j + \vec{\Delta}\omega_j \times \vec{r}_{kj}\right) - \left(\Delta\vec{v}_i + \vec{\Delta}\omega_i \times \vec{r}_{ki}\right)\right), \tag{4.255a}$$

$$\vec{u} = \left(\left(\frac{\vec{F}}{m_j} + \left(\mathbf{I}_j^{-1}\left(\vec{r}_{kj} \times \vec{F}\right)\right) \times \vec{r}_{kj}\right) - \left(-\frac{\vec{F}}{m_i} + \left(-\mathbf{I}_i^{-1}\left(\vec{r}_{ki} \times \vec{F}\right)\right) \times \vec{r}_{ki}\right)\right) \tag{4.255b}$$

Using the cross matrix notation for the cross products we can isolate $\vec{F}$ as,

$$\vec{u} = \left(\left(\frac{\vec{F}}{m_j} + \left(\mathbf{I}_j^{-1}\left(\mathbf{r_{kj}}^{\times}\vec{F}\right)\right)\mathbf{r_{kj}}^{\times}\right) - \left(-\frac{\vec{F}}{m_i} + \left(-\mathbf{I}_i^{-1}\left(\mathbf{r_{ki}}^{\times}\vec{F}\right)\right)\mathbf{r_{ki}}^{\times}\right)\right) \tag{4.256a}$$

$$= \left(\frac{1}{m_j} + \frac{1}{m_i}\right)\vec{F} - \left(\mathbf{r_{kj}}^{\times}\mathbf{I}_j^{-1}\mathbf{r_{kj}}^{\times}\right)\vec{F} - \left(\mathbf{r_{ki}}^{\times}\mathbf{I}_i^{-1}\mathbf{r_{ki}}^{\times}\right)\vec{F} \tag{4.256b}$$

$$= \left(\frac{1}{m_j} + \frac{1}{m_i}\right)\vec{F} - \left(\mathbf{r_{kj}}^{\times}\mathbf{I}_j^{-1}\mathbf{r_{kj}}^{\times} + \mathbf{r_{ki}}^{\times}\mathbf{I}_i^{-1}\mathbf{r_{ki}}^{\times}\right)\vec{F} \tag{4.256c}$$

$$= \underbrace{\left(\left(\frac{1}{m_j} + \frac{1}{m_i}\right)\mathbf{1} - \left(\mathbf{r_{kj}}^{\times}\mathbf{I}_j^{-1}\mathbf{r_{kj}}^{\times} + \mathbf{r_{ki}}^{\times}\mathbf{I}_i^{-1}\mathbf{r_{ki}}^{\times}\right)\right)}_{\mathbf{K}}\vec{F} \tag{4.256d}$$

$$\vec{u} = \mathbf{K}\vec{F}. \tag{4.256e}$$

The matrix $\mathbf{K}$ is the familiar collision matrix [100]. Thus, we have derived the same relationship between force and contact point velocity in a first order world as for collision impulses and contact point velocities in a second order world. Measuring the relative contact velocity in only the normal direction, $\vec{n}$, yields

$$u_n = \vec{n}^T\vec{u} = \vec{n}^T\left(\mathbf{K}\vec{F}\right), \tag{4.257}$$

and restricting the contact force $\vec{F}$, to be parallel to the normal direction is simply

$$u_n = \left(\vec{n}^T \mathbf{K} \vec{n}\right) f, \tag{4.258}$$

where f is a scalar denoting the magnitude of the force $\vec{F}$, and likewise u_n is a scalar denoting the magnitude of relative contact velocity in the normal direction.

4.16.1.1 Penetration Correction

The previously derived relation (4.258) provides a convenient way for projecting penetrating bodies out of each other. The idea is as follows: setup (4.258) such that you solve for the force $\vec{F}$, which will yield a change in the relative contact velocity $\vec{u}$ that will correct the penetration during the next time-step h.

Let the penetration depth be given by the distance d. Then the correcting velocity at the contact points must be,

$$u_n = \frac{d}{h}, \tag{4.259}$$

and solving for the correcting force yields,

$$f = \frac{\frac{\vec{d}}{h}}{\vec{n}^T \mathbf{K} \vec{n}}. \tag{4.260}$$

Now use, $\vec{F} = \vec{n} f$, in the state function (4.243) to perform a single forward Euler step. This yields a position update, which will correct the penetration between the two objects, i.e.

$$\vec{r}_i^{\,t+1} = \vec{r}_i^{\,t} + h \frac{\left(-\vec{F}\right)}{m_i}, \tag{4.261}$$

$$q_i^{t+1} = q_i^t + h \frac{1}{2} \left(\mathbf{I}_i^{-1} \left(\vec{r}_{ki} \times \left(-\vec{F} \right) \right) \right) q_i^t, \tag{4.262}$$

$$\vec{r}_j^{\,t+1} = \vec{r}_j^{\,t} + h \frac{\vec{F}}{m_j}, \tag{4.263}$$

$$q_j^{t+1} = q_j^t + h \frac{1}{2} \left(\mathbf{I}_j^{-1} \left(\vec{r}_{kj} \times \vec{F} \right) \right) q_j^t. \tag{4.264}$$

In the above, we have applied a forward Euler scheme, meaning that we have linearized the sought displacements. The correction is therefore only correct to within first order, but in practice this is seldom a problem, since most simulation paradigms try to prevent penetrations. This indicates that penetration errors are likely to be small and only require small corrections.

4.16.1.2 Continuous Motion

The constraint forces needed in a first order world simulation can also be derived from (4.256e). Taking external forces into account, say forces $\vec{F}_{\text{ext}}^{\,i}$ and $\vec{F}_{\text{ext}}^{\,j}$ and torques $\vec{\tau}_{\text{ext}}^{\,i}$ and $\vec{\tau}_{\text{ext}}^{\,j}$ acting on bodies i and j respectively, then it is seen by substituting the forces into (4.255a) that (4.256e) changes into the form,

$$\vec{u} = \mathbf{K} \vec{F} + \vec{b}, \tag{4.265}$$

where $\vec{b}$ is given by,

$$\vec{b} = \frac{\vec{F}^{\,j}_{\text{ext}}}{m_j} - \frac{\vec{F}^{\,i}_{\text{ext}}}{m_i} + \mathbf{I}_j^{-1}\vec{\tau}^{\,j}_{\text{ext}} \times \vec{r}_{kj} - \mathbf{I}_i^{-1}\vec{\tau}^{\,i}_{\text{ext}} \times \vec{r}_{ki}. \tag{4.266}$$

Here, $\vec{F}$ denotes the constraint force at the contact point, which must prevent penetration from occurring. This implies that the relative contact velocity in the normal direction must be non-negative, since a first order world is frictionless. According to the principle of virtual work, it makes sense to require that the constraint force is parallel to the normal direction. Also, the bodies are allowed to separate, which means that the constraint force can only be repulsive. To summarize, we have

$$u_n \geq 0, \quad \text{and} \quad f \geq 0. \tag{4.267}$$

Furthermore, u_n and f must be complementary, meaning that if u_n initially is greater than zero, there is no need for a constraint force, i.e.

$$u_n > 0 \Rightarrow f = 0. \tag{4.268}$$

On the other hand, if we have a constraint force acting at the contact point, the normal velocity must be zero. That is,

$$f > 0 \Rightarrow u_n = 0. \tag{4.269}$$

In conclusion, we discover a linear complementarity condition similar to (4.22).

4.16.2 Multiple Contact Points

In this section, we will extend the machinery for a two body problem to handle a n-body system.

Let $\vec{u}_k$ denote the change in relative contact velocity and at the k'th contact, then

$$\vec{u}_k = \Delta\vec{u}_{j_k} - \Delta\vec{u}_{i_k}. \tag{4.270}$$

As previously we also have,

$$\Delta\vec{u}_{i_k} = \Delta\vec{v}_{i_k} + \Delta\vec{\omega}_{i_k} \times \vec{r}_{ki_k}, \tag{4.271}$$
$$\Delta\vec{u}_{j_k} = \Delta\vec{v}_{j_k} + \Delta\vec{\omega}_{j_k} \times \vec{r}_{kj_k}. \tag{4.272}$$

However, this time we do not have a single contact force contributing to the k'th contact. To get the total force and torque contributing to the k'th contact, we have to sum over all forces and torques. Using the indexing introduced in Section 4.3 we write,

$$\Delta\vec{v}_{j_k} = \frac{\vec{F}^{j^k}_{\text{ext}} + \sum_{h,j_h=j_k}\vec{F}_h - \sum_{h,i_h=i_k}\vec{F}_h}{m_{j_k}}, \tag{4.273}$$

$$\Delta\vec{\omega}_{j_k} = \mathbf{I}_{j_k}^{-1}\left(\vec{\tau}^{j_k}_{\text{ext}} + \left(\sum_{h,j_h=j_k}\vec{r}_{hj_h} \times \vec{F}_h\right) - \left(\sum_{h,i_h=j_k}\vec{r}_{hi_h} \times \vec{F}_h\right)\right), \tag{4.274}$$

where $\vec{F}_h$ is the contact force at the h'th contact point. If we perform the same substitutions as done in the case of a single point of contact (4.265), the contributions from

the external forces and torques can be collected in a single vector $\vec{b}_k$. The contribution stemming from the contact forces can then be written as a linear combination,

$$\vec{u}_k = \left(\sum_h \mathbf{K}_{kh} \vec{F}_h \right) + \vec{b}_k, \tag{4.275}$$

where

$$\mathbf{K}_{kh} = \begin{cases} \left(\frac{1}{m_{j_k}} + \frac{1}{m_{i_k}}\right)\mathbf{1} - \left(\mathbf{r_{kj_k}}^{\times}\mathbf{I}_{j_k}^{-1}\mathbf{r_{hj_k}}^{\times} + \mathbf{r_{ki_k}}^{\times}\mathbf{I}_{i_k}^{-1}\mathbf{r_{hi_k}}^{\times}\right) & \text{if } i_h = i_k \text{ and } j_h = j_k, \\ \left(\frac{-1}{m_{i_k}}\right)\mathbf{1} + \left(\mathbf{r_{ki_k}}^{\times}\mathbf{I}_{i_k}^{-1}\mathbf{r_{hi_h}}^{\times}\right) & \text{if } i_h = i_k \text{ and } j_h \neq j_k, \\ \left(\frac{-1}{m_{j_k}}\right)\mathbf{1} + \left(\mathbf{r_{kj_k}}^{\times}\mathbf{I}_{j_k}^{-1}\mathbf{r_{hi_h}}^{\times}\right) & \text{if } i_h = j_k \text{ and } j_h \neq i_k, \\ \left(\frac{1}{m_{i_k}}\right)\mathbf{1} - \left(\mathbf{r_{ki_k}}^{\times}\mathbf{I}_{i_k}^{-1}\mathbf{r_{hj_h}}^{\times}\right) & \text{if } j_h = i_k \text{ and } i_h \neq j_k, \\ \left(\frac{1}{m_{j_k}}\right)\mathbf{1} - \left(\mathbf{r_{kj_k}}^{\times}\mathbf{I}_{j_k}^{-1}\mathbf{r_{hj_h}}^{\times}\right) & \text{if } j_h = j_k \text{ and } i_h \neq i_k, \\ \mathbf{0} & \text{otherwise.} \end{cases} \tag{4.276}$$

If we only consider relative contact velocity in the normal direction and only allow contact forces to be parallel with the contact normals, it is easily seen that the equation transforms into

$$u_{nk} = \vec{n}_k^T \vec{u}_k = \left(\sum_h \vec{n}_k^T \mathbf{K}_{kh} \vec{n}_h f_h \right) + \vec{b}_k, \tag{4.277}$$

where f_h denotes the magnitude of the h'th contact force. The equations for all the relative normal contact velocities can now be written as in a single matrix equation,

$$\underbrace{\begin{bmatrix} u_{n0} \\ \vdots \\ u_{nK-1} \end{bmatrix}}_{\vec{a}} = \underbrace{\begin{bmatrix} \vec{n}_0^T\mathbf{K}_{00}\vec{n}_0 & \dots & \vec{n}_0^T\mathbf{K}_{0K-1}\vec{n}_{K-1} \\ \vdots & & \vdots \\ \vec{n}_{K-1}^T\mathbf{K}_{K-10}\vec{n}_0 & \dots & \vec{n}_{K-1}^T\mathbf{K}_{K-1K-1}\vec{n}_{K-1} \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} f_0 \\ \vdots \\ f_{K-1} \end{bmatrix}}_{\vec{x}} + \underbrace{\begin{bmatrix} \vec{n}_0^T\vec{b}_0 \\ \vdots \\ \vec{n}_{K-1}^T\vec{b}_{K-1} \end{bmatrix}}_{\vec{b}}, \tag{4.278}$$

$$\Rightarrow \vec{a} = \mathbf{A}\vec{f} + \vec{b}. \tag{4.279}$$

This equation is similar to (4.26), and recalling the definition of the contact Jacobian in Section 4.9.7 and the assembly process in Section 4.13, we immediately generalize,

$$\mathbf{A} = J_{\text{contact}} \mathbf{M}^{-1} J_{\text{contact}}^T, \tag{4.280}$$

$$\vec{b} = J_{\text{contact}} \mathbf{M}^{-1} \vec{f}_{\text{ext}}. \tag{4.281}$$

Notice that in a first order world $\vec{f}_{\text{ext}}$ does not contain inertial forces. An LCP can now be formulated for the continuous movement of a first order world, as

$$\vec{a} \geq 0, \vec{f} \geq 0, \quad \text{and} \quad \vec{f}^T\vec{a} = 0. \tag{4.282}$$

A penetration correcting force can be computed by redefining the $\vec{b}$-vector,

$$\vec{b} = \frac{1}{h} \begin{bmatrix} -d_0 \\ \vdots \\ -d_{K-1} \end{bmatrix}, \tag{4.283}$$

and then solving the same LCP as we formulated above. For the case of a single point contact, we could solve the linear system, but for multiple contact points the projection of a penetration of one contact point might fix the penetration of another, and this induces a complementarity condition where there can be no correcting force if there is no penetration. On the other hand, if there is a correcting force, the penetration depth must be zero.

Chapter 5

Contact Graphs in Multibody Dynamics Simulation

Historically, contact graphs are used for splitting objects into disjoint groups that can be simulated independently. Contact graphs are frequently mentioned between people working with rigid body simulation, as can be seen by searching through the archives of [1], but they are often not formally described in the literature. For instance, [101] uses the word "contact group", but nowhere is it explained. Most of the time people just mention the idea of using contact groups to break down contact force computations into smaller independent problems [9, 38]. The benefit of doing this is obvious that not many people would spend a lot of time explaining it. To our knowledge, [103] is the first advanced attempt to use contact groups for other things than contact force computations, and the first use of the word "graph" appeared in [71], where a contact graph is used to properly back-up penetrating objects in the simulation. In our opinion, this is the first example of a primitive time-control algorithm using contact graphs. Recently, [70] developed a shock propagation algorithm for the efficient handling of stacked objects which uses a contact graph. The contact graph in [70] is constructed in a different manner than the one described here.

Today, simulators exploit contact groups for breaking down the computations into smaller independent problems. For instance, Open Dynamics Engine (ODE) (v 0.035) and Vortex (v 2.0.1) from CMLabs compute contact groups which they call islands and partitions respectively. However, they do not store an actual graph data structure as the one we propose in this chapter.

Not very surprisingly, alternatives to contact graphs are neither mentioned nor explained. The closest thing to an alternative appears to be putting the contact-matrix into block-form as briefly described in [24]. As far as we know this is another idea that is not well described in the computer graphics literature. In comparison with the contact graph approach, the "block-form" matrix approach is limited to contact force and collision impulse computations, and can not be used for anything else in a simulator.

Lastly, we feel that contact graphs are a good companion for our rigid body simulator modular design, see [45, 56], and as such they are a step further in the direction towards a more standardized and powerful framework.

The contact graph algorithm we present in this section is part of the Spatial-Temporal Coherence (STC) analysis module. The algorithm shows that STC analysis is scattered

throughout the other phases of the collision detection engine. We use contact graphs for caching information, such as contact points. The cached information can be used to improve the run time performance of a rigid body simulator.

5.1 The Contact Graph

A contact graph consists of a set of nodes, where a node is an entity in the configuration, such as a rigid body or a fixed body. However, a node can also be a virtual entity. That is, something which does not have a physical influence on other entities in the configuration. For instance, trigger volumes, i.e. volumes placed in the physical world, raise events when other "physical" objects move in and out of them. Nodes could also be something without a shape, for instance a timer event. Another example of a shapeless node could be logical rules, such as grouping of configuration objects and/or filters.

The node types are easily divided into three categories: Physical nodes, Container nodes, and Logical nodes. Table 5.1 shows the node types and their respective categories together with the symbolic notation we use. The physical nodes are those nodes repre-

Category	Type	Symbol
Physical	Rigid Bodies	(R)
	Fixed Bodies	(F)
	Link Bodies	(L)
	Scripted Bodies	(S)
Container	Composite bodies	(C)
	Multibodies	(M)
Logical	Logical Rules	(A)
	Trigger Volumes	(V)
	Timers	(T)

Table 5.1: Node types.

senting entities which can physically interact with each other. Rigid bodies, fixed bodies, scripted bodies (see Chapter 3) and link bodies are all physical objects. Rigid bodies can be rigidly attached to each other to form a composite body. Therefore a composite body is therefore a "container" type. Link bodies and joints can form a jointed mechanism which is called a multibody or articulated figure, therefore a multibody is also a "container" type. Logical rules, trigger volumes and timers are all nodes which do not have a physical meaning. They are used for generating events which have logical consequences in the sense that an end user reacts to them, and constraining physical interactions to a specified set of objects. We call all such kind of nodes logical nodes.

When objects interact with each other, contact information is usually computed and cached. It is particularly easy to use the edges in the contact graph for storing information of interactions between objects. Edges are also useful for keeping structural and proximity information.

All the edge types are listed in Table 5.2. An edge between a logical rule and a physical object means that the logical rule applies to the physical object. This sort of edge is static in the sense that it is defined by an end user prior to the simulation.

Category	Type
Logical	Rule vs. Rigid
	Rule vs. Fixed
	Rule vs. Scripted
	Rule vs. Link
Structural	Rigid vs. Composite
	Link vs. Multi
Geometrical	Trigger vs. Rigid
	Trigger vs. Fixed
	Trigger vs. Scripted
	Trigger vs. Link
Physical	Rigid vs. Rigid
	Rigid vs. Fixed
	Rigid vs. Scripted
	Rigid vs. Link
	Fixed vs. Link
	Scripted vs. Link
	Link vs. Link

Table 5.2: Edge types.

An edge between a composite body and a rigid body tells that the rigid body is part of a composite body. This kind of edge gives us structural information about how the composite body is built, and it is a static edge, i.e. defined prior to simulation by an end user.

An edge between a multibody and a link indicates that the link is part of the multibody. Again, the edges are static, giving us structural information about a multibody.

An edge between a physical object and a trigger volume indicates that the physical object has moved inside the trigger volume. Therefore, this kind of edge can be used to generate trigger volume event notifications. This type of edge is a dynamic edge, meaning that it is inserted and removed dynamically by the collision detection engine during the simulation.

The last type of edge we can encounter are those which tell us something about how the objects in the configuration interact with each other at this moment in time. For instance, if two rigid bodies come into contact then an edge is created between them. There are some combinations of edges which do not make sense, such as an edge between two fixed bodies.

In order to access cached information unambiguously and fast, nodes and edges must be found in constant time, and edges are bidirectionally and uniquely determined by the two nodes they connect. These properties can be obtained by letting every entity in the configuration have a unique index, and letting edges refer to these indices, such that the smallest indexed entity is always known as A and the other as B.

Figure 5.1 contains a pseudo-code outline of the contact graph data structure. All objects should be inherited from the Node class such that they can be inserted directly into the contact graph.

It is fairly easy to visualize a contact graph. Figure 5.2 shows a small complex example

```
Class Node
  int idx
  Enum {...} type
  List<Edge> edges
Class Edge
  int idxA
  int idxB
Class ContactGraph
  Hashtable<Node> nodes
  Hashtable<Edge> edges
```

Figure 5.1: Contact graph data structures.

of a contact graph.

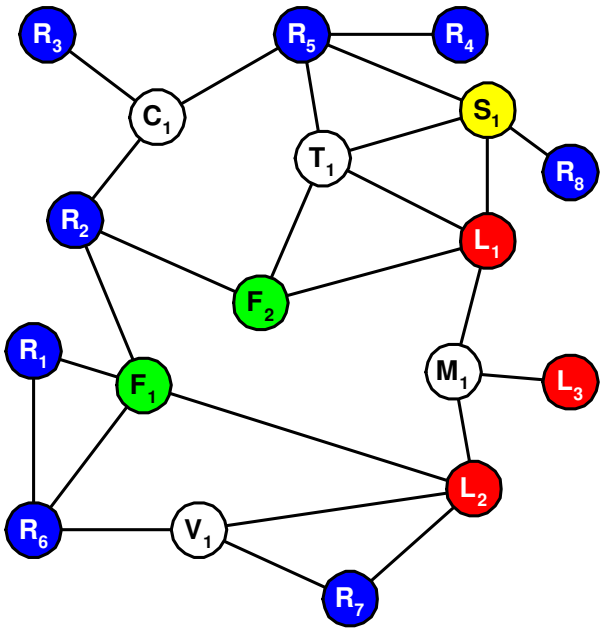


Figure 5.2: A contact graph example. The symbolic notation is listed in Table 5.1.

5.2 The Contact Graph Algorithm

We will now outline how a contact graph can be used in the collision detection pipeline. Notice, that although we claim a contact graph to be a higher order contact analysis phase, it is not a phase that is isolated to a single place in the pipeline. Instead, it is spread out to all the other phases, i.e. in between the broad phase, narrow phase and contact determination modules.

In the following subsections we will walk through what happens in the collision detection pipeline step by step.

5.2.1 Edge Insertion and Removal

The first step in our algorithm is to update the edges in the contact graph, which is done by looking at the results of the broad phase collision detection algorithm. The result of the broad phase collision detection algorithm is an unsorted list of pairs of nodes, where each pair denotes a detected overlap according to the broad phase algorithm. Observe that each pair is equivalent to a contact graph edge. Therefore, we can insert new edges into the contact graph which we have not seen before. At the same time, we can handle all close proximity information. That is, detection of vanished, persistent, and new close proximity contacts. This is done by comparing the state of edges with their old state. The pseudo-code in Figure 5.3 outlines the general idea. The notation e_{old} in Figure 5.3

```
O = BroadPhase.getOverlaps()
for all edges e ∉ O do
  mark e as vansihed close proximity
  if touching(e_old) then
    mark e as vansihed touching contact
  end if
  if obselete(e) then
    remove e from graph
  end if
next e

for all e ∈ O and e ∈ graph do
  mark e as persistent close proximity
next e
for all e ∈ O and e ∉ graph do
  create edge e in graph
  mark e as new close proximity
next e
```

Figure 5.3: Edge insertion and removal.

refers to the "state" of the edge in the previous iteration. It is not a new instance.

The main idea behind removing edges is to avoid the case of edges accumulating to $O(n^2)$ size. So when the nodes between an edge are far apart, and it is unlikely that they come close in the near future, it is "safe" to remove the corresponding edge. We call this "obsolete testing".

In our simulator we apply a heuristic approach to obsolete testing by simply requiring that the orthogonal distance along the axes between the $AABB$s of the corresponding nodes be twice the maximum edge size of the $AABB$s. The test is based on the idea that edges which exist for a long period of time correspond to close objects. Of course one could use more sophisticated tests which, for instance, used information about time-step and velocities. We favor the simple test, because it is computationally inexpensive, since it only uses a couple of subtractions and comparisons. Notice, that if a bad test is chosen, it will result in the same edge being removed and inserted over and over again. The test is thus related only to performance.

5.2.2 Logical and Coherence Testing

We can perform logical testing and exploit caching by scanning through all the reported overlaps and remove those overlaps we do not have or want to treat any further. In this phase logical rules are applied. Any kind of logical construct could be used such as: "Ignore all interactions between objects in group X and/or group Y".

Overlaps with passive objects are also removed. Passive objects do not really exist in the configuration. They are merely objects kept in memory in case they should become active later on. In this way, objects can be preallocated, and there is no penalty in reallocating objects that dynamically enter and leave the configuration during runtime. We refer to objects using the passive/active scheme as being light weighted. The opposite is called heavy weighted, and it means that objects are explicitly deallocated and reallocated whenever they are added or removed from the configuration. One drawback of light weighted objects is that there is a penalty in the broad phase collision detection algorithm. Fortunately, broad phase collision detection algorithms often have linear running time with very low constants, so the penalty is negligible.

The last screening test is for change in relative placement. Every edge stores a transform, xform$(\cdot)$, indicating the relative placement of the end node objects. If the transform is unchanged then there is no need to run narrow phase collision detection nor contact determination, because these algorithms would return the exact same results as in the previous iteration. This is illustrated in Figure 5.4. Notice that when the relative place-

```
N = empty set
for each e ∈ O do
  if not rule(e) then
    continue
  else if A(e) and B(e) are triggers then
    continue
  else if A(e) or B(e) is passive then
    continue
  else if not xform(e)  is changed then
    if touching(e) then
      mark e as persistent
    end if
    if penetration(e) and ShotCircuit then
      terminate
    end if
    continue
  end if
  add e to N
next e
```

Figure 5.4: Logical and coherence testing.

ment of objects is unchanged, it is then tested if objects are in a persistent touching contact.

5.2.3 Narrow Phase and Short Circuiting

We are now ready for doing narrow phase collision detection and contact determination on the remaining overlaps. Output from these algorithms are typically a set of feature pairs forming principal contacts, PCs, and a penetration state. The edges of the contact graph provide a good place for storing this kind of information. The output of the narrow phase should of course also be cached in the edge, because most narrow phase collision detection algorithms reuse their results from the previous iteration to run in constant time. Sometimes, the closest principal contact is needed. For instance when using impulse based simulation or estimating time of impact. At this stage it is therefore possible to search the output of the narrow phase for the the closest principal contact. Next, it is tested if any contact state changes occurred, such as if touching or penetrating contacts vanish or persist. That is, if a contact was also present in the last iteration. If one of the nodes is a trigger volume then we do not mark touching contact, but rather *in-* and *out-* events of the trigger volume. The same applies to the marking that took place earlier. The pseudo-code is shown in Figure 5.5. As it can be seen in Figure 5.5 the

```
for each e ∈ N do
  NarrowPhase.run(e, PCs(e))
  if penetration(e) and ShotCircuit then
    terminate
  end if
  if not only proximity info then
    ContactDetermination.addSeed(e, PCs(e))
  end if

  minPC(e) = min{PCs(e)}

  if not separation(e_old) then
    if not separation(e) then
      mark e as persistent touching contact
    else
      mark e as vanishing touching contact
    end if
  else
    if not separation(e) then
      mark e as new touching contact
    end if
  end if
next e
```

Figure 5.5: Narrow phase and short circuiting.

output from the narrow phase collision detection, $PCs(e)$, are often used as a seed for the contact determination. This is why the method *addSeed()* is invoked after the narrow phase collision detection has been run. The meaning of the surrounding if-statement will be explained in the next section.

5.2.4 Contact Determination

Finally, we can run the contact determination for all those edges, where their end node objects are not separated. The pseudo-code is shown in Figure 5.6. Observe the out-most

```
if not only proximity info then
  for each e ∈ N do
    if A(e) and B(e) are physical then
      if not separated(e) then
        ContactDetermination.run(e)
      end if
    end if
  next e
end if
```

Figure 5.6: Contact determination.

if-statement in the pseudo-code. In an impulse based simulator it is often not necessary to do a full contact determination. Only the closest points are actually needed [100], so an end user might want to turn off contact determination completely.

In the pseudo-code we have chosen to skip contact determination on nodes representing things like trigger volumes. Such entities are merely used for event notification, so there is no need for contact determination.

5.2.5 The Contact Groups

Now that we have completed exploiting the logical and caching benefits we can gain from a contact graph, we are now ready to use the contact graph for its intended purpose: determining contact groups. The actual contact groups are found by a traditional connected components search algorithm, restricted to the union of the list N introduced in Figure 5.4, and the structural edges. The algorithm works by first marking all edges that should be traversed as "white". Afterwards, edges are treated one by one until no more white edges exist. The pseudo-code is shown in Figure 5.7 and 5.8. In Figure 5.7 we have

```
if should compute groups then
  for all edges, e ∈ N do
    color(e) = white
  next e
  for all edges, e ∈ N do
    if color(e) = white then
      let G be an empty group
      traverseGroup(e, G)
    end if
  next e
end if
```

Figure 5.7: Connected components search.

once more placed a surrounding if-statement in order to provide the most flexibility for an end user. Fixed and scripted bodies are rather special, and they behave as if they

```
algorithm traverseGroup(e:Edge,G:Group)
  color(e) = grey
  add e to G
  for end nodes, n ∈ e do
    if not n is fixed or scripted body then
      for all edges, w ∈ n do
        if color(w) = white then
          traverseGroup(w, G)
        end if
      next e
    end if
  next n
  color(e) = black
end algorithm
```

Figure 5.8: Traverse group

had infinite mass, such that they can support any number of bodies without ever getting affected themselves. They work like an insulator, which is why we ignore edges from these nodes when we search for contact groups.

Let us look at the contact groups of the example from Figure 5.2. As it can be seen in Figure 5.9, we have four contact groups A, B, C, and D.

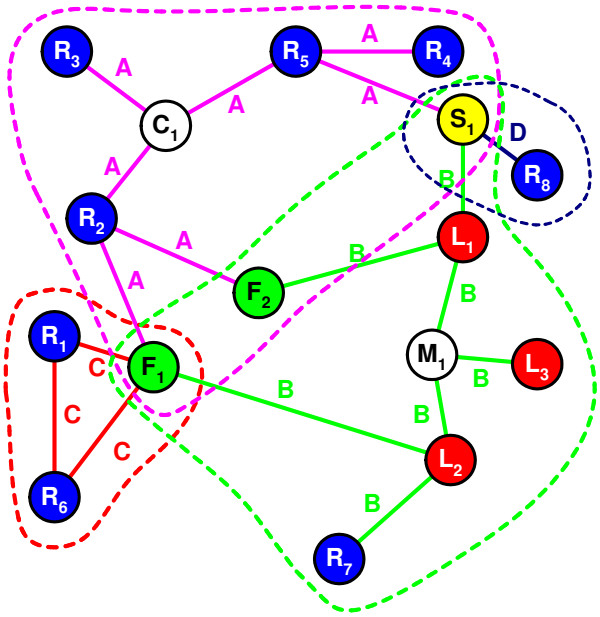


Figure 5.9: Example contact groups.

5.3 The Event Handling

In the pseudo-code we have outlined so far, we have not explicitly stated when events get propagated back to an end user. Instead, we have very clearly shown when and how the events should be detected. Table 5.3 summarizes the types of events we have talked about. We can traverse the edges of the graph, and simply generate the respective event

In Trigger Volume
Out Trigger Volume
New Touching Contact
Persistent Touching Contact
Vanishing Touching Contact
New Proximity
Persistent Proximity
Vanishing Proximity
Timer Tick

Table 5.3: Event types.

notifications for all those edges that have been marked with an event. This is shown in Figure 5.10. The only problem is those edges we removed due to the obsolete testing.

```
for each edge e ∈ graph do
  if marked(e) then
    for each mark m ∈ e do
      generateEvent(m, e)
      marked(e_old) = marked(e)
    next m
  end if
next e
```

Figure 5.10: Event handling.

However, this can be handled gracefully by only allowing edges to become obsolete if they were at least marked as vanishing close proximities last time the collision detection query was run.

There is one major subtlety to event handling: some simulators are based on backtracking algorithms, also called retroactive detection, meaning that they keep on running forward until something goes wrong, and then they backtrack, correct things, and then go forward once again. This behavior could occur many times during the simulation of a single frame, and the consequences are that we might detect events which are disregarded.

The problem of backtracking can be handled in two ways. In the first solution, events can be queued during the simulation together with a time-stamp indicating the simulation time at which they were detected. During backtracking, one simply dequeues all events with a time-stamp greater than the time the simulator backtracks to. After having dequeued the events, one would have to reestablish the "marked" state of the edges at that time which can be done by scanning through the queue. In the second solution, events

are restricted to only be generated when it is "safe", i.e. whenever a backtrack cannot occur or on completion of the frame computation. Therefore the e_{old} state should only be updated at these places, such that the events that are generated reflect the changes since the last time events were generated.

The second solution would clearly miss events which the first method might catch, and as the time between event generation gets bigger, it will probably miss even more events. This is not because the events that are returned, indicate a faulty picture of what has occurred, they merely show the same picture but with less detail. On the other hand, the first solution is capable of catching more details at the cost of dynamic memory allocation, something we really would like to avoid.

We favor the second solution, because event notifications are most likely to be used in a gaming context, and in such a context a backtracking algorithm would not be favorable. In predicting motion or validating offline simulation, event notifications might not even be used, so fever details might not be an issue in such contexts.

As a final remark, we should note that overshooting and missing contact transitions will occur with both solutions, due to the fact that the collision detection is only invoked at discrete times during a simulation. From this viewpoint, the second solution can be made just as detailed as one wants, by lowering the time-step of the simulator at an added performance degradation.

5.4 The Spatial-Temporal Coherence Analysis Module

Having outlined how the contact graph should be used in the collision detection pipeline, we can now sketch how a STC analysis module that works together with the three other modules in a collision detection engine. That is, the broad phase collision detection module, the narrow phase collision detection module, and the contact determination module. Figure 5.11 illustrates the interaction, from which we see that the STC analysis occurs in three phases, post-broad-phase, post-narrow-phase, and post-contact-determination. We have not used a pre-broad-phase in the STC analysis, but if any initialization is supposed to take place, then a pre-broad-phase analysis would be a good place for doing this.

We have treated the narrow phase collision detection in an one-by-one approach. Other narrow phase collision detection algorithms handles all pairs of overlap at the same time, see e.g. [77, 114]. However, it is rather straightforward to modify the pseudo-code we have outlined to accommodate this behavior. Simply rewrite the *for*-loop in Figure 5.5, such that invocation of the narrow phase collision detection algorithm occurs before the *for*-loop and works simultaneously on all overlaps.

5.5 Using Contact Groups

In our opinion, there are basically three different ways to exploit the contact groups in rigid body simulation. We will briefly talk about them in the following.

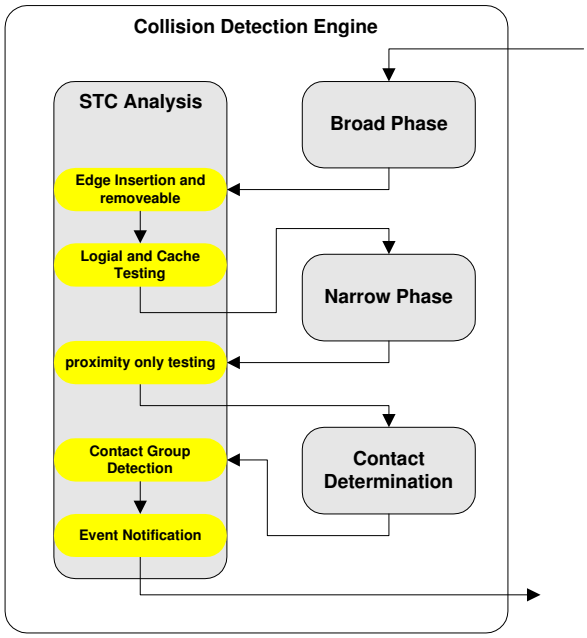


Figure 5.11: Spatial-temporal coherence analysis module.

Time Warping Traditionally, one would backtrack the entire configuration when an illegal state is found, such as the penetration of two objects.

This is very inefficient, since there might be a lot of objects in the scene, whose motion is completely independent of the two violating objects. The result of backtracking them is simply that one would have to repeat the previous computations, thus wasting computation time.

Knowing the contact groups, one could only backtrack those contact groups, where the violating objects belong to, and leave all other contact groups alone. The reader should refer to [103] for more details.

Subdivision of Contact Force Computation Constraint-based methods for computing contact forces are often NP-hard, so it is intractable to solve large problems. However, the contact forces needed in one contact group is totally independent of all the other contact groups. This knowledge can be exploited, and instead of computing the contact forces for all contact groups, the problem is broken down into smaller problems by solving for the contact forces of each contact group separately [9, 38].

Caching Contact Forces If contact forces from the previous iteration are cached in the contact graph edges then these forces can be used as initial guesses for the contact force computation in the current iteration [14].

The contact graph also holds information about change of relative placement. This can also be exploited, because this means that the contact forces are the same as in the previous iteration, so these can simply be reused. Of course, contact forces are dependent on external forces, so one could only exploit this idea if the current absolute placement and external forces "physically" agree with the previous absolute placement. As an example, think of a stack of books on a table: the books do not change placement at all. Similarly, for

a block sliding down an inclined plane: The total external force on the block is independent of the blocks motion down the plane. An example where this does not hold could be a block sliding down an inclined plane, where the inclination angle decreases as the block slides down the plane.

5.6 Results

We will elaborate on several speed up methods that relies on, or relates to contact graphs. The speed up methods are generally applicable to any kind of rigid body simulator. In order to show the effects we have chosen to extend our own multibody simulator with the speed ups. The multibody simulator used is a velocity based complementarity formulation [140] using distance fields for collision detection. Example code is available from the OpenTissue Project [113]. Our simulator was originally developed for medical applications to simulate skeleton bone movements. Performance was not a concern but accuracy was. In this section, we will focus on performance speed up only. For this reason we have chosen a semi-implicit fixed time-stepping scheme with a rather large time-step, 0.01 second. For a medical application we would have used a fix-point time-stepping scheme and done a convergence analysis to determine the time-step.

Using distance fields for collision detection and the above mentioned time-stepping scheme has one major drawback. During the "fake" position update, objects tend to be deeply penetrating. In these cases a large number of contact points will be generated, and the consequence will be a performance degradation due to the large number of variables that must be resolved. Performance improvements are therefore particularly important, even if real-time simulation is out of our grasp.

We have done several performance measurements and statistics on 120 spheres falling onto an inclined plane with the word "DIKU" engraved. The configuration is shown in Figure 5.12. The total duration of the simulation is 10 seconds. In Figure 5.13, measure-

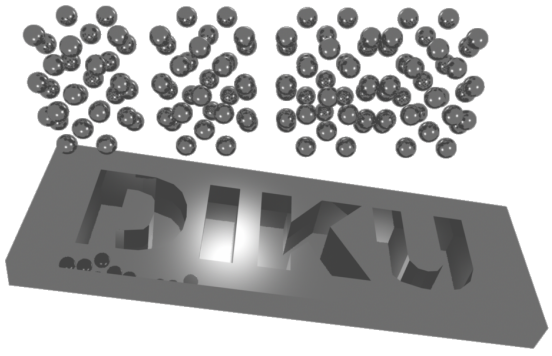


Figure 5.12: 120 falling spheres onto inclined plane with engravings.

ments of the brute force method is shown, i.e. without using a contact graph. Observe that the number of variables and real-life time per iteration are increasing until the point when the spheres settle down to rest, and the curves then flatten out.

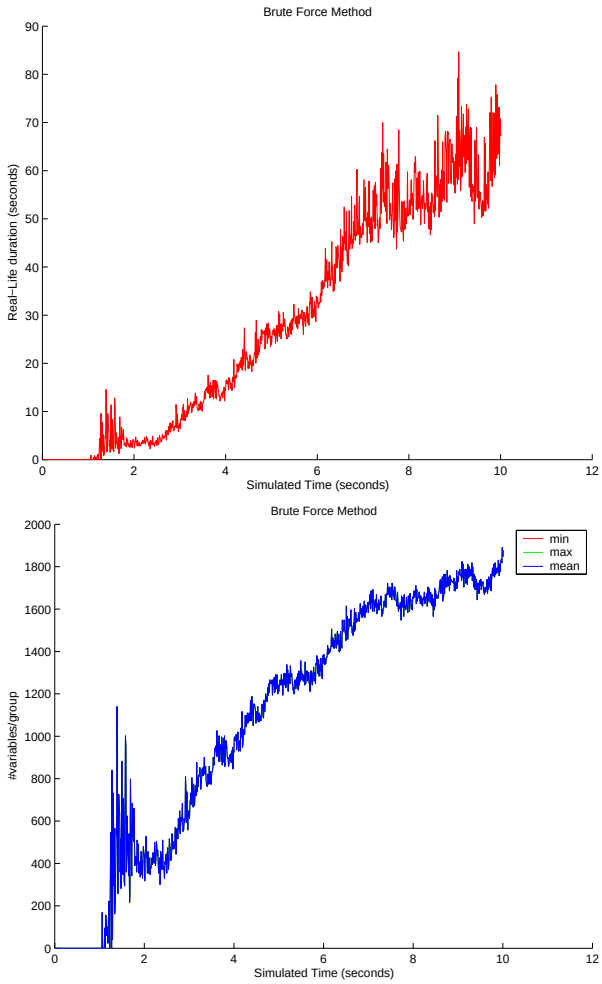


Figure 5.13: The brute force method. In this case there is only one large contact group.

In comparison, Figure 5.14 shows how the curves from Figure 5.13 change when a contact graph is used. Notice that the number of variables per contact group is much smaller than in the brute force method. Also observe the impact on the real-life duration curves. Table 5.4 shows how the total real-life running time in seconds is affected by using a contact graph to divide a simulation into independent contact groups. However, one

	Time (secs.)
Brute Force Method	28424
Contact Graph	1011,4

Table 5.4: The performance effect of dividing simulation into independent contact groups.

can do even better. In the following we will explain 7 more speed up methods we have used successfully.

An obvious improvement comes from ignoring contact groups where all objects appear to be at rest. We call such objects sleepy objects, and we compute them by tracking their

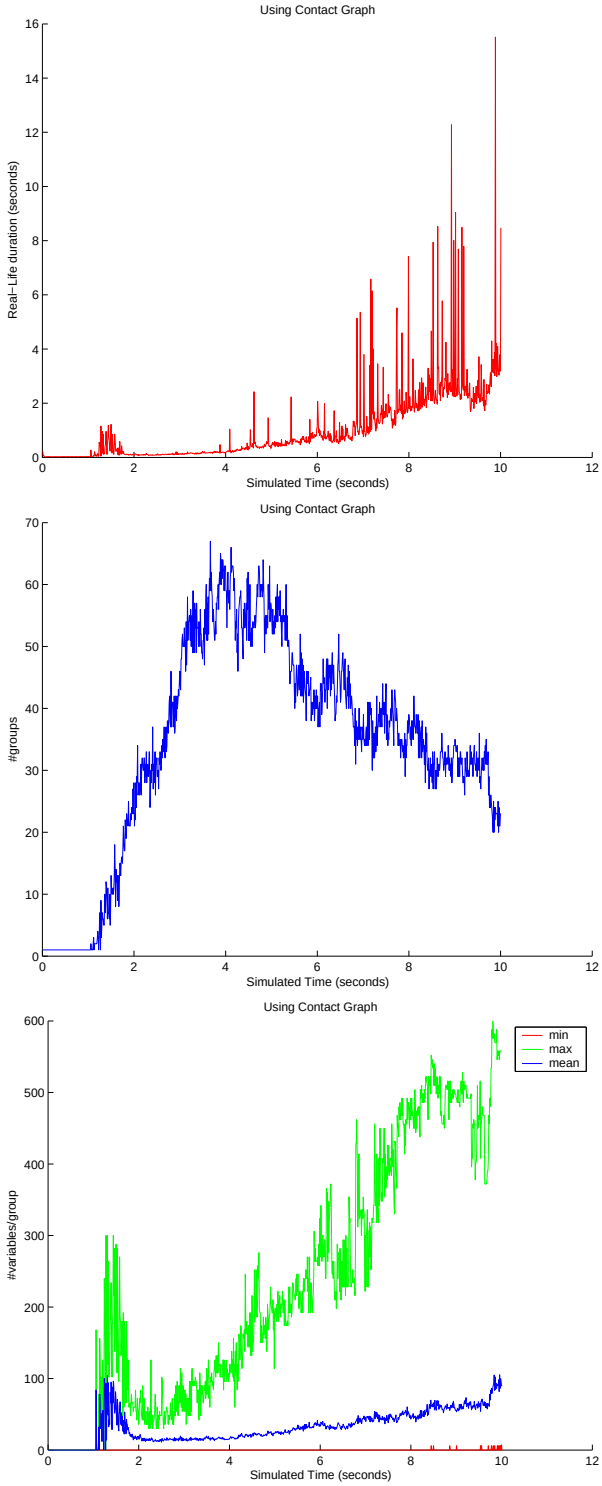


Figure 5.14: The performance impact of using a contact graph to determine independent contact groups.

kinetic energy. An object is flagged as sleepy whenever its kinetic energy has been zero within a numerical threshold over a number of iterations specified by a user. If a contact group only contains sleepy objects the group is completely ignored. Contact graph nodes provide the means for tracking the kinetic energy. Since this speed up is computationally inexpensive, we have invoked it in all of the measurements given in Table 5.5. The speed up could have a potentially disastrous effect on a simulation if the scheme for tagging sleepy objects is not well-designed. Too greedy an approach could leave objects hanging in the air, and too lazy an approach would result in no performance gain. Detailed treatment of sleepy objects are given in Section 6.5.

We exploit contact graph edges for caching contact points and contact forces. Cached contact points are used to skip narrow phase and contact determination, whenever two incident objects of a contact edge are at absolute rest. Cached contact forces are used to warm-start the LCP solver, where we use Path from CPNET [115]. Hopefully, the LCP solver is able to converge much more rapidly. From here on we call this speed up "caching".

A further speed up can sometimes be obtained by limiting the number of times the LCP solver is allowed to iterate. Currently, we change the limit from the default value of 500 to 15. As a consequence, the motion is altered but still looks plausible. The speed up has nothing to do with contact graphs, but it is interesting to examine in combination with the other speed ups we apply, see Table 5.5. We refer to this speed up as "tweaking".

Another speed up we use is to reduce the number of contacts between two objects in contact. The reduction is applied to objects that are deeply penetrating. During the reduction, all contacts are pruned except the single contact of deepest penetration. Again, the edges of the contact graph provide a convenient storage. We have named this speed up "reduction". Reduction has an effect on the motion of the objects we believe that is actually more correct, because intuitively, the deepest point of penetration better resembles the idea of using the minimum translational distance as a separation measure. Besides, theoretical reduction should give a decrease in the number of variables used in the complementarity formulation.

Inspired by the speed up of detecting independent contact groups, further subdivision into groups that could be simulated independently seems feasible. An idea to further subdivide is to prune away sleepy objects from those contact groups containing both non-sleepy and sleepy objects. We refer to this speed up as "subgrouping". The idea is to think of the sleepy object as a fixed object during the computation of the contact groups. Of course, this speed up will have a drastic impact on the motion of the objects. However, from a convergence theory point of view the effect should vanish as the time-step goes to zero. What we have done is to interleave the simulation of subgroups by one frame. Other subgrouping/sleepy object schemes can be found in [24, 130]. To help objects settle down and become sleepy faster, intuitively, it seems to be a good idea to let the coefficient of restitution drop to zero the more sleepy an object gets, meaning that sleepy objects are sticky objects. Currently, we simply set the coefficient of restitution to zero whenever at least one of the incident objects is sleepy. The speed up is referred to as "zeroing". In the same spirit, a linear viscous damping term is added to the motion of all objects in the simulation. The intention is to slow down objects, making them less willing to become non-sleepy. We call this "damping". The contact graph is used for the subgrouping and zeroing. For instance, subgrouping is done by setting the edge color to

black in the pseudo-code of Figure 5.7 whenever the incident objects both are sleepy.

The last method we have applied consists of setting the inverse mass and inertia tensor to zero for all sleepy objects. The main intuition behind this is to "force" sleepy objects to stay sleepy. We have named this "fixation", and it has a dramatic impact on the simulation. Object motion is visually altered as it can be seen from ♠ in Figure 5.15. Applying fixation only makes sense when subgrouping is used. Otherwise, the LCP solver has to solve for contacts between two fixated objects.

Table 5.5 contains performance measurements of all sensible combinations of the speed up methods mentioned previously. Figure 5.15 shows motion results of four selected combinations: ♢,♡,♣, and ♠ from Table 5.4. These are compared to the motion of the brute force method. The four combinations were picked because they resemble the best performance whenever a new speed up was used. Observe that the resulting motion diverges more and more from the brute force method the more speed ups are used. Especially ♠ is different. During the last seconds, objects actually fly up in the air. This is due to the constraint stabilization which corrects large errors in the simulation. Using error correction by projection instead would not cause the object to fly up in the air. In Figure 5.16 a comparison is done between the performance statistics of the four selected combinations ♢,♡,♣, and ♠. Observe that the plots of the first three combinations, ♢, ♡, and ♣, are similar to those shown in Figure 5.14. The fourth combination, ♠, has very different plots for the real-life duration and variables per group plots. These appear to be nearly asymptotically constant.

5.7 Discussion

It is obvious from Table 5.5 that the proper combination of the speed ups is capable of producing a speed up factor of $\frac{28424}{135} \approx 210$, but it is difficult to describe the impact on the resulting motion. However, it is clear that using Contact Graphs, Caching Contact Forces and Sleepy Groups does not change the motion of the brute force method, but all other speed ups we presented changed the physical properties and as a consequence motion is altered, as can be seen in Figure 5.15. Especially, the reduction and the subgrouping speed ups has great impacts on the motion. However, the motion still looks plausible in the author's opinion.

The tagging of sleepy objects can have a rather drastic impact on the simulation, such as leaving objects hanging when they should not. For instance, in the simulation shown in Figure 5.17, near the K-letter, a bunch of spheres land on top of each other. While the top-most spheres rumble off the top, the bottom-most spheres are kept in place and prohibited from gaining kinetic energy. At the end of the simulation, a single sticky sleepy sphere can be seen on the inclined plane. We have to be careful not to make general conclusions based on the measurements in this chapter, since only one configuration has been examined.

It is clear that contact graphs are a valuable extension to a multibody simulator. They can be used for more than finding independent groups of objects. Even in the realm of physical accurate simulation, a speed up factor of the order of 20-30 is not unlikely. Disregarding accuracy completely, the speed up factor can be increased by an order of magnitude.

Cache	Tweak	Reduce	Zero	Damp	Subgroup	Fixate	Time
+	-	-	-	-	-	-	658,499
-	+	-	-	-	-	-	589,523
+	+	-	-	-	-	-	952,519
-	-	+	-	-	-	-	712,567
+	-	+	-	-	-	-	♢ 624,274
-	+	+	-	-	-	-	895,157
+	+	+	-	-	-	-	1157,27
-	-	-	+	-	-	-	840,673
+	-	-	+	-	-	-	771,285
-	+	-	+	-	-	-	1205,34
+	+	-	+	-	-	-	575,343
-	-	+	+	-	-	-	1246,65
+	-	+	+	-	-	-	965,894
-	+	+	+	-	-	-	599,258
+	+	+	+	-	-	-	688,454
-	-	-	-	+	-	-	578,171
+	-	-	-	+	-	-	578,339
-	+	-	-	+	-	-	♡ 528,633
+	+	-	-	+	-	-	533,934
-	-	+	-	+	-	-	788,291
+	-	+	-	+	-	-	890,056
-	+	+	-	+	-	-	789,72
+	+	+	-	+	-	-	1093,42
-	-	-	+	+	-	-	766,438
+	-	-	+	+	-	-	627,623
-	+	-	+	+	-	-	736,771
+	+	-	+	+	-	-	663,246
-	-	+	+	+	-	-	608,286
+	-	+	+	+	-	-	956,992
-	+	+	+	+	-	-	1273,16
+	+	+	+	+	-	-	730,885
-	-	-	-	-	+	-	700,215
+	-	-	-	-	+	-	541,854
-	+	-	-	-	+	-	785,367
+	+	-	-	-	+	-	561,064
-	-	+	-	-	+	-	855,264
+	-	+	-	-	+	-	569,153
-	+	+	-	-	+	-	523,856
+	+	+	-	-	+	-	618,052
-	-	-	+	-	+	-	1077,2
+	-	-	+	-	+	-	761,245
-	+	-	+	-	+	-	760,623
+	+	-	+	-	+	-	903,259
-	-	+	+	-	+	-	767,546
+	-	+	+	-	+	-	1329,69
-	+	+	+	-	+	-	949,809
+	+	+	+	-	+	-	535,479
-	-	-	-	+	+	-	515,74
+	-	-	-	+	+	-	♣ 460,561
-	+	-	-	+	+	-	703,067
+	+	-	-	+	+	-	578,634
-	-	+	-	+	+	-	863,636
+	-	+	-	+	+	-	576,862
-	+	+	-	+	+	-	612,122
+	+	+	-	+	+	-	502,618
-	-	-	+	+	+	-	625,918
+	-	-	+	+	+	-	569,84
-	+	-	+	+	+	-	550,011
+	+	-	+	+	+	-	702,223
-	-	+	+	+	+	-	958,467
+	-	+	+	+	+	-	871,314
-	+	+	+	+	+	-	958,066
+	+	+	+	+	+	-	643,04
-	-	-	-	-	+	+	253,577
+	-	-	-	-	+	+	222,027
-	+	-	-	-	+	+	747,439
+	+	-	-	-	+	+	176,69
-	-	+	-	-	+	+	383,162
+	-	+	-	-	+	+	264,526
-	+	+	-	-	+	+	134,679
+	+	+	-	-	+	+	147,884
-	-	-	+	-	+	+	259,678
+	-	-	+	-	+	+	313,898
-	+	-	+	-	+	+	171,455
+	+	-	+	-	+	+	172,426
-	-	+	+	-	+	+	410,996
+	-	+	+	-	+	+	306,881
-	+	+	+	-	+	+	1115,21
+	+	+	+	-	+	+	174,302
-	-	-	-	+	+	+	191,925
+	-	-	-	+	+	+	273,825
-	+	-	-	+	+	+	176,747
+	+	-	-	+	+	+	168,905
-	-	+	-	+	+	+	186,134
+	-	+	-	+	+	+	311,081
-	+	+	-	+	+	+	179,803
+	+	+	-	+	+	+	♠ 134,721
-	-	-	+	+	+	+	363,336
+	-	-	+	+	+	+	296,197
-	+	-	+	+	+	+	704,94
+	+	-	+	+	+	+	176,358
-	-	+	+	+	+	+	222,81
+	-	+	+	+	+	+	276,723
-	+	+	+	+	+	+	181,782
+	+	+	+	+	+	+	137,757

Table 5.5: Comparison of various combinations of performance speed-up methods. “+” means enabled, “-” means disabled.

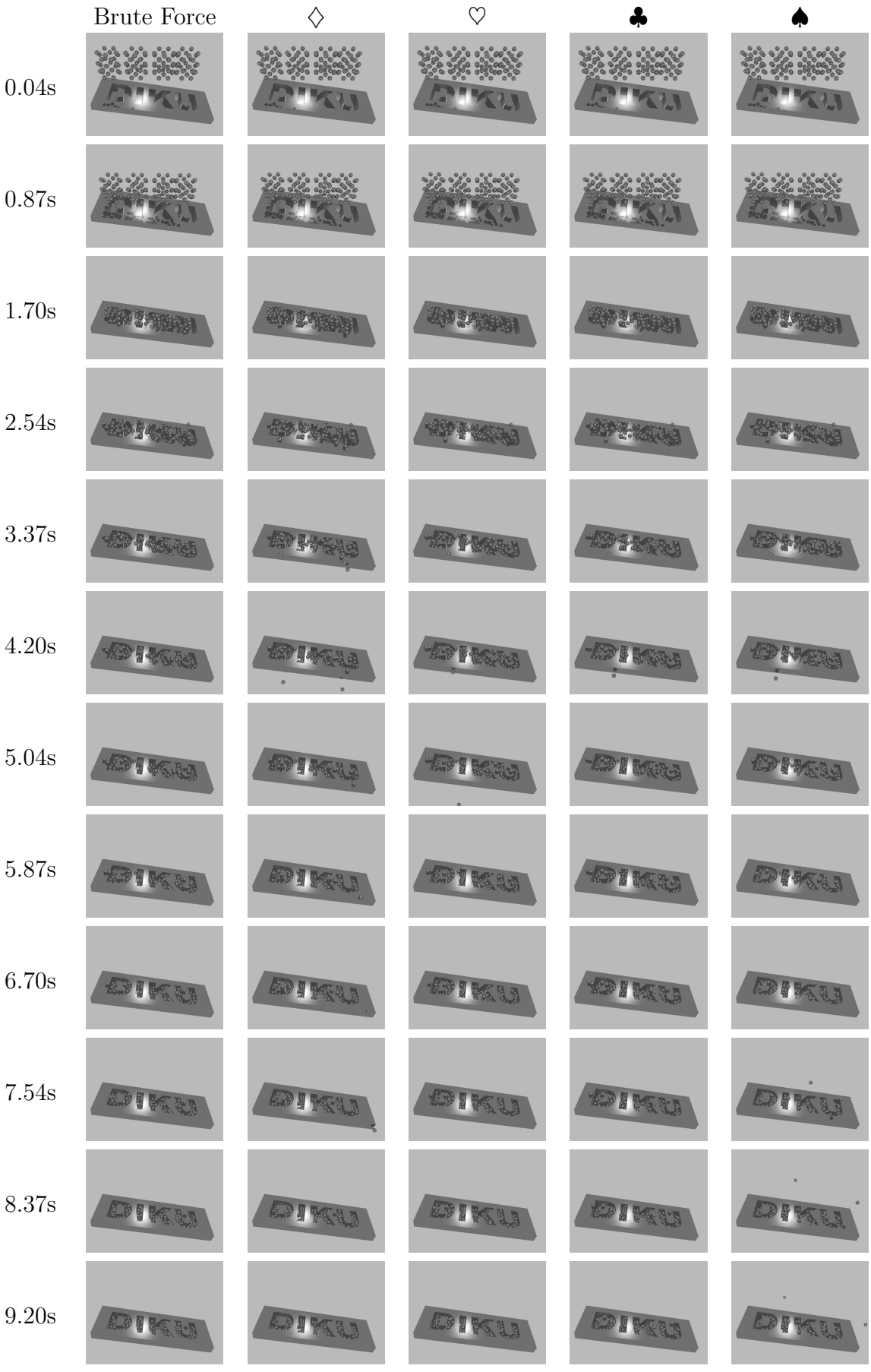


Figure 5.15: Motion results of selected combinations of speed up.

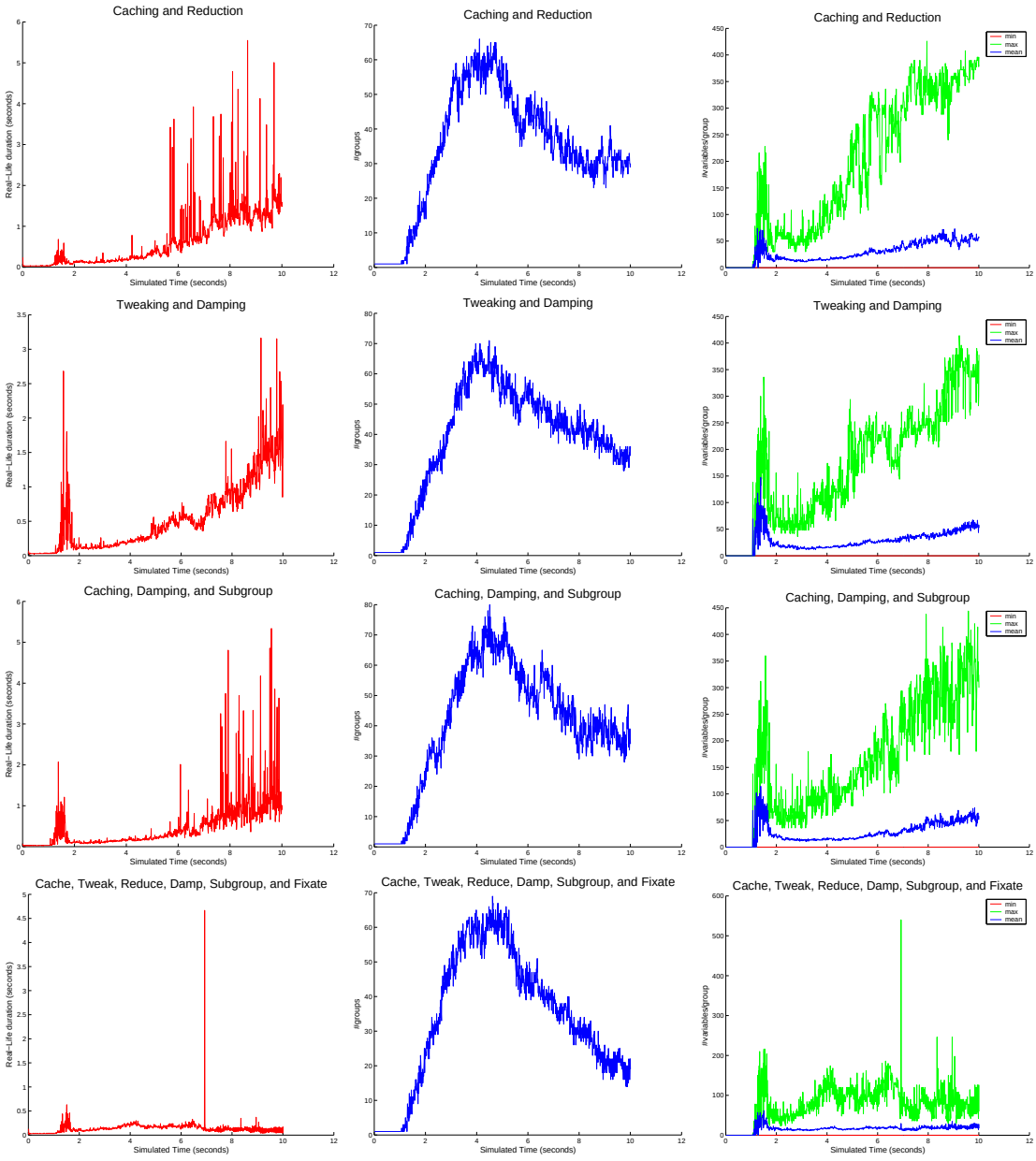


Figure 5.16: Performance measurements on the four selected combinations of speed ups.

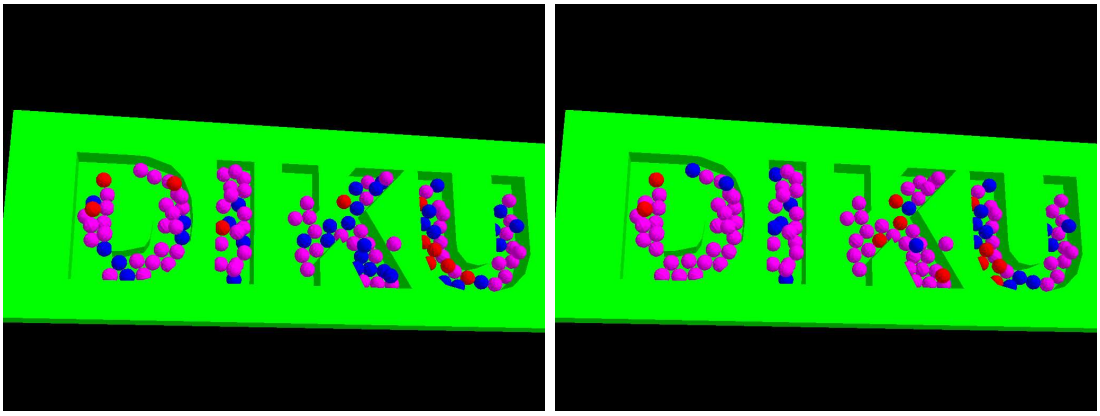


Figure 5.17: Figure showing sleepy object hanging in the air. Purple means sleepy, red moving, blue absolute rest, and green fixed. Frame grabs of simulation at time 9.8 secs and 9.87 secs using zeroing, damping, subgrouping and fixation.

Using more speed ups does not always imply better performance. In some cases, one speed up cancels the effect of another. For instance, using caching seems to make tweaking needless, since the cached solutions result in fast convergence. Only seldom is the upper iteration limit dictated by the "tweaking" reached.

Our experiments indicate that a promising avenue for high-performance simulations is a combination of subgrouping and fixation. However, from the resulting motions shown in Figure 5.15 it is also clear that this is far from trivial to device such a scheme.

We believe that contact graphs can also be exploited to determine when a paradigm switch should occur in a hybrid simulation as described in [99, 100]. In [99] it is suggested to track the contact state, i.e. the relative contact velocity, and use this information to determine when a contact should be solved by a constraint based method or by an impulse based method. Contact edges provide a perfect place for storing this tracking information, because they also provide one with the possibility of taking neighboring contacts into consideration. In [100], large stacks of objects are shown to be infeasible to simulate with an impulse based method. Examining the size of contact groups and their structure might give a clue to switch from an impulse based method to a constraint based method when lots of objects settle into resting contact upon each other.

Our numerical experiments clearly indicate that sleepy objects are a promising strategy. Therefore, it seems promising to look into better methods for more quickly making objects become sleepy and stay sleepy. For instance, to pre-process the complementarity formulation with a sequential collision method one can truncate impulses [35]. This was successfully applied to sequential collision resolving [70]. The novelty would be to extend the ideas to simultaneous contact resolving.

Chapter 6

Velocity Based Shock-Propagation

In this chapter we will present a time-stepping scheme combining a velocity based complementarity formulation with shock-propagation. Several techniques and methods are combined together to yield a stable, robust, versatile and high performance simulator, capable of dealing with dense stacking of objects.

In Section 6.1 we address the issue of implementing a fast iterative LCP solver for solving velocity based complementarity formulations, and convergence results are presented and discussed. Hereafter, in Section 6.2 we develop a re-sampling strategy for signed distance maps and a convenient sphere-tree for accelerating collision queries with signed distance maps. To overcome the performance cost of using signed-distance maps we analyze and modify a traditional box-box primitive contact generation algorithm in Section 6.3, showing that improvements in contact generation yield better quality of the simulation results. To overcome the disadvantages of simulation errors introduced by the use of iterative solvers, we extend complementarity based simulation with shock-propagation in Section 6.4. Finally, in Section 6.5 we present an improved approach for determining the sleepy state of objects together with a discussion of how to use a sleepy policy with the new simulator presented in this chapter.

An implementation of the theory presented in this chapter can be obtained from [113]. It is part of a multibody framework called "Retro".

6.1 Iterative Methods for solving LCPs in Multibody Dynamics

We will now look into numerically efficient algorithms for multibody dynamics. That is, efficient numerical methods for solving linear complementarity problems (LCPs). The theory presented is based on [109, 16, 21, 30, 112].

The name of the game is iterative methods. To be explained briefly, these have been around for a long time, and are often rejected because they converge very slowly towards an accurate, high precision solution. However, for animation purposes our major concern is not high precision. Therefore, we just need a method that in very few iterations produces a crude solution not too far from an accurate solution to our problem.

The concerned reader might fear that this will never work. The crude solution will obviously result in small simulation errors, such as small penetrations and/or too weak

contact forces. There can exist a cause for concern, however, Computer Animation or Computer Gaming requires fixed time-stepping algorithms to set a limit on the maximum time spent on computing a simulation update. These time-stepping schemes are often first order, meaning they will introduce errors of order $O(\Delta t)$, where Δt is the time-step size. Such errors are countered with stabilization [36] or projection methods [17] to reduce the errors. To recapitulate, the crude solution is not a problem in Computer Animation, since we need error correction anyway to counter numerical artifacts from the simulation method.

In short, an iterative method can be described as a method which iteratively improves a possible solution until an acceptable answer is found. In mathematical terms this can be stated as

$$\vec{x}^{\,k+1} = f(\vec{x}^{\,k}), \tag{6.1}$$

where the superscript k denotes the iteration. As it can be seen from the equation, a solution to the iterative method is actually a fix point for the function $f(\cdot)$,

$$\vec{x}^{\,*} = f(\vec{x}^{\,*}). \tag{6.2}$$

Here $\vec{x}^{\,*}$ is a solution which the sequence $\left\{\vec{x}^{\,0}, \vec{x}^{\,1}, \ldots, \vec{x}^{\,k}\right\}$ converges to in the limit as k goes to infinity, see Definition 6.1.2 for a definition of convergence.

Looking at equation (6.1) and equation (6.2), they appear to have very little in common with LCPs. We will begin our voyage towards formulating an LCP problem as a fix-point problem/iterative method, by examining iterative matrix solvers.

6.1.1 Iterative Matrix Solvers

Historically, iterative methods have been known to be very effective for solving large sparse matrix systems, such as

$$\mathbf{A}\vec{x} = \vec{b}, \tag{6.3}$$

where

$$\mathbf{A} = \begin{bmatrix} A_{0,0} & \cdots & A_{0,n-1} \\ \vdots & & \vdots \\ A_{n-1,0} & \cdots & A_{n-1,n-1} \end{bmatrix}. \tag{6.4}$$

At first glance, this matrix system appears to be nothing like equation (6.1) or equation (6.2). However, reworking the equation will reveal the basic idea. First we decompose $\mathbf{A}$ into a sum of three matrices, a strictly lower triangular matrix $\mathbf{L}$, a diagonal matrix $\mathbf{D}$, and a strictly upper triangular matrix $\mathbf{U}$:

$$A = \mathbf{L} + \mathbf{D} + \mathbf{U}. \tag{6.5}$$

Now the matrix equation (6.3) can be written

$$\begin{aligned} \mathbf{A}\vec{x} &= \vec{b} \\ (\mathbf{L} + \mathbf{D} + \mathbf{U})\,\vec{x} &= \vec{b} \\ \mathbf{D}\vec{x} &= \vec{b} - (\mathbf{L} + \mathbf{U})\,\vec{x} \\ \vec{x} &= \mathbf{D}^{-1}\vec{b} - \mathbf{D}^{-1}\,(\mathbf{L} + \mathbf{U})\,\vec{x}. \end{aligned} \tag{6.6}$$

Defining

$$\mathbf{T} = -\mathbf{D}^{-1}\left(\mathbf{L}+\mathbf{U}\right), \tag{6.7}$$

$$\vec{c} = \mathbf{D}^{-1}\vec{b}, \tag{6.8}$$

and

$$f(\vec{x}) = \mathbf{T}\vec{x} + \vec{c}. \tag{6.9}$$

From equation (6.6) we now have

$$\vec{x} = f(\vec{x}), \tag{6.10}$$

which is exactly the same as equation (6.2). All iterative methods which can be written in the form of equation (6.9) are called stationary iterative methods if neither the matrix $\mathbf{T}$ nor the vector $\vec{c}$ depend on the iteration count k. An example of a non-stationary iterative method is the Conjugate Gradient method [132]. Non-stationary iterative methods are characterized by computations that involve information which changes at each iteration. Implementing an iterative scheme using the mathematical formulation like equation (6.6) would be very inefficient for large matrix systems. To achieve a better implementation, we see that the i'th variable can be written as

$$\vec{x}_i = \frac{\left(\vec{b}_i - \sum_{j=0}^{i-1}\mathbf{L}_{i,j}\vec{x}_j - \sum_{j=i+1}^{n-1}\mathbf{U}_{i,j}\vec{x}_j\right)}{\mathbf{A}_{i,i}}. \tag{6.11}$$

Thus, the iterative scheme is

$$\vec{x}_i^{\,k+1} = \frac{\left(\vec{b}_i - \sum_{j=0}^{i-1}\mathbf{L}_{i,j}\vec{x}_j^{\,k} - \sum_{j=i+1}^{n-1}\mathbf{U}_{i,j}\vec{x}_j^{\,k}\right)}{\mathbf{A}_{i,i}}. \tag{6.12}$$

This scheme is called the Jacobi method, and a pseudo-code version is shown in Figure 6.1. From the pseudo-code it can be seen that the algorithm iterates indefinitely until a convergence test succeeds. This test is also called a stopping criteria. This is discussed in detail in Section 6.1.2.

Looking closely at equation (6.12), we see that when we update $\vec{x}_i^{\,k+1}$ we have already computed $\vec{x}_j^{\,k+1}$ for all $j < i$. It appears to be a better idea to use the most recent values in the computation of $\vec{x}_i^{\,k+1}$, in the hope that this aggressive approach yields a solution faster. Applying this idea equation (6.12) now becomes

$$\vec{x}_i^{\,k+1} = \frac{\left(\vec{b}_i - \sum_{j=0}^{i-1}\mathbf{L}_{i,j}\vec{x}_j^{\,k+1} - \sum_{j=i+1}^{n-1}\mathbf{U}_{i,j}\vec{x}_j^{\,k}\right)}{\mathbf{A}_{i,i}}. \tag{6.13}$$

This new iterative update scheme is called the Gauss-Seidel method.

From a computational viewpoint, the major difference between the Jacobi method and the Gauss-Seidel method is that in the Jacobi method all variables can be updated simultaneously, since each variable only depends on the solution from the previous iteration. This is not the case in the Gauss-Seidel method. Here, the computation is partly dependent on both the previous and the current solution. Thus, the Gauss-Seidel method is sequential in nature, whereas the Jacobi method is parallel.

```
algorithm Jacobi(A,x,b)
  x = initial guess
  for k=0,1,2,3,... do
    for i=0 to n-1 do
      delta = 0
      for j=0 to i-1 do
        delta += A(i,j)*x(j)
      next j
      for j=i+1 to n-1 do
        delta += A(i,j)*x(j)
      next j
      y(i) = (b(i) - delta)/A(i,i)
    next i
    x = y
    check convergence, continue if necessary
  next k
end algorithm
```

Figure 6.1: The Jacobi method.

Another point to be made here is that the Gauss-Seidel method is dependent on the order in which the variables are updated. This implies that if we compute the solution, $\vec{x}^{\,k}$, and then repeat the computation with a reordering, $\pi(\cdot)$, of the variables to obtain the solution , $\pi(\vec{x})^{\,k}$, then it is very likely that we find

$$\vec{x}^{\,k} \neq \pi(\vec{x})^{\,k}. \tag{6.14}$$

Thus, reordering can affect the rate of convergence of the Gauss-Seidel method. This fact can be exploited in order to try to find a reordering that will enhance convergence rate.

In Figure 6.2 a pseudo-code version of the Gauss-Seidel method is listed. To write the Gauss-Seidel method in matrix form, multiply both sides of equation (6.13) by $\mathbf{A}_{i,i}$ and collect all $k+1$'th terms on the left hand side,

$$\mathbf{A}_{i,i}\vec{x}_i^{\,k+1} + \sum_{j=0}^{i-1} \mathbf{L}_{i,j}\vec{x}_j^{\,k+1} = \vec{b}_i - \sum_{j=i+1}^{n-1} \mathbf{U}_{i,j}\vec{x}_j^{\,k}, \tag{6.15}$$

from which it follows that

$$\begin{aligned}(\mathbf{D}+\mathbf{L})\,\vec{x}^{\,k+1} &= \vec{b} - \mathbf{U}\vec{x}^{\,k} \\ \vec{x}^{\,k+1} &= (\mathbf{D}+\mathbf{L})^{-1}\left(\vec{b} - \mathbf{U}\vec{x}^{\,k}\right).\end{aligned} \tag{6.16}$$

Letting $\mathbf{T} = -(\mathbf{D}+\mathbf{L})^{-1}\,\mathbf{U}$ and $\vec{c} = (\mathbf{D}+\mathbf{L})^{-1}\,\vec{b}$ we can write the Gauss-Seidel method in the form

$$\vec{x}^{\,k+1} = \mathbf{T}\vec{x}^{\,k} + \vec{c}. \tag{6.17}$$

```
algorithm Gauss-Seidel(A,x,b)
  x = initial guess
  for k=0,1,2,3,... do
    for i=0 to n-1 do
      delta = 0
      for j=0 to i-1 do
        delta += A(i,j)*x(j)
      next j
      for j=i+1 to n-1 do
        delta += A(i,j)*x(j)
      next j
      x(i) = (b(i) - delta)/A(i,i)
    next i
    check convergence, continue if necessary
  next r
end algorithm
```

Figure 6.2: The Gauss-Seidel method.

From this it is evident that in order to study the convergence of both the Jacobi method and the Gauss-Seidel method, it is enough to study the iterative scheme given by the above equation. We refer the interested reader to [30] for details.

In general, the Gauss-Seidel method is considered superior to the Jacobi method. However, there are systems where one converges and the other does not, and vice versa.

For the next and last iterative matrix solver method we will present, we need the definition of a residual vector.

Definition 6.1.1 (The Residual Vector). Suppose $\tilde{x} \in \mathbb{R}^n$ is an approximation to the linear system $\mathbf{A}\vec{x} = \vec{b}$. The residual vector for $\tilde{x}$ with respect to this system is

$$\vec{r} = \mathbf{A}\tilde{x} - \vec{b}. \tag{6.18}$$

Let $\left\{\vec{r}_i^{\,k}\right\}$ denote the sequence of residual vectors for the Gauss-Seidel method, corresponding to the sequence of solutions $\left\{\vec{x}_i^{\,k}\right\}$. The subscript might appear confusing, but it refers to currently known solution in the k'th iteration, just before updating the i'th variable. By definition we see that the m'th coordinate of $\vec{r}_i^{\,k+1}$ is given by

$$\begin{aligned}
\vec{r}_{mi}^{\,k+1} &= \vec{b}_m - \sum_{j=0}^{i-1} \mathbf{A}_{m,j}\vec{x}_j^{\,k+1} - \sum_{j=i+1}^{n-1} \mathbf{A}_{m,j}\vec{x}_j^{\,k} - \mathbf{A}_{m,i}\vec{x}_m^{\,k} \\
&= \vec{b}_m - \sum_{j=0}^{i-1} \mathbf{A}_{m,j}\vec{x}_j^{\,k+1} - \sum_{j=i}^{n-1} \mathbf{A}_{m,j}\vec{x}_j^{\,k}.
\end{aligned} \tag{6.19}$$

If we look at the i'th coordinate we have

$$\vec{r}_{ii}^{\,k+1} = \vec{b}_i - \sum_{j=0}^{i-1} \mathbf{A}_{i,j}\vec{x}_j^{\,k+1} - \sum_{j=i+1}^{n-1} \mathbf{A}_{i,j}\vec{x}_j^{\,k} - \mathbf{A}_{i,i}\vec{x}_i^{\,k} \tag{6.20}$$

$$\mathbf{A}_{i,i}\vec{x}_i^{\,k} + \vec{r}_{ii}^{\,k+1} = \vec{b}_i - \sum_{j=0}^{i-1} \mathbf{A}_{i,j}\vec{x}_j^{\,k+1} - \sum_{j=i+1}^{n-1} \mathbf{A}_{i,j}\vec{x}_j^{\,k}.$$

Looking at the left hand side and comparing with equation (6.13), we see that it is equal to $A_{i,i}\vec{x}_i^{\,k+1}$, so we have

$$A_{i,i}\vec{x}_i^{\,k} + \vec{r}_{ii}^{\,k+1} = A_{i,i}\vec{x}_i^{\,k+1}, \tag{6.21}$$

or if we rearrange terms we get

$$\vec{x}_i^{\,k+1} = \vec{x}_i^{\,k} + \frac{\vec{r}_{ii}^{\,k+1}}{A_{i,i}}. \tag{6.22}$$

Up to now we have only rewritten the update step for the i'th variable of the Gauss-Seidel method in terms of the residual vector. We can gain some more insight into the Gauss-Seidel method by repeating the above steps for the $i+1$'th variable. We have

$$\vec{r}_{i,i+1}^{\,k+1} = \vec{b}_i - \sum_{j=0}^{i} \mathbf{A}_{i,j}\vec{x}_j^{\,k+1} - \sum_{j=i+1}^{n-1} \mathbf{A}_{i,j}\vec{x}_j^{\,k} \tag{6.23}$$

$$= \underbrace{\vec{b}_i - \sum_{j=0}^{i-1} \mathbf{A}_{i,j}\vec{x}_j^{\,k+1} - \sum_{j=i+1}^{n-1} \mathbf{A}_{i,j}\vec{x}_j^{\,k}}_{\mathbf{A}_{i,i}\vec{x}_i^{\,k+1}} - \mathbf{A}_{i,i}\vec{x}_i^{\,k+1} \tag{6.24}$$

$$= 0. \tag{6.25}$$

The last step follows from equation (6.13). Thus, the Gauss-Seidel method is characterized by choosing $\vec{x}_i^{\,k+1}$ in such a way that the i'th component of $\vec{r}_{i+1}^{\,k+1}$ is zero. Reducing one coordinate of the residual vector to zero is not generally the most efficient way to reduce the overall size of the vector $\vec{r}_{i+1}^{\,k+1}$. We need to choose $\vec{x}_i^{\,k+1}$ to make $||\vec{r}_{i+1}^{\,k+1}||$ small. Let us modify equation (6.22) to

$$\vec{x}_i^{\,k+1} = \vec{x}_i^{\,k} + \omega\frac{\vec{r}_{ii}^{\,k+1}}{A_{i,i}}, \tag{6.26}$$

where $\omega > 0$. For certain values of ω the norm of the residual vector is reduced and this leads to faster convergence. The ω parameter is a relaxation of the Gauss-Seidel method. The new iterative scheme is called successive over relaxation (SOR). Whenever $0 < \omega < 1$ we call it under-relaxation, and when $1 < \omega$ we call it over-relaxation. Notice that when $\omega = 1$ the method is simply the Gauss-Seidel method.

If we insert equation (6.20) into equation (6.26) we can reformulate the SOR method into a more implementation-friendly formula

$$\vec{x}_i^{\,k+1} = \vec{x}_i^{\,k} + \omega\frac{\vec{b}_i - \sum_{j=0}^{i-1} \mathbf{A}_{i,j}\vec{x}_j^{\,k+1} - \sum_{j=i+1}^{n-1} \mathbf{A}_{i,j}\vec{x}_j^{\,k} - \mathbf{A}_{i,i}\vec{x}_i^{\,k}}{A_{i,i}}$$

$$= (1-\omega)\,\vec{x}_i^{\,k} + \frac{\omega}{A_{i,i}}\left(\vec{b}_i - \sum_{j=0}^{i-1} \mathbf{A}_{i,j}\vec{x}_j^{\,k+1} - \sum_{j=i+1}^{n-1} \mathbf{A}_{i,j}\vec{x}_j^{\,k}\right). \tag{6.27}$$

```
algorithm SOR(A,x,b)
  x = initial guess
  for k=0,1,2,3,... do
    for i=0 to n-1 do
      delta = 0
      for j=0 to i-1 do
        delta += A(i,j)*x(j)
      next j
      for j=i+1 to n-1 do
        delta += A(i,j)*x(j)
      next j
      delta = (b(i) - delta)/A(i,i)
      x(i) = x(i) + w(delta - x(i))
    next i
    check convergence, continue if necessary
  next k
end algorithm
```

Figure 6.3: The SOR method.

The pseudo-code of the SOR method can be seen in Figure 6.3. To determine the matrix-form of the SOR method, we rewrite equation (6.27) as

$$A_{i,i}\vec{x}_i^{\,k+1} + \omega \sum_{j=0}^{i-1} \mathbf{A}_{i,j}\vec{x}_j^{\,k+1} = (1-\omega)\, A_{i,i}\vec{x}_i^{\,k} + \omega\vec{b}_i - \omega \sum_{j=i+1}^{n-1} \mathbf{A}_{i,j}\vec{x}_j^{\,k}. \tag{6.28}$$

From which we see that

$$(\mathbf{D} + \omega\mathbf{L})\,\vec{x}^{\,k+1} = ((1-\omega)\,\mathbf{D} - \omega\mathbf{U})\,\vec{x}^{\,k} + \omega\vec{b}, \tag{6.29}$$

or

$$\vec{x}^{\,k+1} = (\mathbf{D} + \omega\mathbf{L})^{-1}\,((1-\omega)\,\mathbf{D} - \omega\mathbf{U})\,\vec{x}^{\,k} + \omega\,(\mathbf{D} + \omega\mathbf{L})^{-1}\,\vec{b}. \tag{6.30}$$

If we let $\mathbf{T} = (\mathbf{D} + \omega\mathbf{L})^{-1}\,((1-\omega)\,\mathbf{D} - \omega\mathbf{U})$ and $\vec{c} = \omega\,(\mathbf{D} + \omega\mathbf{L})^{-1}\,\vec{b}$ then we see that the SOR method can be written in the form

$$\vec{x}^{\,k+1} = \mathbf{T}\vec{x}^{\,k} + \vec{c}. \tag{6.31}$$

In short, the SOR method has the same form as the Jacobi and the Gauss-Seidel methods.

To summarize, in this section we have derived three iterative matrix solver methods built on top of each other. The purpose of this is twofold. Firstly, we want to familiarize the reader with some basic matrix concepts and elementary methods, which will show up again later. Secondly, we want to give the reader an impression of how these kinds of matrix problems relate to the well-known fix-point problem.

We end this section by listing the matrix-forms of the three iterative matrix solver methods in Table 6.1.

$$\vec{x}^{\,k+1} = \mathbf{D}^{-1}\left(\vec{b} - (\mathbf{L}+\mathbf{U})\,\vec{x}^{\,k}\right)$$
$$\vec{x}^{\,k+1} = \mathbf{D}^{-1}\left(\vec{b} - \mathbf{L}\vec{x}^{\,k+1} - \mathbf{U}\vec{x}^{\,k}\right)$$
$$\vec{x}^{\,k+1} = \vec{x}^{\,k} + \omega\left(\mathbf{D}^{-1}\left(\vec{b} - \mathbf{L}\vec{x}^{\,k+1} - \mathbf{U}\vec{x}^{\,k}\right) - \vec{x}^{\,k}\right)$$

Table 6.1: Matrix forms of iterative matrix solver methods. Top is Jacobi method, middle is Gauss-Seidel method, bottom is SOR method.

6.1.2 Convergence Testing and Stopping Criteria

A convergence test is a test for how close the vector $\vec{x}^{\,r+1}$ is to a fix point (see equation (6.2)). To formalize this we need a definition of convergence.

Definition 6.1.2 (Convergence). An infinite sequence $\left\{\vec{x}^{\,k}\right\}_{k=0}^{\infty} = \{\vec{x}^{\,0}, \vec{x}^{\,1}, \vec{x}^{\,2}, \ldots\}$ of vectors in $\mathbb{R}^n$ is said to converge to $\vec{x}^{\,*}$ wrt. the vector norm $||\cdot||$ if and only if, for any $\varepsilon > 0$, there exists an integer $N(\varepsilon)$ such that

$$||\vec{x}^{\,k} - \vec{x}^{\,*}|| < \varepsilon \qquad \text{for all} \quad k \geq N(\varepsilon). \tag{6.32}$$

The vector $\vec{x}^{\,*}$ is called the limit point or accumulation point.

As is seen from Definition 6.1.2 the concept of a vector norm is vital in order to define convergence.

Definition 6.1.3 (Vector Norm). A vector norm on $\mathbb{R}^n$ is a function $||\cdot||$, from $\mathbb{R}^n$ into $\mathbb{R}$ with the following properties:

(i) $||\vec{x}|| > 0$ for all $\vec{x} \in \mathbb{R}^n$,

(ii) $||\vec{x}|| = 0$ if and only if $\vec{x} = \vec{0}$,

(iii) $||\alpha\vec{x}|| = |\alpha|||\vec{x}||$ for all $\vec{x} \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$,

(iv) $||\vec{x} + \vec{y}|| \leq ||\vec{x}|| + ||\vec{y}||$ for all $\vec{x}, \vec{y} \in \mathbb{R}^n$.

In theory, any vector norm can be used. However, we will only consider two specific vector norms.

Definition 6.1.4 (The l_2 and l_∞ norms). The l_2 and l_∞ norms for the vector $\vec{x} = [x_0, x_1, \ldots, x_{n-1}]^T$ are defined by

$$||\vec{x}||_2 = \sqrt{\left(\sum_{i=0}^{n-1} x_i^2\right)} \qquad \text{and} \qquad ||\vec{x}||_\infty = \max_{0 \leq i \leq n-1} |x_i|. \tag{6.33}$$

The l_2 norm is called the Euclidean norm of the vector $\vec{x}$. For a vector $\vec{x} \in \mathbb{R}^n$ the norms are related by

$$||\vec{x}||_\infty \leq ||\vec{x}||_2 \leq \sqrt{n}||\vec{x}||_\infty. \tag{6.34}$$

Often, the convergence test does not consist of a single stopping criteria, and several are used in combination. An absolute estimate for the error can be taken by measuring the difference between the solutions of two succeeding iterations, $\vec{x}^{\,k+1}$ and $\vec{x}^{\,k}$. Only if the distance between these are below a user specified threshold, ε, the sequence is considered to converge. That is, if

$$||\vec{x}^{\,k+1} - \vec{x}^{\,k}|| < \varepsilon, \tag{6.35}$$

then the iterative method converges. By comparison with Definition 6.1.2 it is evident that the test will succeed in case the iterative method converges. However, the test can be fooled if convergence is too slow. Even worse, there exist sequences with the property that the difference $\vec{x}^{\,k+1} - \vec{x}^{\,k}$ converges to zero while the sequence itself diverges, as the following example illustrates.

Example 6.1.5. Let the sequence $\left\{x^{\,k}\right\}_{k=1}^{\infty}$ of real numbers be defined by

$$x^{\,k} = \sum_{r=1}^{k} \frac{1}{r}. \tag{6.36}$$

Now, the limit of the difference is

$$\lim_{k\to\infty} \left(x^{\,k} - x^{\,k-1}\right) = \lim_{k\to\infty} \left(\sum_{r=1}^{k} \frac{1}{r} - \sum_{r=1}^{k-1} \frac{1}{r}\right) = \lim_{k\to\infty} \left(\frac{1}{k}\right) = 0. \tag{6.37}$$

However,

$$\lim_{k\to\infty} x^{\,k} = \lim_{k\to\infty} \left(\sum_{r=1}^{k} \frac{1}{r}\right) = \infty. \tag{6.38}$$

Meaning that the sequence is divergent.

Thus, to overcome the difficulties with the absolute error measure, a measure of relative error is sometimes applied instead,

$$\frac{||\vec{x}^{\,k+1} - \vec{x}^{\,k}||}{||\vec{x}^{\,k+1}||} < \varepsilon. \tag{6.39}$$

Observe that the $k+1$'th solution must be non-zero in order for the relative test to work. As mentioned before, any convenient norm can be used. The usual being the l_∞ norm, since it is cheap to compute. Also, it is much more robust towards numerical deficiencies, such as overflow and underflow, which could occur with the l_2 norm. Let us formalize our formulas into the following definitions.

Definition 6.1.6 (The Absolute Convergence Test). Let the solutions of two succeeding iterations be $\vec{x}^{\,k+1}$ and $\vec{x}^{\,k}$, and given $\varepsilon > 0$. Then, if

$$||\vec{x}^{\,k+1} - \vec{x}^{\,k}|| < \varepsilon, \tag{6.40}$$

the sequence is considered to converge.

Definition 6.1.7 (The Relative Convergence Test). Let the solutions of two succeeding iterations be $\vec{x}^{\,k+1}$ and $\vec{x}^{\,k}$, where $\vec{x}^{\,k+1} \neq \vec{0}$, and given $\varepsilon > 0$. Then if

$$\frac{||\vec{x}^{\,k+1} - \vec{x}^{\,k}||}{||\vec{x}^{\,k+1}||} < \varepsilon, \tag{6.41}$$

the sequence is considered to converge.

It is often a good idea to specify a maximum limit on the number of iterations that one is willing to use in an iterative method. There are mainly two good reasons for this. In a time critical computation one wants to control how much time is spent on a computation. A maximum limit can help enforce this. In case the problem is unsolvable, i.e. it diverges, an iterative method would loop for ever. A maximum limit helps avoid infinite loops.

For sequences of real numbers, we can define the rate of convergence as follows:

Definition 6.1.8 (Rate of Convergence). Assume the sequence $\left\{x^{\,k}\right\}_{k=0}^{\infty}$ with $x^{\,k} \in \mathbb{R}$ converges to α, and the sequence $\left\{\beta^{\,k}\right\}_{k=0}^{\infty}$ with $\beta^{\,k} \in \mathbb{R}$ converges to zero. If a positive constant K exists with

$$|x^{\,k} - \alpha| \leq K|\beta^{\,k}| \qquad \text{for large} \quad k. \tag{6.42}$$

then we say that $x^{\,k}$ converges to α with rate of convergence $O(\beta^{\,k})$. This can be written

$$x^{\,k} = \alpha + O(\beta^{\,k}). \tag{6.43}$$

In many cases the sequence $\left\{\beta^{\,k}\right\}$ has the form

$$\beta^{\,k} = \frac{1}{k^p}, \tag{6.44}$$

for some number $p > 0$. We consider the largest value of p such that $x^{\,k} = \alpha + O(1/k^p)$ to describe the rate at which $x^{\,k}$ converges to α.

We have given this formal presentation of rate of convergence, because iterative methods might reorder the variables such that variables with slowest rate of convergence are updated before variables with faster rate of convergence. In practice Definition 6.1.8 is not used. Instead the magnitudes of absolute error of the individual variables are used as an ad hoc measure of rate of convergence. That is, if

$$|\vec{x}_i^{\,k+1} - \vec{x}_i^{\,k}| < |\vec{x}_j^{\,k+1} - \vec{x}_j^{\,k}|, \tag{6.45}$$

then the i'th variable is considered to converge faster than the j'th variable. This ad-hoc way of measuring rate of convergence allows a simple approach in practice, where variables are reordered in decreasing order wrt. the absolute error.

6.1.3 Iterative Methods For Solving LCPs

Let us briefly restate the linear complementarity problem (LCP) we want to solve,

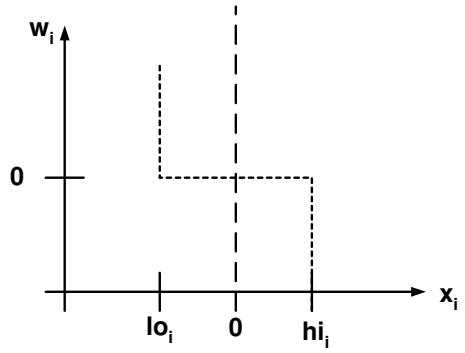


Figure 6.4: Illustration of the complementarity condition on the i'th variable.

Definition 6.1.9 (The Linear Complementarity Problem). Given a symmetric matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, a vector $\vec{b} \in \mathbb{R}^n$, a vector of lower limits, $\overline{lo} \leq 0$, and upper limits $\overline{hi} \geq 0$, where $\overline{lo}, \overline{hi} \in \mathbb{R}^n$. Find $\vec{x} \in \mathbb{R}^n$ and $\vec{w} \in \mathbb{R}^n$ such that

$$\vec{w} = \mathbf{A}\vec{x} - \vec{b}, \tag{6.46a}$$

$$\overline{lo} \leq \vec{x} \leq \overline{hi}, \tag{6.46b}$$

and for all $i = 0, \ldots, n-1$, one of the three conditions below holds

$$\vec{x}_i = \overline{lo}_i, \vec{w}_i \geq 0, \tag{6.47a}$$

$$\vec{x}_i = \overline{hi}_i, \vec{w}_i \leq 0, \tag{6.47b}$$

$$\overline{lo}_i < \vec{x}_i < \overline{hi}_i, \vec{w}_i = 0. \tag{6.47c}$$

Figure 6.4 helps illustrate the complementarity conditions in equation (6.47). Notice that as long as $\vec{x}_i$ is within its lower and upper bounds, $\vec{w}_i$ is forced to zero. Only at the lower and upper bounds is $\vec{w}_i$ non-zero. Usually, the general LCP is formulated with $\overline{lo} = \vec{0}$ and $\overline{hi} = \vec{\infty}$, in which case the above definition reduces to

$$\vec{w} = \mathbf{A}\vec{x} - \vec{b} \geq 0, \tag{6.48a}$$

$$\vec{x} \geq \vec{0}, \tag{6.48b}$$

$$\vec{x}^T \vec{w}. \tag{6.48c}$$

In multibody dynamics problems the $\mathbf{A}$-matrix is often symmetric, unless a linearized friction cone is used, where auxiliary variables are used to couple the normal force to the tangential force at a contact point. The $\mathbf{A}$-matrix is often PSD or sometimes PD. Even if it is PSD, tricks such as constraint force mixing can be applied to make it PD. To make a long story short, the $\mathbf{A}$-matrix can be made numerically more pleasant, such that we know a solution exists to the LCP problem.

Theorem 6.1.10 (Unified Matrix Notation). All iterative matrix solvers presented in Section 6.1.1, can be written in the iterative matrix-form,

$$\vec{x}^{\,k+1} = \lambda \left(\vec{x}^{\,k} - \omega \mathbf{E}^k \left(\mathbf{A}\vec{x}^{\,k} - \vec{b} + \mathbf{K}^k \left(\vec{x}^{\,k+1} - \vec{x}^{\,k} \right) \right) \right) + (1 - \lambda)\, \vec{x}^{\,k}, \tag{6.49}$$

where $\mathbf{E}^k$ is a diagonal matrix with positive diagonal elements. $\mathbf{K}^k$ is either a strictly lower or strictly upper matrix, λ and ω are parameters satisfying $0 < \lambda \leq 1$, $\omega > 0$.

Proof. In all cases pick $\lambda = 1$. Thus we have the more simple matrix form

$$\vec{x}^{\,k+1} = \left(\vec{x}^{\,k} - \omega \mathbf{E}^k \left(\mathbf{A}\vec{x}^{\,k} - \vec{b} + \mathbf{K}^k \left(\vec{x}^{\,k+1} - \vec{x}^{\,k}\right)\right)\right). \tag{6.50}$$

In case of the Jacobi method, choose $\mathbf{E}^k = D^{-1}$, $\omega = 1$, and $\mathbf{K}^k = \mathbf{0}$, and we have,

$$\vec{x}^{\,k+1} = \left(\vec{x}^{\,k} - \mathbf{D}^{-1}\left(\mathbf{A}\vec{x}^{\,k} - \vec{b}\right)\right). \tag{6.51}$$

Now exploit that $\mathbf{A} = D + L + U$

$$\begin{aligned}
\vec{x}^{\,k+1} &= \left(\vec{x}^{\,k} - \mathbf{D}^{-1}\left((\mathbf{D}+\mathbf{L}+\mathbf{U})\,\vec{x}^{\,k} - \vec{b}\right)\right) \\
&= \left(\vec{x}^{\,k} - \mathbf{D}^{-1}\left(\mathbf{D}\vec{x}^{\,k} + (\mathbf{L}+\mathbf{U})\,\vec{x}^{\,k} - \vec{b}\right)\right) \\
&= \left(\vec{x}^{\,k} - \mathbf{D}^{-1}\mathbf{D}\vec{x}^{k} + \mathbf{D}^{-1}\left((\mathbf{L}+\mathbf{U})\,\vec{x}^{\,k} - \vec{b}\right)\right) \\
&= \mathbf{D}^{-1}\left(\vec{b} - (\mathbf{L}+\mathbf{U})\,\vec{x}^{\,k}\right).
\end{aligned} \tag{6.52}$$

By comparison of the first equation in Table 6.1 we conclude that the Jacobi method can be written in the form of equation (6.49). For the Gauss-Seidel method we pick: $\lambda, \omega = 1$, $\mathbf{K}^k = \mathbf{L}$, and $\mathbf{E}^k = \mathbf{D}^{-1}$. Substituting into equation (6.49) yields

$$\begin{aligned}
\vec{x}^{\,k+1} &= \left(\vec{x}^{\,k} - \mathbf{D}^{-1}\left((\mathbf{D}+\mathbf{L}+\mathbf{U})\,\vec{x}^{\,k} - \vec{b} + \mathbf{L}\left(\vec{x}^{\,k+1} - \vec{x}^{\,k}\right)\right)\right) \\
&= \left(\vec{x}^{\,k} - \mathbf{D}^{-1}\left(\mathbf{D}\vec{x}^{\,k} + (\mathbf{L}+\mathbf{U})\,\vec{x}^{\,k} - \vec{b} + \mathbf{L}\vec{x}^{\,k+1} - \mathbf{L}\vec{x}^{\,k}\right)\right) \\
&= \mathbf{D}^{-1}\left(\vec{b} - \mathbf{U}\vec{x}^{\,k} - \mathbf{L}\vec{x}^{\,k+1}\right).
\end{aligned} \tag{6.53}$$

By comparison with the second equation in Table 6.1, we conclude that the Gauss-Seidel method can be written in the form of equation (6.49). The SOR method follows from the last derivation if ω is left unspecified. □

Definition 6.1.11 (Clamping). Given a vector $\vec{x} \in \mathbb{R}^n$, a vector of lower limits, $\overline{lo} \leq 0$, and upper limits $\overline{hi} \geq 0$, where $\overline{lo}, \overline{hi} \in \mathbb{R}^n$, the clamping operation on $\vec{x}$ is written $(\vec{x})^{\,+}$, and means, for each $j = 0$ to $n-1$

$$\vec{x}_j{}^{+} = \max\left(\min\left(\vec{x}_j, \overline{hi}_j\right), \overline{lo}_j\right). \tag{6.54}$$

As it can be seen from Definition 6.1.11 it works element-wise. If a coordinate exceeds a lower or upper bound it is projected back onto the violated limit.

We can now write the general iterative LCP scheme.

Definition 6.1.12 (The General Iterative LCP Method). Given an iterative scheme as in Theorem 6.1.10, an iterative solver to the LCP problem in Definition 6.1.9 is given by

$$\vec{x}^{\,k+1} = \lambda\left(\vec{x}^{\,k} - \omega\mathbf{E}^k\left(\mathbf{A}\vec{x}^{\,k} - \vec{b} + \mathbf{K}^k\left(\vec{x}^{\,k+1} - \vec{x}^{\,k}\right)\right)\right)^{+} + (1-\lambda)\,\vec{x}^{\,k}. \tag{6.55}$$

According to [109] the only requirement is that $\mathbf{A}$ is symmetric. If so, it can be proven that if the sequence of solutions converges to a limit point, that point will be a solution of the LCP (see Theorem 9.9 pp. 369 in [109]). Furthermore, it can be shown that if $\mathbf{A}$ is also PD then the sequence will converge to a solution of the LCP (see Corollary 9.3, pp. 372 in [109]). This is often enough for our purpose. We refer the interested reader to [109] for detailed proofs regarding the general iterative scheme.

However, in our specific case where all iterative matrix solver methods take the form of Theorem 6.1.10, we can use the simpler form.

Corollary 6.1.13 (The Specific Iterative LCP Scheme). Given an iterative scheme, as in Table 6.1, an iterative solver to the LCP problem in Definition 6.1.9 is given by

$$\vec{x}^{\,k+1} = \left(\vec{x}^{\,k} - \omega \mathbf{D}^{-1}\left(\mathbf{A}\vec{x}^{\,k} - \vec{b} + \mathbf{K}\left(\vec{x}^{\,k+1} - \vec{x}^{\,k}\right)\right)\right)^{+}, \tag{6.56}$$

where $\mathbf{K}$ is either zero or $\mathbf{L}$.

It is worth noticing that the iterative scheme used in [112] can be written in matrix form as

$$\vec{x}^{\,k+1} = \left(\vec{x}^{\,k} + \omega \mathbf{D}^{-1}\left(\vec{b} - (\mathbf{U}+\mathbf{D})\,\vec{x}^{\,k} - \mathbf{L}\vec{x}^{\,k+1} - \mathbf{E}_{\mathrm{cfm}}\vec{x}^{\,k}\right)\right)^{+} \tag{6.57}$$

This is a different matrix form than equation (6.49). Thus solution existence and convergence proofs do not follow from the theorems in [109].

6.1.4 Implementation of an iterative LCP solver

In this section we will outline an efficient and practical implementation of an iterative LCP solver. We will use the SOR method as our working example, since it is the most complex of the iterative methods we have presented.

Using the SOR-method with a given upper iteration bound $k_{\max}$, a straightforward implementation of the iterative LCP scheme would look like the pseudo-code sketched in Figure 6.5. As it can be seen, all we have done is to clamp the i'th variable after it has been updated by a classical SOR method. Next, we can try to permute the variables before each iteration, Figure 6.6 illustrates where the permutation should be done. Permutation could be done at random. For instance, whenever a certain number of interleaved iterations have been performed. Or, variables could be reordered according to their rate of convergence. As an example, in [112] a random permutation is used by default, but only every 7'th iteration. In practice, permutation is done much more efficiently by using an index array, and only entries in this index-array need be swapped instead of reordering the actual $\mathbf{A}$-matrix, $\vec{b}$-vector et cetera.

Next, we can make the limits of the variables depend upon the value of some other variable, like friction forces are depend on normal forces. This idea was introduced to the Graphics Community by [16] and is used in [112]. Before updating an variable, its upper and lower limits are re-evaluated. Thus, we need to make sure that during permutation the independent variables come before the dependent variables i.e. normal force variables comes before friction variables. To keep track of the dependency, the index-array can be extended with an entry keeping information about any dependency. Figure 6.7 illustrates how the extensions should be applied. Yet another trick would be to warm-start the

```
Algorithm SOR-LCP-1(A,x,b,hi,lo,w,k_max)
  set x to initial guess
  for k = 1 to k_max do
    for i=0 to n-1 do
      delta = 0
      for j=0 to i-1 do
        delta += A(i,j)*x(j)
      next j
      for j=i+1 to n-1 do
        delta += A(i,j)*x(j)
      next j
      delta = (b(i) - delta)/A(i,i)
      x(i) = x(i) + w*(delta - x(i))
      if(x(i)>hi(i))   x(i) = hi(i)
      if(x(i)<lo(i))   x(i) = lo(i)
    next i
  next k
End algorithm
```

Figure 6.5: LCP SOR method, with upper iteration bound.

```
Algorithm SOR-LCP-2(A,x,b,hi,lo,w,k_max)
  set x to initial guess
  for k = 1 to k_max do
    permute(A,x,b,hi,lo)
    for i=0 to n-1 do
      ...same as SOR-LCP-1...
    next i
  next k
End algorithm
```

Figure 6.6: SOR LCP method with permutation of variables and fixed iteration limit.

iterative solver by feeding it the solution of the previous invocation as the initial solution guess. This is done in the hope that the previous solution is close to the current solution. Imagine a stack of resting boxes. If they are left untouched then it is unlikely that the contact forces between the boxes would change over time, indicating that the LCP solver would find the same solution whenever it is invoked. In [112] warm-starting is supplied by a down scaling of the last solution. This is performed to avoid jerkiness in motor driven joints, that is

$$x = \frac{9}{10} x_{\text{last}}. \tag{6.58}$$

However, it should be noted that this has more to do with the specific "dynamics" model used in [112] than with solving an LCP. Having stored the solution from the last iteration, it is possible to estimate the absolute or relative error for each variable. These estimates

```
Algorithm SOR-LCP-3(A,x,b,hi,lo,w,k_max,idx_array)
  set x to initial guess
  for k = 1 to k_max do
    permute(idx_array)
    for i=0 to n-1 do
      idx = index_array(i).idx
      dep_idx = index_array(i).dependency
      if dep_idx then
        hi(idx) = new_hi(x(dep_idx))
        lo(idx) = new_lo(x(dep_idx))
      end if
      delta = 0
      for j=0 to idx-1 do
        delta += A(idx,j)*x(j)
      next j
      for j=idx+1 to n-1 do
        delta += A(idx,j)*x(j)
      next j
      delta = (b(idx) - delta)/A(idx,idx)
      x(idx) = x(idx) + w*(delta - x(idx))
      if(x(idx)>hi(idx))   x(idx) = hi(idx)
      if(x(idx)<lo(idx))   x(idx) = lo(idx)
    next i
  next k
End algorithm
```

Figure 6.7: Linear dependent limits and index array.

can be used to guide the permutation order of the variables, but it can also be used to yield an early exit. In case a solution is found, this is shown in Figure 6.8. Observe that early exit is only allowed after a certain minimum iteration count, $k_{\min}$, where $1 \leq k_{\min} < k_{\max}$. This is so one can force the LCP solver to improve upon the solution in case warm starting is used.

In Figure 6.8 the absolute error measure is used. This is sensible if one wants to use the error measure as an indication of the rate of convergence as described in Section 6.1.2. If this is not needed, a relative measure could be used.

6.1.5 Optimization by Precomputation

Finally, we will explain a more optimal way to compute the update of the $\vec{x}_i$ variable specific to the "typical" matrix products in multibody dynamics. To recap the typical

```
Algorithm SOR-LCP-4(A,x,b,hi,lo,w,k_min,k_max,idx_array,last_x,threshold)
  set x to last_x
  for k = 1 to k_max do
    if k>k_min then
      error = |x-last_x|
      if  max(error)<threshold then
        exit
    end if
    permute(idx_array,error)
    last_x = x
    for i=0 to n-1 do
      ...same as SOR-LCP-3...
    next i
  next k
End algorithm
```

Figure 6.8: Warm-starting with early exiting.

velocity based complementarity formulation, it can be stated as

$$\mathbf{A} = \mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T + \mathbf{K}_{\text{cfm}}\Delta t \tag{6.59a}$$

$$\vec{b} = \mathbf{J}\left(\vec{u} + \Delta t\mathbf{M}^{-1}\vec{f}_{\text{ext}}\right) - \vec{b}_{\text{error}} \tag{6.59b}$$

$$\vec{f}_{\text{ext},i}^T = [m_i\vec{g}^T, (\vec{\omega}_i \times \mathbf{I}_i\vec{\omega}_i)^T]^T \tag{6.59c}$$

$$\vec{b}_{\text{error}} = \ldots \text{constraint stabilization} \ldots \tag{6.59d}$$

Solving it using SOR-LCP yields

$$\vec{w} = \mathbf{A}\vec{x} - \vec{b} \qquad \text{compl.} \qquad \overline{lo} \leq \vec{x} \leq \overline{hi}. \tag{6.60}$$

Also, constraint forces are often wanted as output, i.e.

$$\vec{F} = \mathbf{J}^T\vec{x}. \tag{6.61}$$

Thus, we regard the $\vec{b}$-vector as a single entity, and constraint force mixing is given as a diagonal matrix in vector form, $\vec{k}_{\text{cfm}}$, pre-multiplied by the time-step. Thus, our matrix-products have the following structure

$$\mathbf{A} = \mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T + \text{diag}(\vec{k}_{\text{cfm}}), \tag{6.62}$$

$$\vec{F} = \mathbf{J}^T\vec{x}. \tag{6.63}$$

Now, the question is: "given the special matrix products of the **A**-matrix, how do we efficiently solve the computations in the inner loop of the iterative SOR LCP method?". For convenience, these are refreshed in Figure 6.9. Let

```
delta = 0
for j=0 to idx-1 do
  delta += A(idx,j)*x(j)
next j
for j=idx+1 to n-1 do
  delta += A(idx,j)*x(j)
next j
delta = (b(idx) - delta)/A(idx,idx)
x(idx) = x(idx) + w*(delta - x(idx))
```

Figure 6.9: Inner loop of iterative SOR LCP method.

```
delta = b(idx) - d(idx)*x(idx)
for j=0 to idx-1
  delta -= J(idx,j) * F'(j)
next j
for j=idx+1 to n-1
  delta -= J(idx,j) * F'(j)
next j
x(i) = x(i) + delta
F' += J'(idx,.)*delta
```

Figure 6.10: Improved inner loop of the iterative SOR LCP method.

$$\vec{d}_i = \frac{\omega}{A_{i,i}}, \tag{6.64}$$

$$\vec{b}_i = \frac{\omega \vec{b}_i}{A_{i,i}} = \vec{b}_i \vec{d}_i, \tag{6.65}$$

$$\vec{F}' = \mathbf{M}^{-1}\mathbf{J}^T \vec{x}^{\,0}, \tag{6.66}$$

$$\mathbf{J}' = \mathbf{M}^{-1}\mathbf{J}^T, \tag{6.67}$$

$$\mathbf{J}_{i,\cdot} = \vec{d}_i \mathbf{J}_{i,\cdot}. \tag{6.68}$$

Here, the prime notation indicates a pre-multiplication by the inverse generalized mass-matrix. Observe that the last equation means that all elements in the i'th row of the Jacobian matrix are multiplied by the i'th element of the $\vec{d}$-vector. Using these precomputed values, the inner loop can be rewritten as shown in Figure 6.10. The benefits of this new loop is that it saves space since one does not have to compute all of $\mathbf{A}$, only its diagonal. By doing divisions in the preprocessing stage, they are taken out of the loop, thus lowering the cost, since divisions are typically more expensive than other operations.

By incrementally updating $\vec{F}'$, one avoids the matrix multiplication: $\vec{F}' = \mathbf{M}^{-1}\mathbf{J}^T \vec{x}^{\,k}$. Thus, it is far more inexpensive to perform this part. Another nice thing is that when computing `delta` and having $\vec{F}'$, one does not need to compute all of $\mathbf{A}$. There is no penalty in this, because the number of multiplications in the two terms are the same.

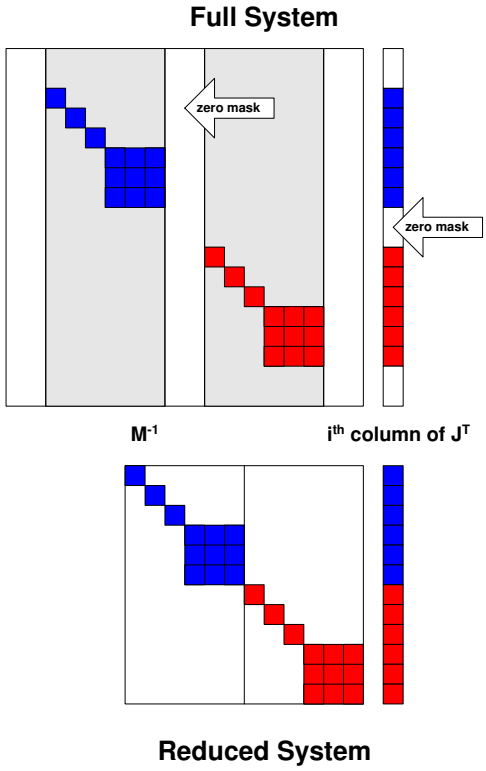


Figure 6.11: Using zero entries of the i'th column of $\mathbf{J}^T$ to reduce computations of the i'th column of $\mathbf{M}^{-1}\mathbf{J}^T$. Entries marked with blue correspond to the k'th body, and entries marked in red corresponds to the l'th body. White entries are zero.

6.1.6 Optimization of Matrix Computations

First, we will look into the computation of the matrix $\left(\mathbf{M}^{-1}\mathbf{J}^T\right)$. Notice that the dimensions of the matrix $\left(\mathbf{M}^{-1}\mathbf{J}^T\right)$ is the same as the matrix $\mathbf{J}^T$, since $\mathbf{M}^{-1}$ is a square symmetric matrix.

Let us try to compute the i'th column of the matrix $\left(\mathbf{M}^{-1}\mathbf{J}^T\right)$. This column is computed by taking the dot-product of each row in the $\mathbf{M}^{-1}$ with the the i'th column of the matrix $\mathbf{J}^T$. However, the matrix $\mathbf{J}^T$ is the transpose of $\mathbf{J}$. Thus the i'th column of $\mathbf{J}^T$ has identical entries with the i'th row in the matrix $\mathbf{J}$. The i'th row of $\mathbf{J}$ have a rather special form. Recall from theory that it describes a relationship between two bodies. We denote these bodies by the labels k and l

$$\begin{aligned}\mathbf{J}_{i,\cdot} = \big[0\ldots 0 \quad k_1 \quad k_2 \quad k_3 \quad k_4 \quad k_5 \quad k_6 \quad 0\ldots \\ \ldots 0 \quad l_1 \quad l_2 \quad l_3 \quad l_4 \quad l_5 \quad l_6 \quad 0\ldots 0\big]\,.\end{aligned} \tag{6.69}$$

That is, it has a total of 12 non-zero entries. The first six entries are related to the k'th body, and the last six to the l'th body. Each block of six non-zero entries can be further divided into three entries relating to translational motion, and the following three to the rotational motion. From this special form of the i'th row of the $\mathbf{J}$-matrix we see that the zero entries will mask out all the corresponding columns of the $\mathbf{M}^{-1}$-matrix. This is illustrated in Figure 6.11. Next, we will exploit the structure of the $\mathbf{M}^{-1}$-matrix. As can be seen from the figure, in the reduced system only mass-elements of the k'th body

are multiplied by the k-entries of the column of $\mathbf{J}^T$, and similarly for the l'th body. In fact, we see that the translational parts are simply multiplied by the inverse mass of the respective bodies. The rotational parts are a little more difficult since these are multiplied by the inverse inertia matrix of the respective bodies. Hence, the computations needed for computing the i'th column of the matrix $\mathbf{M}^{-1}\mathbf{J}^T$ is reduced to

$$\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{k_1,i} = \frac{1}{m_k}\mathbf{J}^T_{k_1,i}, \tag{6.70a}$$

$$\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{k_2,i} = \frac{1}{m_k}\mathbf{J}^T_{k_2,i}, \tag{6.70b}$$

$$\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{k_3,i} = \frac{1}{m_k}\mathbf{J}^T_{k_3,i}, \tag{6.70c}$$

$$\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{k_4..k_6,i} = \mathbf{I}_k^{-1}\mathbf{J}^T_{k_4..k_6,i}, \tag{6.70d}$$

$$\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{l_1,i} = \frac{1}{m_l}\mathbf{J}^T_{l_1,i}, \tag{6.70e}$$

$$\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{l_2,i} = \frac{1}{m_l}\mathbf{J}^T_{l_2,i}, \tag{6.70f}$$

$$\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{l_3,i} = \frac{1}{m_l}\mathbf{J}^T_{l_3,i}, \tag{6.70g}$$

$$\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{l_4..l_6,i} = \mathbf{I}_l^{-1}\mathbf{J}^T_{l_4..l_6,i}, \tag{6.70h}$$

where m_k and m_l are the respective masses of the bodies, and $\mathbf{I}_k^{-1}$ and $\mathbf{I}_l^{-1}$ are the respective inverse inertia tensors. All other entries in the i'th column are zero. Furthermore, we see that the matrix $\left(\mathbf{M}^{-1}\mathbf{J}^T\right)$ have the exact same pattern of zero-entries as the matrix $\mathbf{J}^T$.

Now, let us try to compute the (i,i) entry of the matrix $\mathbf{A}$ consisting of the matrix product $\mathbf{J}$ and $\left(\mathbf{M}^{-1}\mathbf{J}^T\right)$. This entry corresponds to the dot-product between the i'th row of the $\mathbf{J}$-matrix and the i'th column of the $\left(\mathbf{M}^{-1}\mathbf{J}^T\right)$-matrix. However, as we saw in the computation of the $\left(\mathbf{M}^{-1}\mathbf{J}^T\right)$-matrix, it has the same pattern of zero entries as the $\mathbf{J}^T$-matrix. This means that it is particularly easy to compute the dot-product

$$\begin{aligned}
A_{i,i} &= \mathbf{J}_{i,\cdot}\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{\cdot,i} \\
&= \mathbf{J}_{i,k_1}\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{k_1,i} + \mathbf{J}_{i,k_2}\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{k_2,i} + \\
&\quad \mathbf{J}_{i,k_3}\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{k_3,i} + \mathbf{J}_{i,k_4}\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{k_4,i} + \\
&\quad \mathbf{J}_{i,k_5}\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{k_5,i} + \mathbf{J}_{i,k_6}\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{k_6,i} + \\
&\quad \mathbf{J}_{i,l_1}\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{l_1,i} + \mathbf{J}_{i,l_2}\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{l_2,i} + \\
&\quad \mathbf{J}_{i,l_3}\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{l_3,i} + \mathbf{J}_{i,l_4}\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{l_4,i} + \\
&\quad \mathbf{J}_{i,l_5}\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{l_5,i} + \mathbf{J}_{i,l_6}\left(\mathbf{M}^{-1}\mathbf{J}^T\right)_{l_6,i}.
\end{aligned} \tag{6.71}$$

When applying the matrix optimizations, a neat little trick can be used to make the inner loop of the iterative scheme really tight. For instance, the usual Gauss-Seidel iterative LCP solver is given by the scheme

$$\vec{x}^{\,k+1} = \mathbf{D}^{-1}\left(\vec{b} - \mathbf{L}\vec{x}^{\,k+1} - \mathbf{U}\vec{x}^{\,k}\right), \tag{6.72}$$

Using the factorization implementation trick, the update would be

$$\vec{x}^{\,k+1} = \mathbf{D}^{-1}\left(\vec{b} - \mathbf{L}\vec{x}^{\,k+1} - \mathbf{U}\vec{x}^{\,k} - \mathbf{D}\vec{x}^{\,k}\right), \tag{6.73}$$

since there is no way to separate the i'th variable from the term $\mathbf{M}^{-1}\mathbf{J}^T\vec{x}$. This is not the iterative scheme we want. However, if we multiply through by the inverse diagonal matrix, we notice that the last term is simply a subtraction of the solution vector. Thus, we can get the equivalent to the original Gauss-Seidel iterative scheme by adding the solution vector, i.e.

$$\vec{x}^{\,k+1} = \mathbf{D}^{-1}\left(\vec{b} - \mathbf{L}\vec{x}^{\,k+1} - \mathbf{U}\vec{x}^{\,k} - \mathbf{D}\vec{x}^{\,k}\right) + \vec{x}^{\,k}. \tag{6.74}$$

The benefit is that we still have a tight inner loop of the iterative method, where we do not need to compute the big $\mathbf{A}$-matrix, nor worry about index values of the variables.

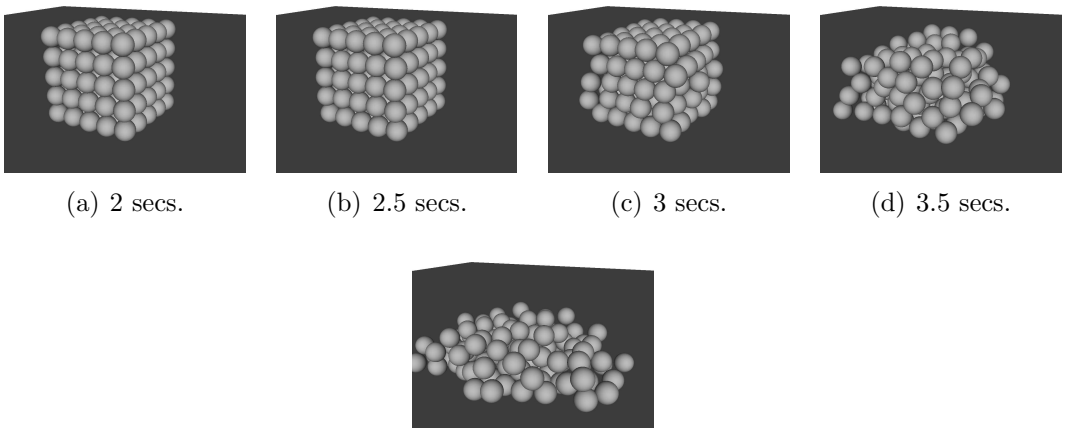
6.1.7 Applying an iterative LCP solver.

It turns out that SOR is really not worth the effort in practice. Instead a lean and mean Gauss-Seidel implementation works the best. The best approach is to just run a fixed number of iterations and simply drop all other convergence test and setting up more advanced stopping criteria. The inner loop of the Gauss Seidel implementation is so tight and efficient that the extra computations involved in evaluating convergence is a waste of time. The same can be said about permutation or reordering of variables. These observations are connected to the fact that we usually only use 10 iterations in the iterative LCP solver. The number 10 was chosen because it gave reasonable performance results.

Warm-starting the LCP solver is actually not as beneficial as it may sound. It turns out that when the contact state changes, like an object sliding over a ridge, then the LCP solution can change dramatically. In these cases, warm-starting means that one can start with a previous solution far from the current true solution. Convergence is thus worse by warm-starting in these cases. In Computer Animation it is interesting to see objects move around. Changing contact states is therefore very common.

According to [107] iterative methods are characterized by

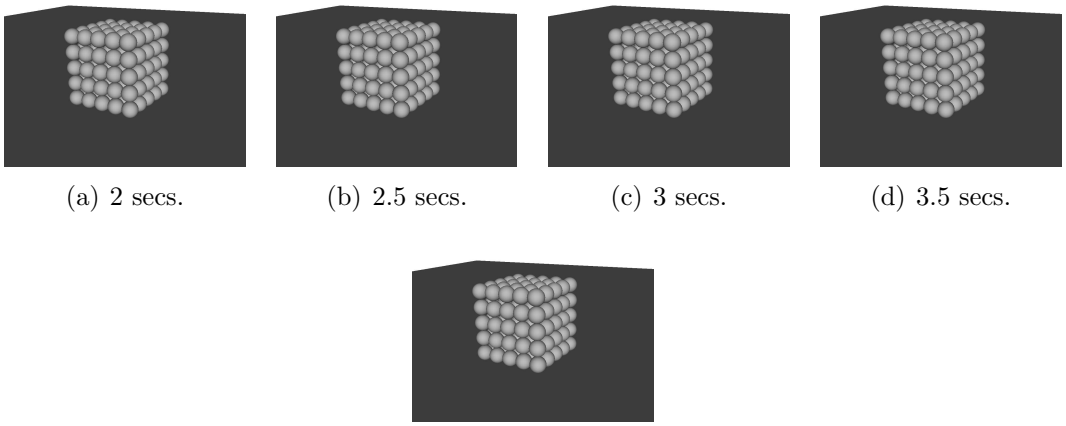
- Iterative methods tend to pick a mean solution when there are multiple solutions.
- Gauss-Seidel, linear convergence at best, but can be much worse.
- Conjugate Gradient, fastest convergence, but requires restart when the "active" set changes.
- SOR: Possible energy gain.
- Jacobi: Terrible convergence.
- Less than 10 iterations to be competitive.



(a) 2 secs. (b) 2.5 secs. (c) 3 secs. (d) 3.5 secs.

(e) 4 secs.

Figure 6.12: A grid stack of 125 balls, using, 10 iterations with Gauss Seidel.



(a) 2 secs. (b) 2.5 secs. (c) 3 secs. (d) 3.5 secs.

(e) 4 secs.

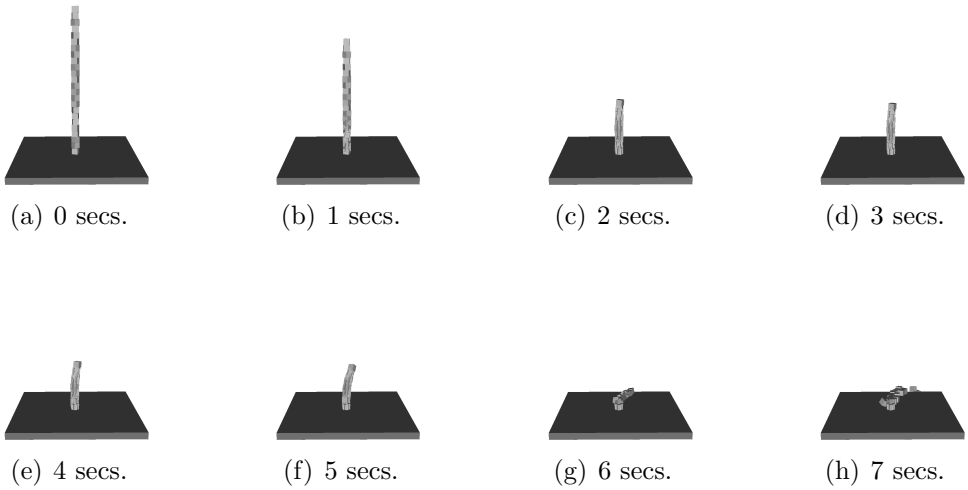
Figure 6.13: A grid stack of 125 balls, using, 100 iterations with Gauss Seidel.

All simulations in this section were done using an explicit time-stepping scheme, and a velocity based complementarity formulation, implemented using the friction model described in Section 6.1.4. The time-step was set to 0.01 seconds. The coefficient of friction was 0.25 and coefficient of restitution was 0.25, in order to mimic typical real-world values.

Figures 6.12-6.13 show a stack of 125 balls placed in a grid formation. The balls have a density of $10\frac{Kg}{m^3}$ and a radius of $\frac{1}{2}m$. As the figures show, even with 10 iterations it takes 2.5 seconds before the simulation error accumulates to a size where the ball stack collapses. With 100 iterations the accuracy of the simulation is good enough for maintaining the grid structure for long simulation times.

Clearly, Figures 6.12-6.13 show that the quality of the simulation results can be improved simply by increasing the number of iterations. From a time-critical computation view-point this is an attractive property of the simulation.

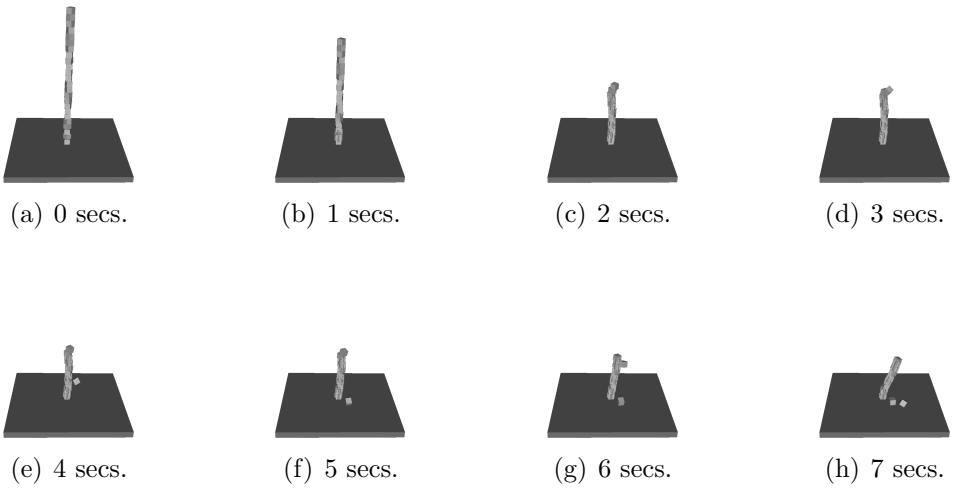
We have a mathematical model, which is well posed in the sense that it guarantees solution existence and convergence towards a solution. The numerical method has a single parameter, the maximum number of iterations, which we can turn up and down in order



(a) 0 secs. (b) 1 secs. (c) 2 secs. (d) 3 secs.

(e) 4 secs. (f) 5 secs. (g) 6 secs. (h) 7 secs.

Figure 6.14: A stack of 25 boxes on top of each other, using 10 iterations with Gauss Seidel.



(a) 0 secs. (b) 1 secs. (c) 2 secs. (d) 3 secs.

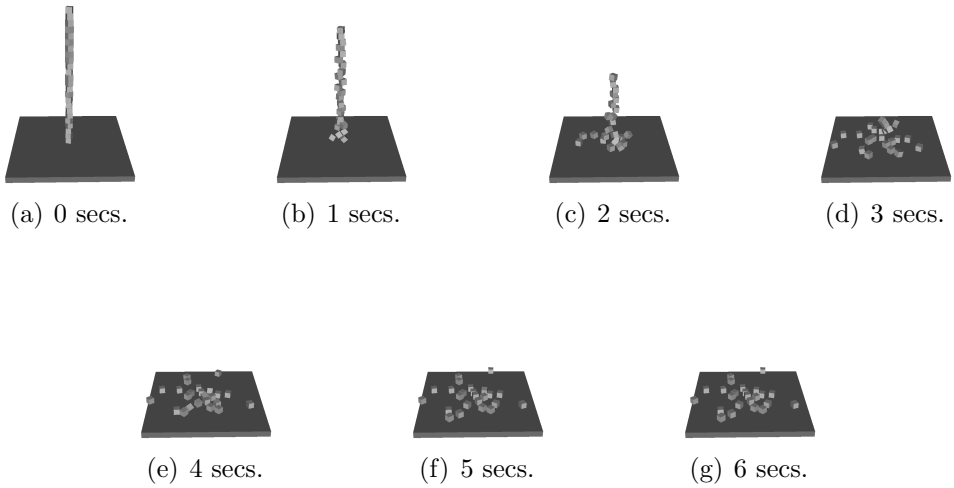
(e) 4 secs. (f) 5 secs. (g) 6 secs. (h) 7 secs.

Figure 6.15: A stack of 25 boxes on top of each other, using 100 iterations with Gauss Seidel.

to obtain either a fast plausible motion or an accurate motion that agrees completely with the mathematical model, all depending on the amount of computation time we have available.

Figures 6.14-6.15 show a numerically more challenging configuration. Here, 25 boxes, all with a density of $10\frac{Kg}{m^3}$ and edge lengths of $1m$, are placed on top of each other. The figures clearly show that the simulations using 10 and 100 iterations fail miserably. However, it is seen that using more iterations do improve the simulation quality, since the stack structure is destroyed at a slower rate.

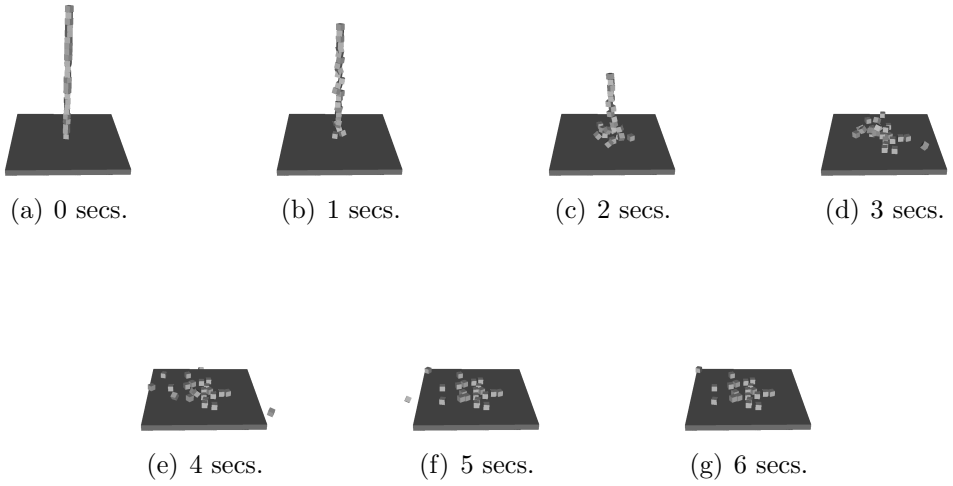
With the box stack configuration it becomes interesting to see what happens when we increase the number of iterations further. This is done in Figures 6.16-6.17. As can be seen, the boxes behave more and more rigidly as the iterations are increased. However, even for these faulty simulation results the iteration count has exceeded far beyond the computational burden we are willing to pay in a computer graphics application. In



(a) 0 secs. (b) 1 secs. (c) 2 secs. (d) 3 secs.

(e) 4 secs. (f) 5 secs. (g) 6 secs.

Figure 6.16: A stack of 25 boxes on top of each other, using 1000 iterations with Gauss Seidel.



(a) 0 secs. (b) 1 secs. (c) 2 secs. (d) 3 secs.

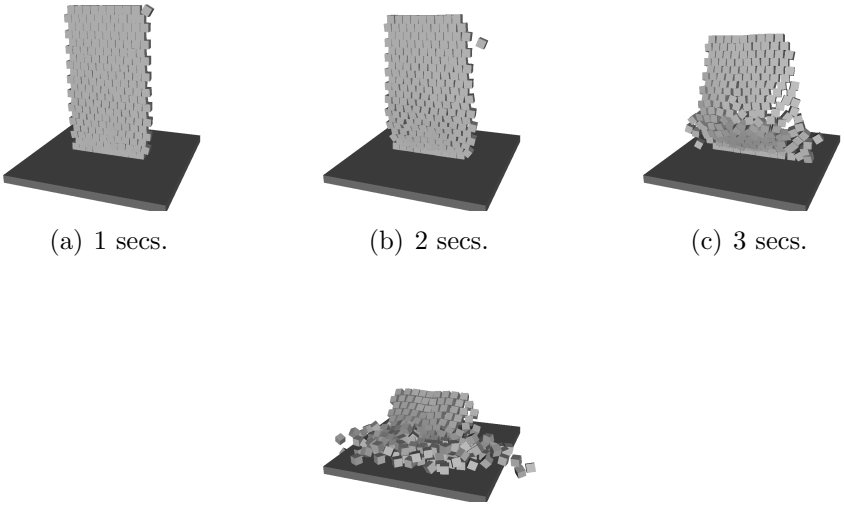
(e) 4 secs. (f) 5 secs. (g) 6 secs.

Figure 6.17: A stack of 25 boxes on top of each other, using 10000 iterations with Gauss Seidel.

Section 6.4 we will present a solution for handling the bad convergence property of this box-stack configuration. The proposed solution will not exceed more than the order of 10 iterations.

We believe that the problem with the slow convergence rate of the box stack configuration is related to the fact that the stress distribution caused by the top-most boxes only has one direction to go. That is directly downwards. To support this claim we have tried to simulate a brick wall with 200 bricks. The results of these simulations can be seen in Figures 6.18-6.19. The bricks have the same physical properties as the boxes in the box stack. However, this time a noticeable stability is seen when the number of iterations is increased to 100.

The structure of the wall appears a little rubber-like near the bottom, and given enough simulation time (close to 10 seconds), the wall will break into two pieces at the middle. Added error correction by projection to the simulation further improves the



(a) 1 secs. (b) 2 secs. (c) 3 secs.

(d) 4 secs.

Figure 6.18: A wall of 200 bricks, using 10 iterations with Gauss Seidel.

Method	Convergence (iterations)
Jacobi	$O(N^2 \log(\frac{1}{\varepsilon}))$
Gauss-Seidel	$O(N^2 \log(\frac{1}{\varepsilon}))$
SOR (with optimal w)	$O(N^{3/2} \log(\frac{1}{\varepsilon}))$
Conjugate Gradient (w/o. pre-cond.)	$O(N^{3/2} \log(\frac{1}{\varepsilon}))$

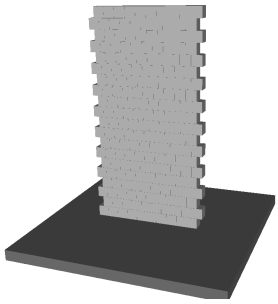
Table 6.2: Convergence rates of iterative LCP solvers. N is number of variables, ε is wanted accuracy.

simulation results, as can be seen from Figures 6.19(d)-6.19(e).

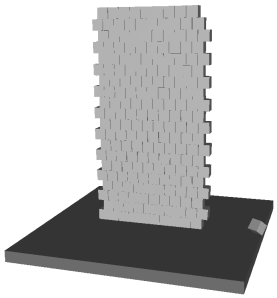
The simulation results of the wall are very promising. However, even more complex structures can be simulated with only 100 iterations. In Figure 6.20 a brick tower simulation is performed. The bricks have dimension of $1.5m \times .5m \times .5m$ and a density of $10\frac{Kg}{m^3}$. Notice the bulge shape near the bottom of the tower. All-though the simulation looks nice, it is not the physical behavior we expect from a brick tower. Figure 6.21 shows what happens when we increase the number of iterations to 100. Notice that after 8 seconds of simulation a very small bulge can be seen. Adding correction almost completely removes this bulge.

As our simulation results presented in this section indicate, quite complex and challenging simulations can be done with an iterative LCP solver. However the results also show that large scale stacking is numerically challenging for an iterative LCP solver. The convergence rate is a particular interesting topic. The bad behavior of the box-stack compared to the good behavior of the three other simulations suggests that a stacking configuration can have different numerical properties depending on the exact structure of the stacking.

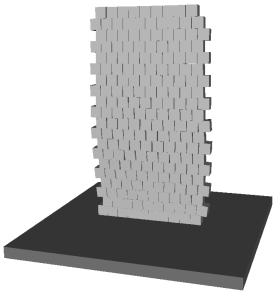
Generally speaking, the convergence rates for a Gauss-Seidel method applied to multibody dynamics is of order $kO(N^2)$, where N is the number of contact points and k is a constant dependent on the accuracy of the solution. Table 6.2 contains more detailed complexities due to [137].



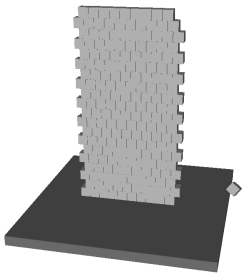
(a) Initial position 0 secs.



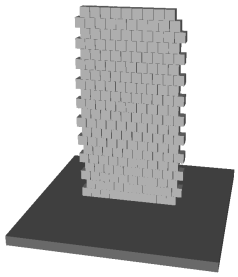
(b) 100 iterations after 4 secs.



(c) 100 iterations after 8 secs.



(d) 100 iterations with error correction after 4 secs.



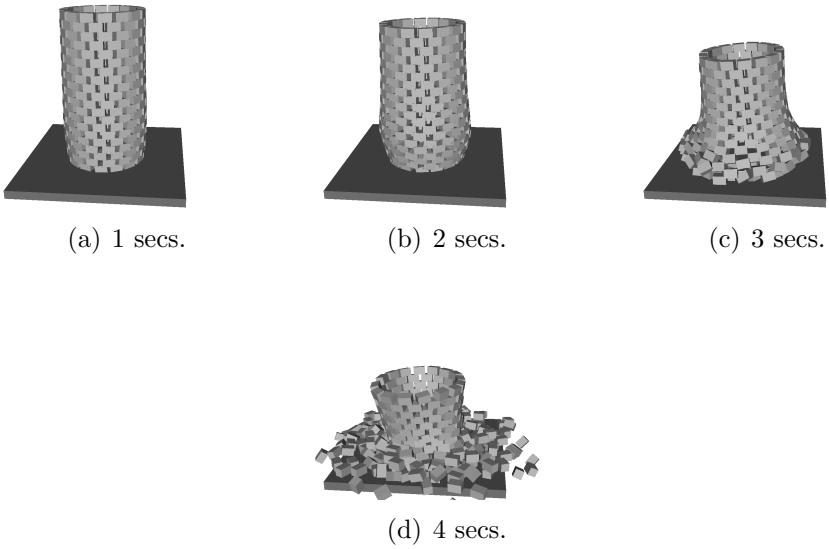
(e) 100 iterations with error correction after 8 secs.

Figure 6.19: A wall of 200 bricks, using Gauss Seidel with and without error correction by projection.

The convergence rates from Table 6.2 do not say anything about the constants involved. Obviously, the box-stack have high constants compared to the other configurations. A possible direction for further work is to investigate the effect of using preconditioners to speed the convergence rate or possibly use other iterative solvers or even multigrid solvers.

In Table 6.3 we have listed the minimum, mean, and maximum frame times for the first 500 time-steps of the ball grid, wall, and tower configurations. Notice that even with 100 iterations, these complex configurations are simulated in reasonable time. Only the ball grid and the wall are done at interactive rates. Clearly, we could increase the number of iterations noticeably and still be within the domain of computer graphics. For instance, [70] reports frame times of 5-7 minutes for configurations of similar complexity to our tower configuration.

Figure 6.22 shows how the frame time changes as we increase the number of iterations



(a) 1 secs. (b) 2 secs. (c) 3 secs.

(d) 4 secs.

Figure 6.20: A tower of 320 bricks, using 10 iterations with Gauss Seidel.

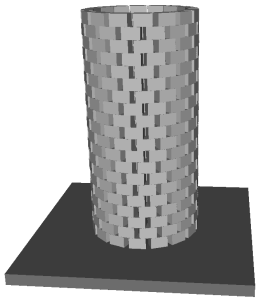
	configuration	min (secs.)	mean (secs.)	max (secs.)
ball	10	0,0100	0,0168	0,0210
wall	10	0,0100	0,0268	0,0700
tower	10	0,0300	0,0513	0,1000
ball	100	0,0200	0,0229	0,0310
wall	100	0,1000	0,1172	0,1600
tower	100	0,2100	0,2179	0,2410

Table 6.3: Frame timings for the first 500 frames of the ball grid, wall, and tower simulations.

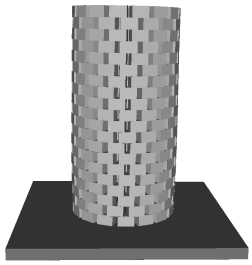
used in the box-stack configuration. Notice that the plots scale linearly wrt. the number of iterations as expected. Thus, we can apply this knowledge to estimate how long it will take to run a simulation given the number of iterations used in the iterative LCP solver.

Finally, we have done some convergence testing with a model of an idealized box-stack configuration. Our results are shown in Figure 6.23. The configuration consists of two balls resting on top of each other on a fixed support plane. The radius of the lower ball is $\frac{1}{2}m$, and the radius of the upper ball is $1m$. The density of the lower ball is set to $1\frac{Kg}{m^3}$ and the density of the upper ball is $1000\frac{Kg}{m^3}$. There is no friction and the coefficient of restitution is zero between all objects except between the two balls where it is set to 1. This configuration is the most simple example illustrating that large mass ratios can cause bad convergence for the iterative LCP solver. The two ball configuration is related to the box-stack configuration in the sense that the upper ball represents the same effect as the weight of the 24 topmost boxes.

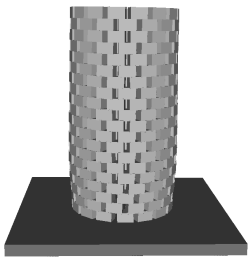
The two ball configuration actually only contains two variables in the LCP problem, but still it requires roughly 64000 iterations in the iterative Gauss-Seidel to converge towards a solution that is accurate enough for a stable visual result. The box stack contains at most $8 \times 24 \times 3 = 576$ variables. It is not a pleasing thought to think of how many iterations there might be needed for such a large problem exhibiting the same



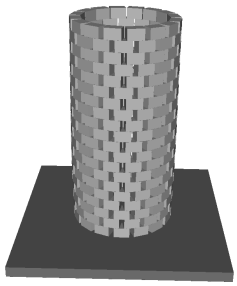
(a) Initial position 0 secs.



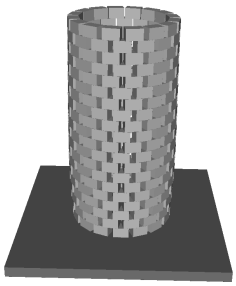
(b) 100 iterations after 4 secs.



(c) 100 iterations after 8 secs.



(d) 100 iterations with error correction after 4 secs.



(e) 100 iterations with error correction after 8 secs.

Figure 6.21: A tower of 320 bricks, using Gauss Seidel with and without error correction by projection.

effects as the simple 2 variable ball example.

6.2 Collision Detection using Signed Distance Maps

In our simulator we applied the signed distance map collision detection technique presented in [70]. Its simplicity and ease of use makes it very attractive. Also, it is well-suited for contact point generation, i.e. contact point normal generation and penetration depth measures, which is usually very difficult using traditional collision detection algorithms.

The basic idea is to keep a double representation for each object in the simulation: a model frame signed distance map and a model frame triangle mesh. When performing collision detection between objects, the vertices of the mesh of one object is looked up in the signed distance map of the other object, and vice versa. All vertices lying inside

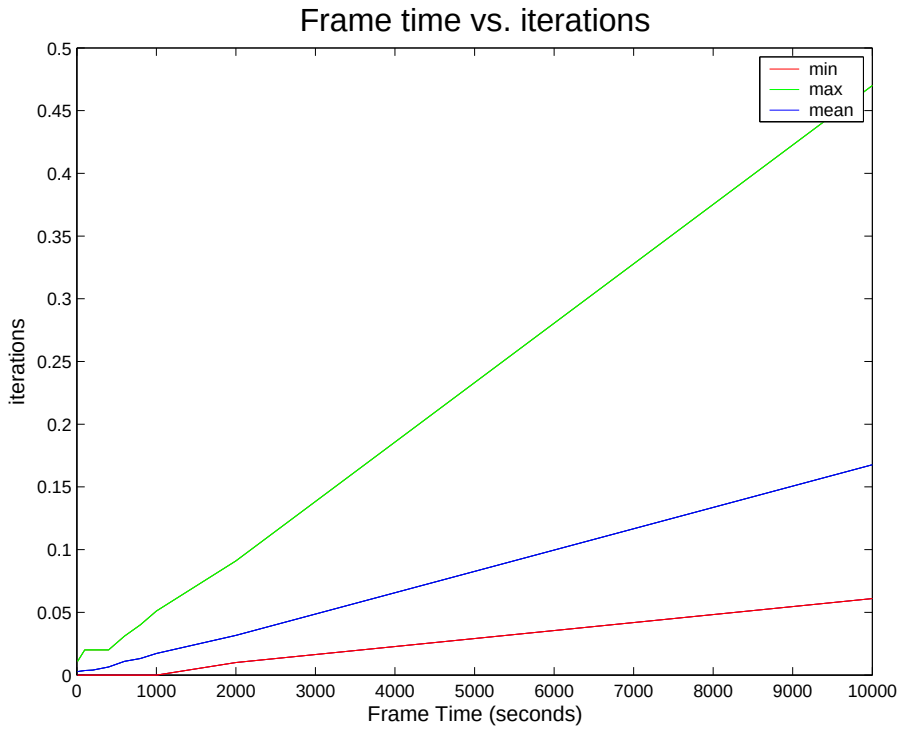


Figure 6.22: Frame times of the box stack configuration as a function of increasing iterations: $10, 100, 200, 400, \ldots, 1000, 2000, 10000$.

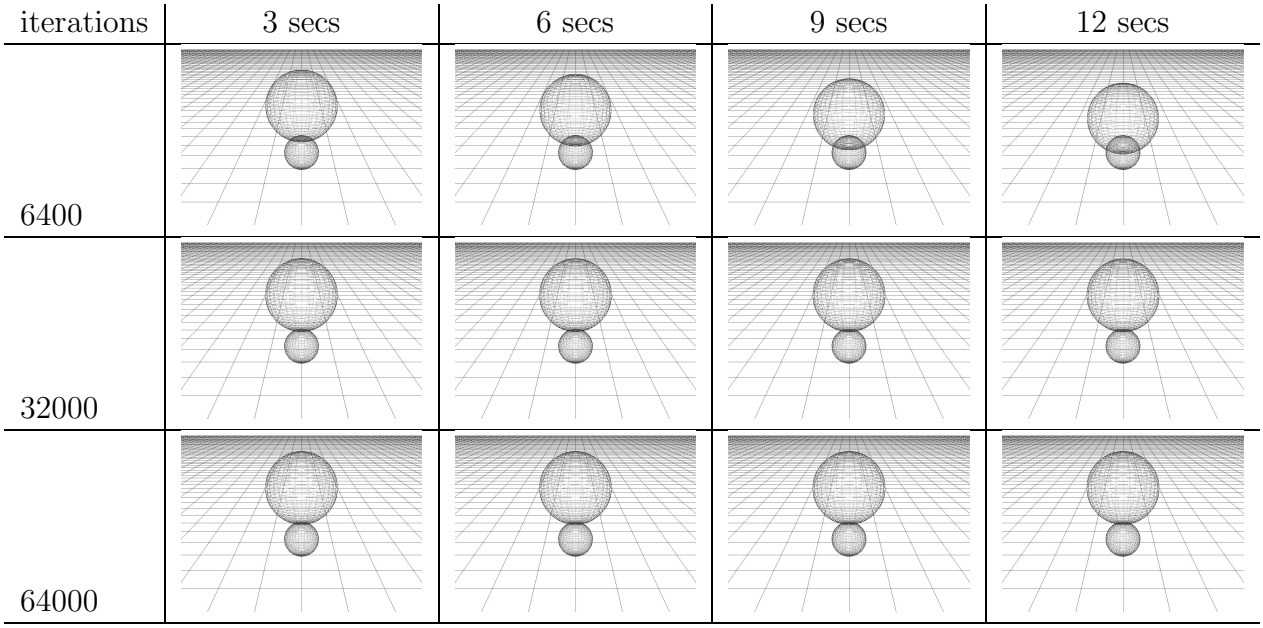


Figure 6.23: Two ball configuration with large mass ratios. For 64000 iterations the simulation becomes visually stable. Frame times take rougly 0.01 seconds when using 64000 iterations per time-step.

or on the zero-level-set surface of the signed distance map are used to generate a contact point. The actual value at the vertex is used as a measure of penetration depth, and the gradient at the vertex as the contact normal. Due to space considerations, we refer the reader to [70] for more details.

Using only the vertices of the mesh requires a very fine tessellated mesh to ensure that possible edge-face penetration will not hurt the eye of an observer or equivalently, edges must be subject to special treatment.

In the original paper several acceleration techniques are presented. Basically, there are three major techniques: Octree representation of signed distance maps to reduce memory requirements, usage of bounding volume hierarchies (BVHs) of the sampling points to improve the speed of the collision queries, and finally storage of boolean values in the signed distance map grid-nodes in order to avoid performance penalties from interpolation of sampling points that clearly are far from the zero-level-set surface.

It is clear that the edge-face case is not easily dealt with using this approach. The edge-face case is not well-described in the original paper, but from personal communications with the corresponding author [70], it was clarified that the edge-face case was handled by interference-query of mesh-edges against mesh faces. Thus, the actual collision testing of edge-face cases is not performed using the signed distance map. Only the contact generation uses the signed distance maps in case of the edge-face case.

We found the signed distance map approach attractive and have sought to implement it. We have addressed several issues. Firstly, we want to be able to handle arbitrary meshes, even if they are coarsely tessellated. Secondly, we want to handle the unpleasant edge-face case without introducing special cases. Thirdly, we want to keep the number of sampling points as low as possible to improve performance. Finally, we want to make collision queries as fast as possible.

In order to achieve these goals, we propose to re-sample the initial mesh to produce more attractive sample points, and further, we store the new point sampling in a sphere-tree. The point sampling is discussed in Section 6.2.1 and sphere-threes in Section 6.2.2.

It is very attractive to handle edge-face cases using point re-sampling. This means that the entire collision query consist of the same simple test: looking a point up in a signed distance map. This kind of test is easy to vectorize in order to run in parallel. Thus using special hardware or parallel machines may yield superior performance.

In the following we will study the properties of the signed distance map algorithm using a box and a cow object as examples. The box was chosen due to its simple geometry which allow us to analyze potential problems more easily. In a real application box collisions would be more efficiently handled using primitive testing such as the one in Section 6.3.

6.2.1 Point Sampling

We have taken a different approach to handle the unpleasant edge-face case. Instead of keeping the mesh, we completely throw this away and work with a point sampling instead. The mesh is only used during a pre-processing phase for generating the point sampling.

As in [70], we use the vertices as sample points. However, we prune all vertices lying in flat regions on the mesh surface. That is, only vertices that have at least one concave or convex incident edge are used as sample points. The remaining vertices are simply

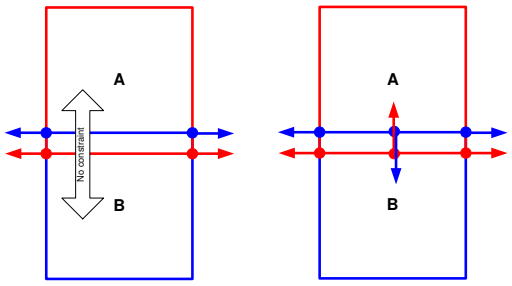


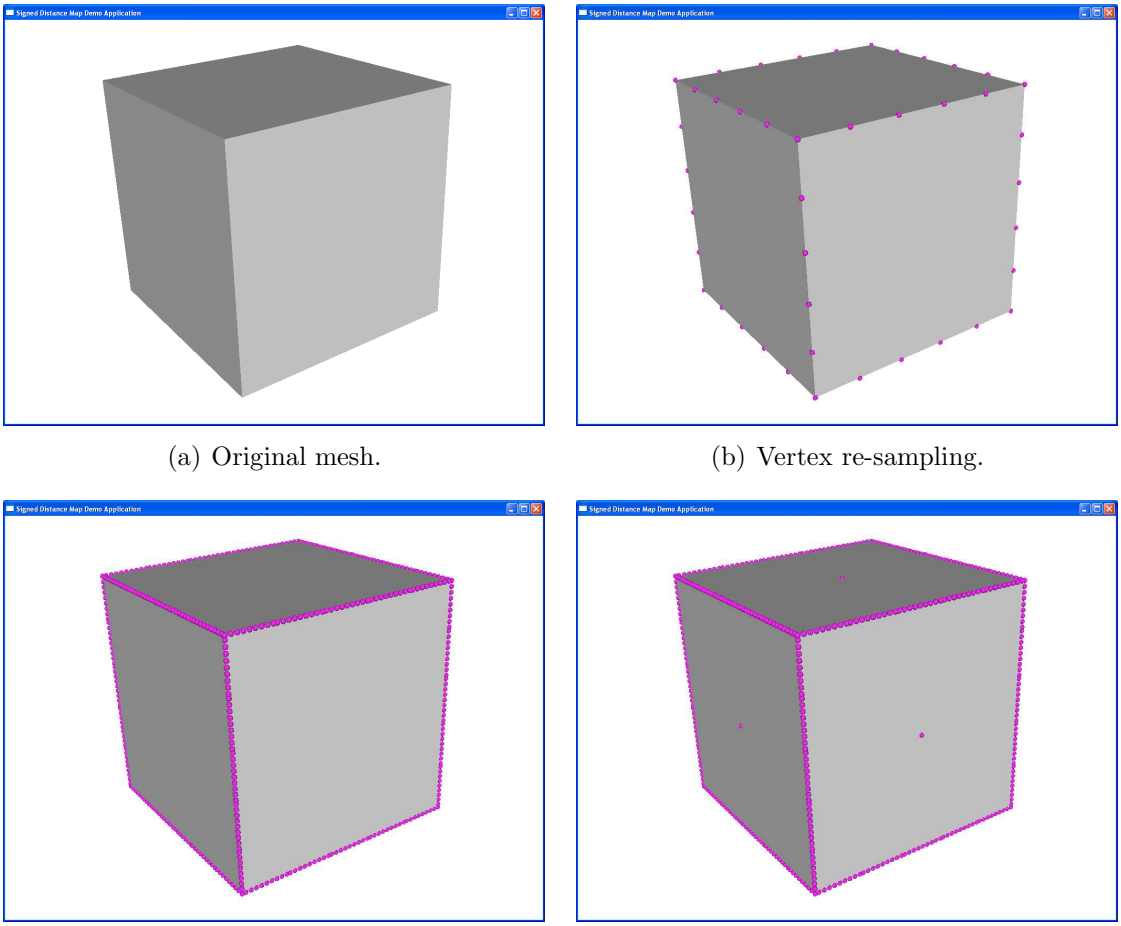
Figure 6.24: An illustration showing why face re-sampling is needed. On the left no face re-sampling is used, and box A is thus not prevented from sinking down into box B, while on the right, a face sampling point ensures a contact point is generated to prevent penetration.

pruned. Vertices lying in flat regions can be ignored, because if a collision occurs in these parts, it will be due to a sampling point from another object penetrating the zero-level-set surface.

Next, we loop over all the edges in the mesh. If an edge is encountered with non-planar neighboring faces and a length greater than a sampling threshold then the edge is simply uniformly sub-sampled into sampling points lying the sampling threshold distance apart from each other. As sampling threshold we use the maximum value of a user specified threshold and the diagonal of a grid cell in the signed distance map. This way, the user can achieve a sampling density along edges which corresponds to the resolution of the signed distance map simply by setting the user specified threshold equal to zero. On the other hand, the user may want a coarse sampling density in order to lower the number of sampling points and thereby gain performance over accuracy. Creating a point sampling this way allows for a consistent treatment of the edge-face case, without having to switch to another representation. Also, an end user has the means of picking accuracy over performance or vice-versa simply by adjusting a single threshold value.

The outlined point sampling scheme is sufficient for detecting contacts. However, due to the local nature of the contact point generation, these contacts may not always yield the expected dynamic behavior. For instance, two boxes perfectly aligned on top of each other, but deeply penetrating, will not generate any contact points with a normal in the "penetration" direction. In fact, this simulation example will result in the top-box sinking all the way down into the bottom-box. This is illustrated in Figure 6.24. To remedy this problem, we re-insert sampling points on flat surfaces of the object. A breadth first traversal is done over the mesh surface to collect regions of coplanar faces. For each such region a single centroid point is computed as the average point of all the face vertices in the region. This centroid point is then added to the point sampling. Now, in the case of the two boxes, each of the aligned faces will have a center-sampling point that will cause a contact point to be generated with the wanted contact normal direction. To make sure that we do not generate too many sampling points, the area of the flat regions are computed, and only in cases where the area is significantly larger than the area of the maximum side of a grid cell, the center point added to the point sampling.

Figure 6.25 and Figure 6.26 show point re-samplings of a simple box object and a complex cow object. Table 6.4 and Table 6.5 contain mesh statistics and sampling point counts. Notice that the number of sample points is increased dramatically but only along



(a) Original mesh. (b) Vertex re-sampling.

(c) Vertex and edge re-sampling. (d) Vertex, edge and face re-sampling.

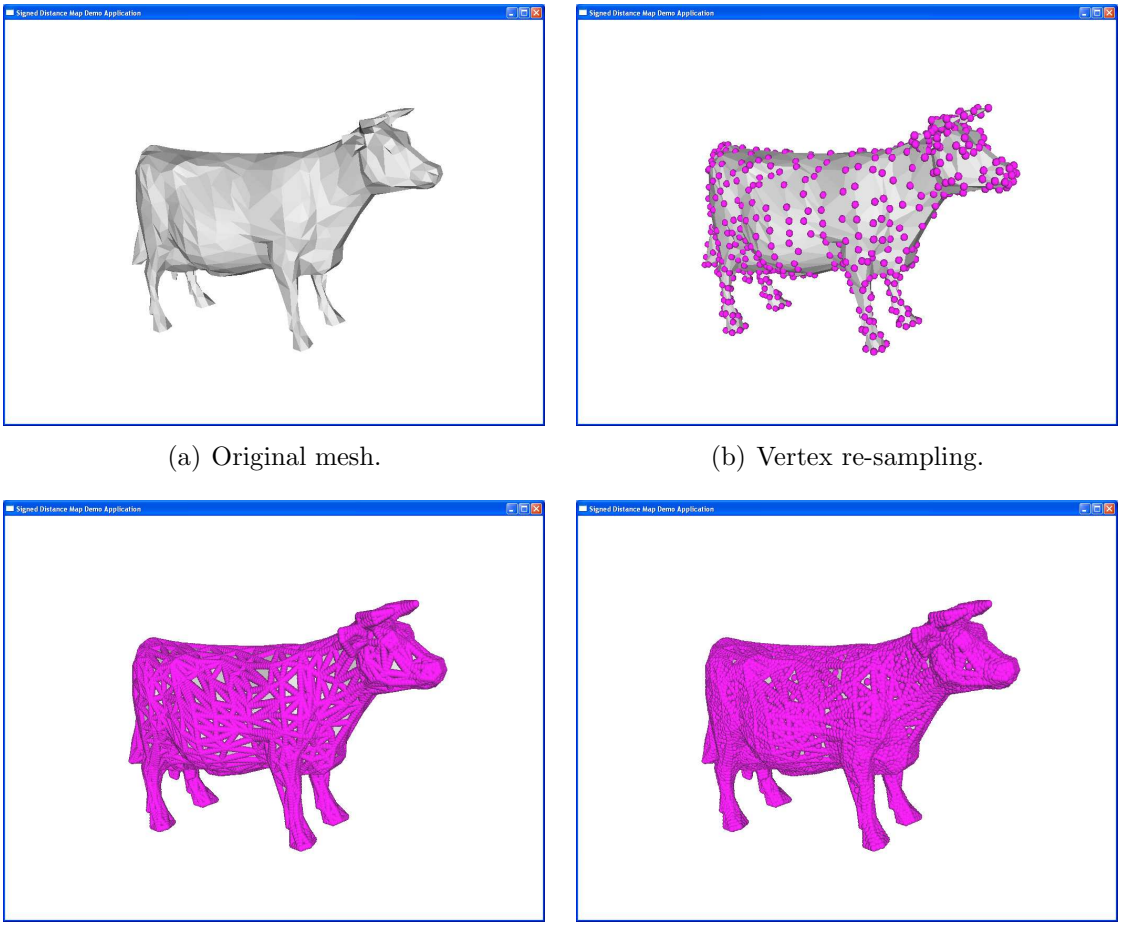
Figure 6.25: Point re-sampling of box object.

Re-sampling Type	BVH nodes	Sample Points
Only vertices	65	56
Edge sampling	745	536
Edge and face sampling	754	542

Table 6.4: Statistics of box object, mesh has 152 vertices, 450 edges, and 300 faces.

non-flat edges. Notice that in the cow example many faces are not re-sampled with a sample point, due to their small size compared to the grid size of the signed distance map used.

The effect of the point re-sampling does have some draw-backs which are illustrated in Figure 6.27. The major problem is that the number of contact points almost explodes. The figure also shows some of the difficulties with using signed distance maps. Notice how the contact normals twist around at edge-edge crossings (special case of edge-face contacts). Clearly, these normals are not what we would expect. In practice this is often not a problem. However, if penetrations occur, the local property of the contact normals can cause a simulation blow-up as the one shown in Figure 6.37. Our experience indicates that aggressive oversampling of edges increases the chance of such blow-ups.



(a) Original mesh.

(b) Vertex re-sampling.

(c) Vertex and edge re-sampling.

(d) Vertex, edge and face re-sampling.

Figure 6.26: Point re-sampling of cow object.

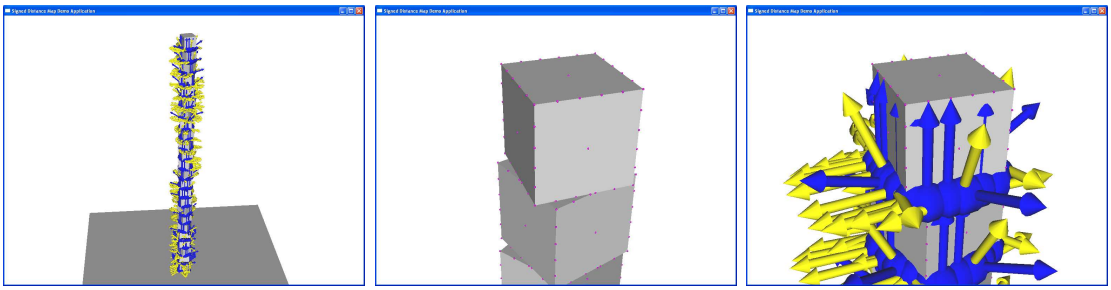
Re-sampling Type	BVH nodes	Sample Points
Only vertices	999	752
Edge sampling	15890	11995
Edge and face sampling	17836	13495

Table 6.5: Statistics of cow object, mesh has 752 vertices, 2250 edges, and 1500 faces.

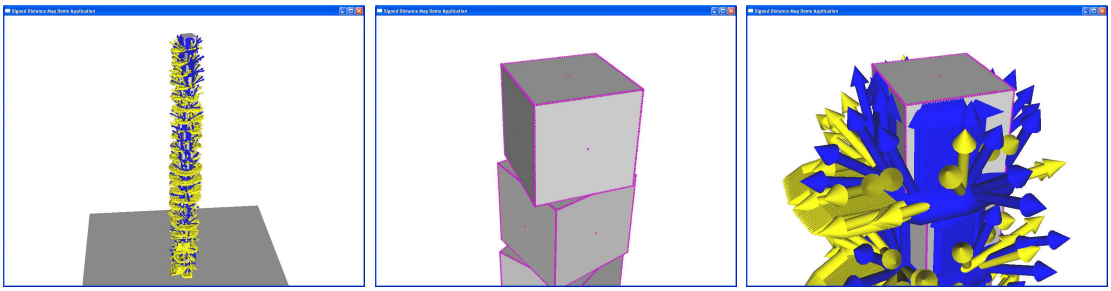
Figure 6.27 also clearly suggests that contact reduction [26, 108] is unlikely to be useful. Even though the contact region is a entirely flat polygon, contact normals are pointing in so many different directions that any analysis for contact reduction would try to create multiple contact regions.

6.2.2 Sphere Tree Acceleration

To speed up the testing of sample points against signed distance maps, we have tried to use sphere-trees. These were built using an octree top-down splitting strategy [67, 41, 27]. That is, in each split, a set of points are subdivided into at most eight subsets. First, the



(a) Sensible sampling.



(b) Oversampling.

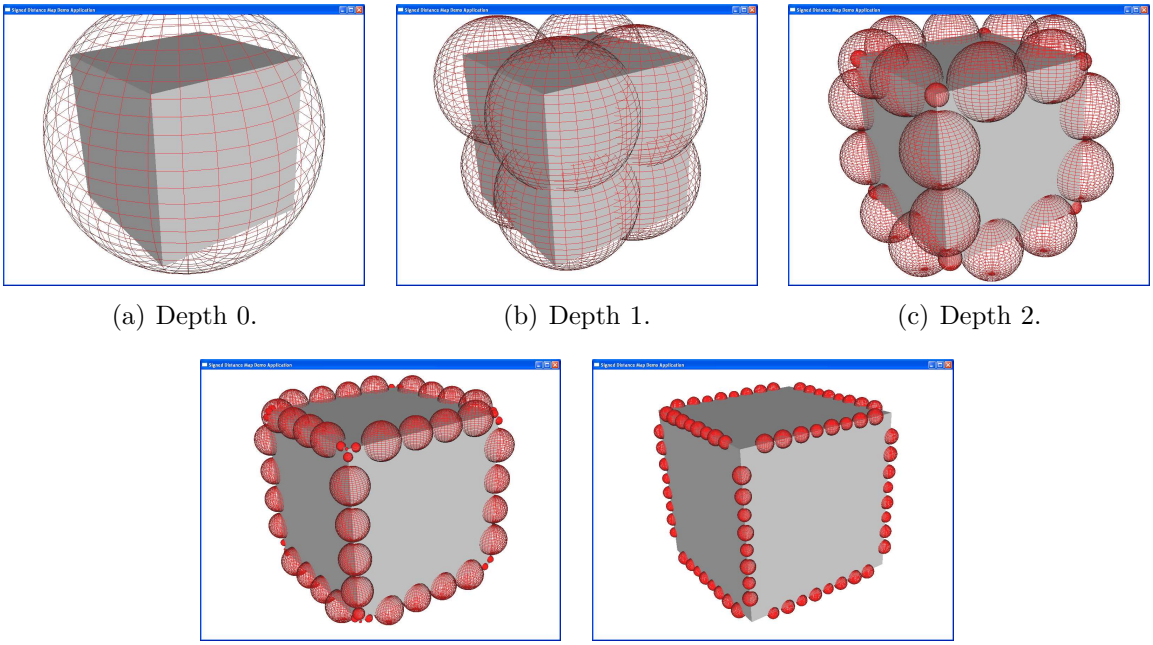
Figure 6.27: The difference between sensible re-sampling and oversampling. Notice how normals rotate when edges cross. Where they are horizontal instead of vertical due to the local property of signed distance maps.

mean sample point is computed and used as the origin in a local axis aligned coordinate frame. Then, each sample point is mapped into the quadrants of this local frame. Upon completion, a subset of sample points is created, one for each non-empty quadrant. If a sample point-set only contains 8 sample points then each sample point simply generates a one-point large subset. This splitting scheme was chosen due to its simplicity. We did try a splitting scheme based on a maximum covariance axis. However the generated binary BVHs were inferior compared to the octree BVH. The actual sphere fitting was done using the randomized algorithm of Welzl [151], yielding best fitting spheres.

Figure 6.28 and Figure 6.29 show the spheres at increasing depths in the sphere-trees of the cow and box object. Table 6.4 and Table 6.5 contain node counts for the sphere trees when applying different re-sampling strategies.

Spheres were chosen due to the simplicity by which they can be tested against a signed distance map. That is, during a collision query we perform a single traversal of the sphere BVH, testing it against the signed distance map of another object. Before testing a sphere against the signed distance map, it is transformed into the model frame of the signed distance map. If the sphere center lies outside the signed distance map, but the sphere surface intersects the AABB of the signed distance map, then we descend to the children of the sphere. If the center of the sphere lies inside the signed distance map, then we look up the signed distance values of the eight surrounding grid nodes. If the minimum distance of these are less than the sphere radius plus the collision envelope then we descend the sphere. Otherwise we simply prune the sphere.

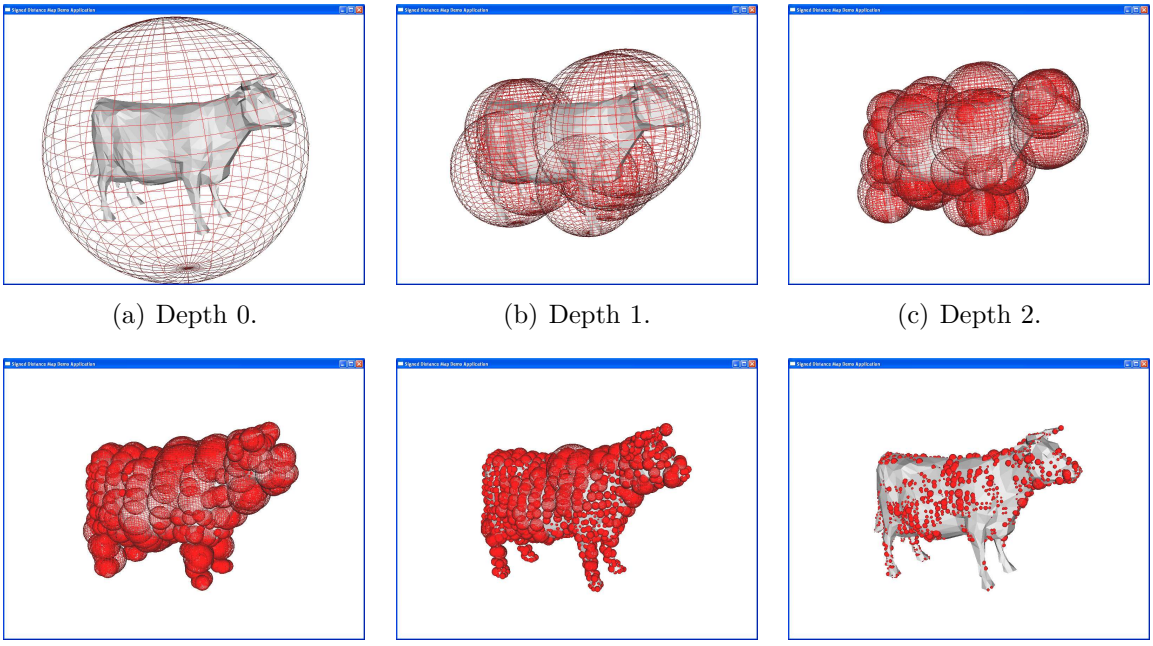
Leaf spheres are simply the sampling points, and these can be treated by doing an interpolation of the eight surrounding distance values, if the sphere was not pruned. If



(a) Depth 0. (b) Depth 1. (c) Depth 2.

(d) Depth 3. (e) Depth 4.

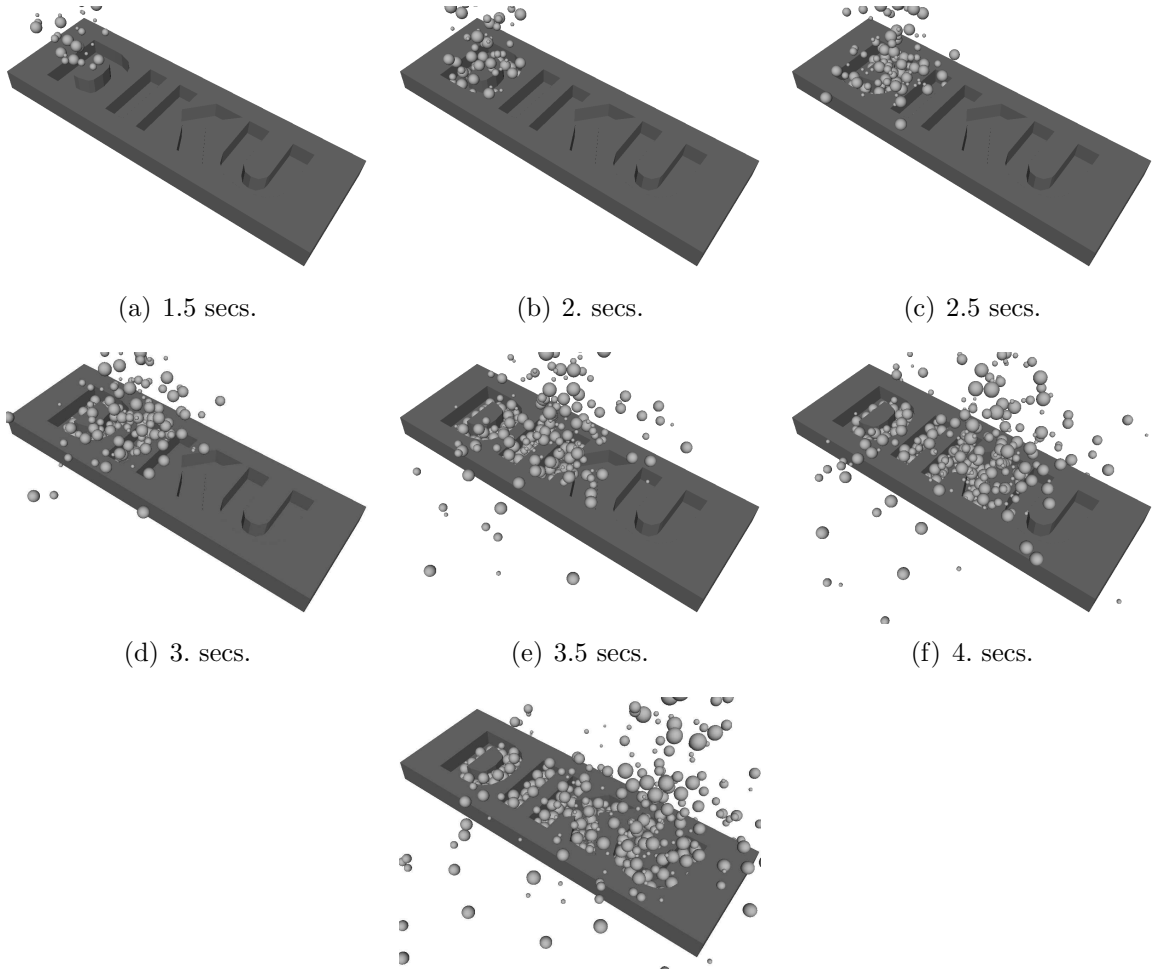
Figure 6.28: Sphere tree of vertex, edge, and face point re-sampling of box object.



(a) Depth 0. (b) Depth 1. (c) Depth 2.

(d) Depth 3. (e) Depth 4. (f) Depth 5.

Figure 6.29: Sphere tree of vertex, edge, and face point re-sampling of cow object.



(a) 1.5 secs. (b) 2. secs. (c) 2.5 secs.

(d) 3. secs. (e) 3.5 secs. (f) 4. secs.

(g) 4.5 secs.

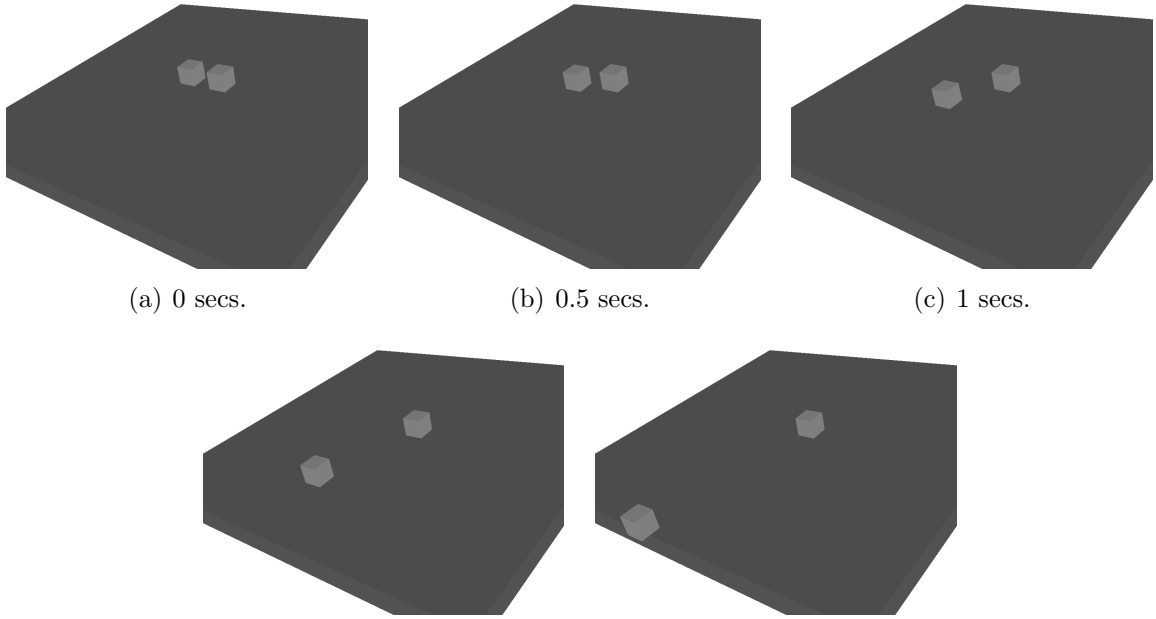
Figure 6.30: The sphere pruning test has been extended to a sphere-primitive collision detection test against the signed distance map.

the interpolated value is less than the collision envelope, a gradient can be found using finite differences and a contact point can be generated.

In fact, the pruning test can easily be extended to provide a sphere-primitive test against a signed distance map. We have pursued this idea. If the sphere-surface crosses the zero-level-set surface then the closest point on the zero-level-set surface of the signed distance map to the center of the sphere is found. This closest point is used as a contact point, and the gradient of the signed distance map at this point is used as contact normal. The penetration distance is simply estimated by how far the closest point lies inside the sphere. Figure 6.30 shows still frames from an animation, where this sphere-primitive testing was applied.

6.2.3 Results

To verify the usefulness of our point re-sampling and sphere-tree acceleration, we have performed a total of 18 simulations, 9 simulations using the sphere-tree and 9 using the



(a) 0 secs. (b) 0.5 secs. (c) 1 secs.

(d) 1.5 secs. (e) 2 secs.

Figure 6.31: Still-frames from a sliding boxes configuration using the brute force approach.

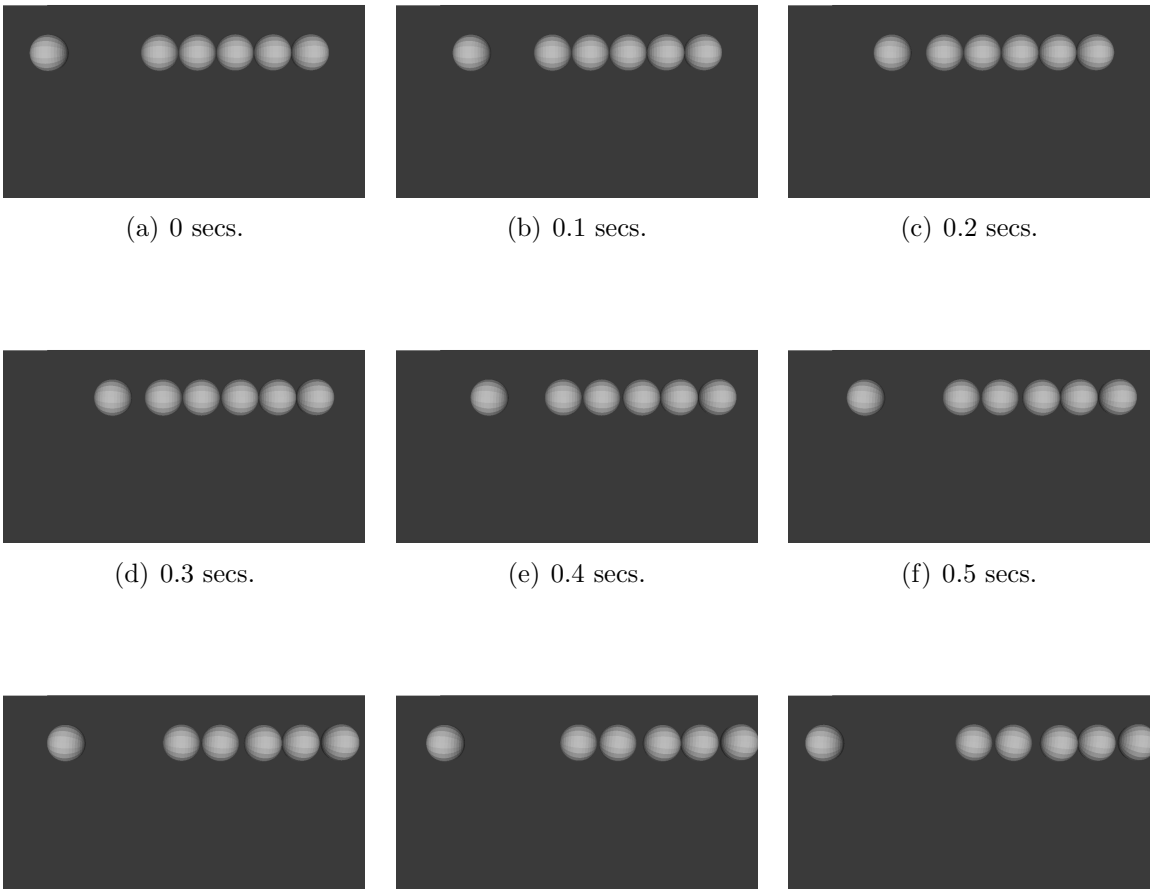
brute force approach of simply looking up every sample point in the other object's signed distance map. In all simulations we used the simulator described in Section 6.4 with a time-step of size 0.01 secs.

The first four of our test configurations was taken from [70]. The are the cases of sliding boxes, billiard balls, see-saw and flip-over. Simulation results of these four configurations can be seen in Figures 6.31-6.34. We then used a sensible point-sampled box stack and a completely oversampled box stack. The same was done for the large scale wall configuration. These configurations can be seen in Figure 6.35- 6.37. Our final test case consists of piling 250 cows on a plane as shown in Figure 6.38.

For all 18 simulations we counted the total number of contact points in each frame, and the size of the frame time, i.e. the total computation time for computing the next frame. Table 6.6 shows minimum, mean, and maximum values for the time measurements, and Table 6.7 shows minimum, mean, and maximum values for the contact point counts.

From Table 6.6 it can be seen that only the cow-pile configurations have a significant benefit from using sphere-trees. In the remaining configurations, sphere-trees appear to be comparable to the brute force approach, with some cases slightly better and others slightly worse. Figure 6.39 shows actual plots comparing frame times of the 9 test configurations. They clearly show what the statistics in Table 6.6 indicated.

Surprisingly, Table 6.7 shows that configurations using sphere-trees sometimes have a different contact count. Intuition would dictate that this indicates an implementation error in the sphere-trees. However, remember that sphere nodes in the sphere tree are pruned based on the distance computed at the sphere centers. A sphere center may have a very different location then the sample points contained in the sphere. Thus, the inherent error that a signed distance map poses causes some spheres to be pruned, because the



(a) 0 secs. (b) 0.1 secs. (c) 0.2 secs.

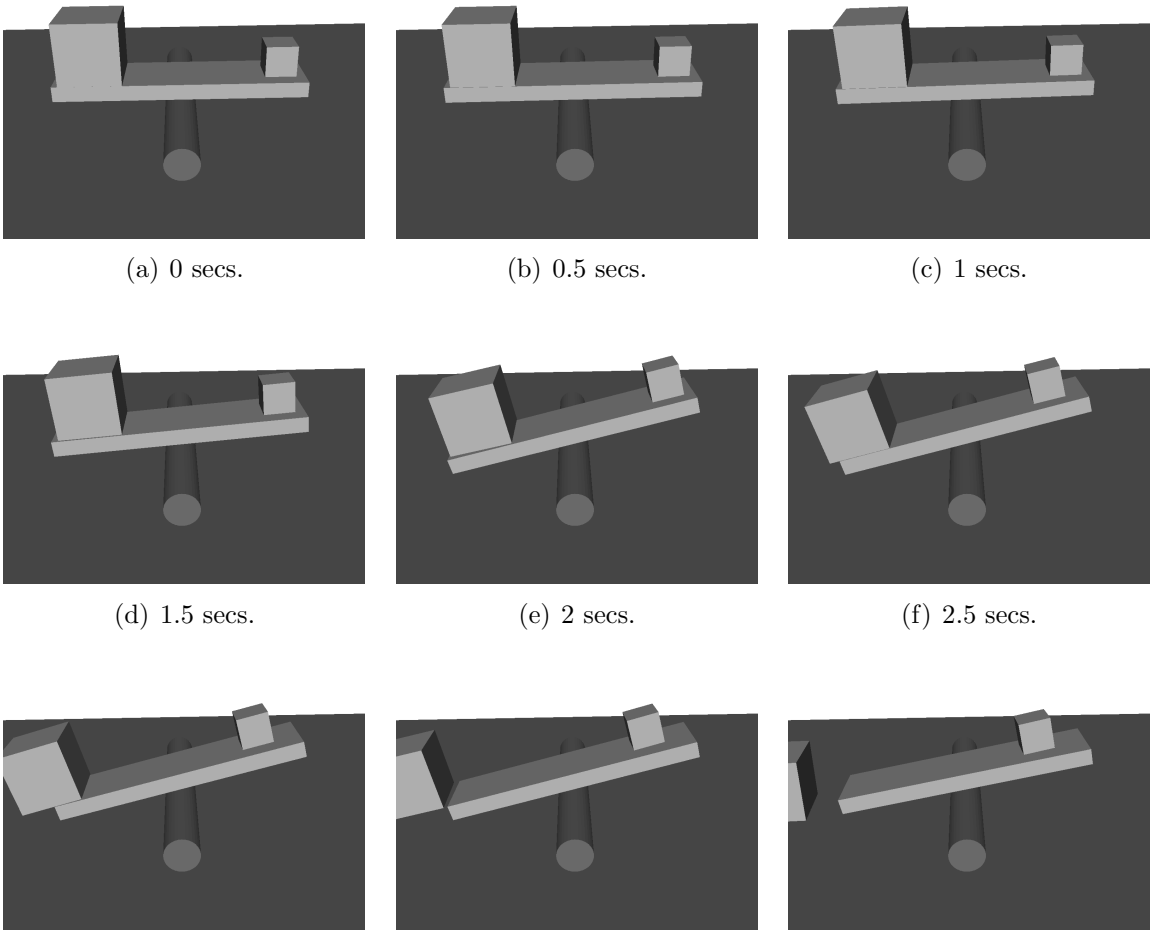
(d) 0.3 secs. (e) 0.4 secs. (f) 0.5 secs.

(g) 0.6 secs. (h) 0.7 secs. (i) 0.8 secs.

Figure 6.32: Still-frames from a billiard ball configuration using sphere-trees.

	brute			sphere-tree		
configuration	min	mean	max	min	mean	max
sliding boxes	0	0,0063	0,0200	0	0,0059	0,0200
billiard balls	0	0,0114	0,0300	0	0,0029	0,0200
see saw	0	0,0018	0,0100	0	0,0017	0,0200
flip over	0	0,0024	0,0200	0	0,0019	0,0200
box stack	0	0,0075	0,0200	0	0,0071	0,0200
oversampled box stack	0,0600	0,0738	0,0900	0,0600	0,0774	0,1000
wall	0,1400	0,1594	0,1910	0,1400	0,1608	0,1910
wall blow-up	0,0300	0,3785	0,8210	0,0300	0,3888	0,6400
cow pile	1,2420	9,0709	15,9930	0,0300	0,3899	0,7510

Table 6.6: Frame time statistics for different configurations using signed distance maps. A value of zero means that time duration were less than the timer resolution.



(a) 0 secs. (b) 0.5 secs. (c) 1 secs.

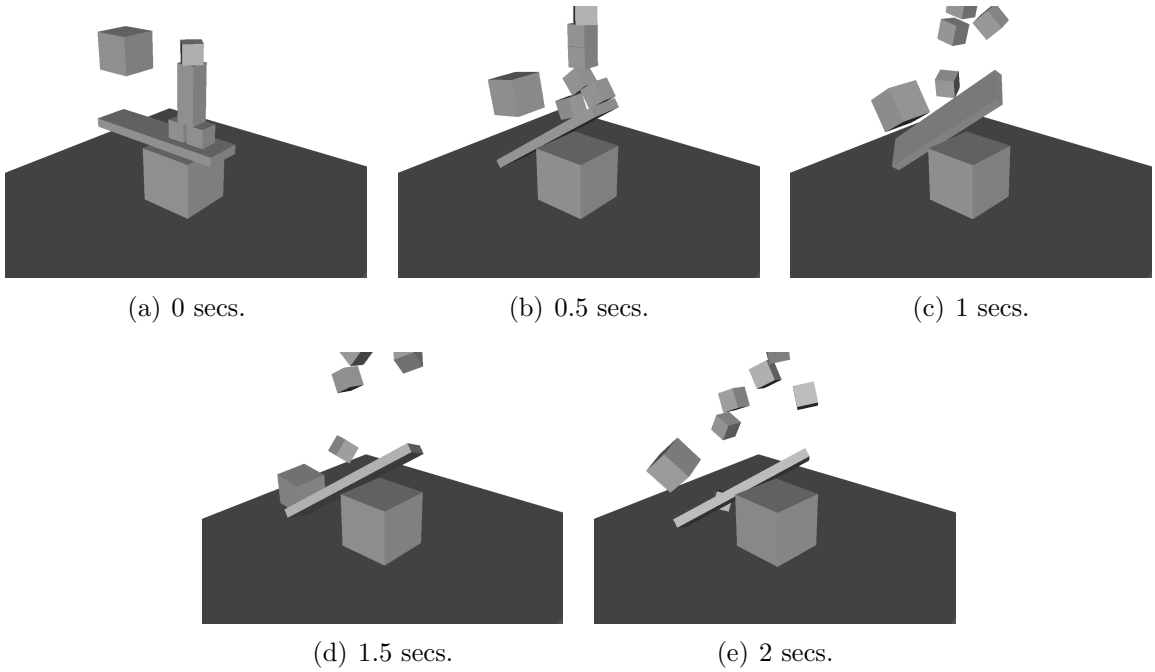
(d) 1.5 secs. (e) 2 secs. (f) 2.5 secs.

(g) 3 secs. (h) 3.5 secs. (i) 4 secs.

Figure 6.33: Still-frames from a see-saw configuration using sphere-trees.

	brute			sphere-tree		
configuration	min	mean	max	min	mean	max
sliding boxes	362	362	362	362	362	362
billiard balls	6	65,7	187	6	65,0	187
see saw	3	47,6	82	16	44,8	82
flip over	62	88,8	350	62	89,6	350
box stack	596	596	596	596	596	596
oversampled box stack	4517	4517	4517	4517	4517	4517
wall	7315	7410,9	7431	7303	7409,8	7431
wall blow-up	1811	16600	26416	1811	19735	27688
cow pile	862	5395	14835	784	5008	13391

Table 6.7: Statistics over the number of contacts for different configurations using signed distance maps.



(a) 0 secs. (b) 0.5 secs. (c) 1 secs.

(d) 1.5 secs. (e) 2 secs.

Figure 6.34: Still-frames from a flip-over configuration using sphere-trees.

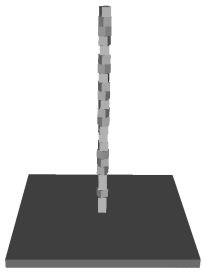


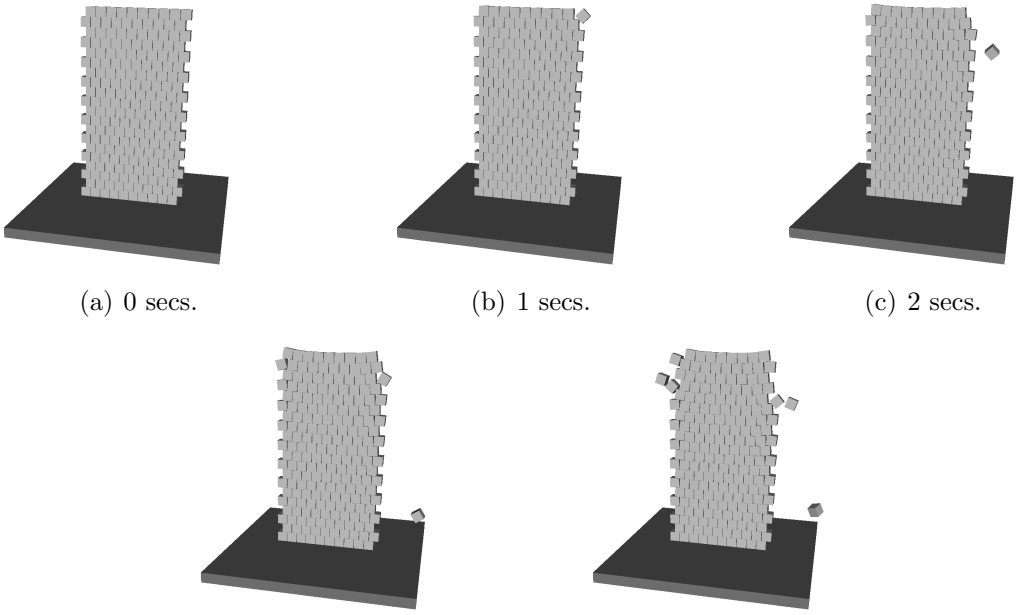
Figure 6.35: Still-frames from a box-stack configuration using sphere-trees. The boxes are not moving as the simulation progresses.

distance computed at the sphere center (due to interpolation error) says it, although the more accurate distances at the contained sample points would indicate that the sphere node should not have been pruned. The artifact becomes worse when on the coarse grids we have used for our signed distance maps, which typically have a dimension of $128 \times 128 \times 128$. Figure 6.40 shows plots comparing contact point counts. The figures clearly show plots with the same asymptotic behavior, but the plots are slightly different.

6.2.4 Discussion

It might very well be that spheres and the octree top-down splitting approach do not yield the best kind of BVH, in terms of pruning capability. However, it is simple to implement. Spheres are the perfect generalization of points and are therefore naturally combined to be used with signed distance map as shown in Figure 6.30.

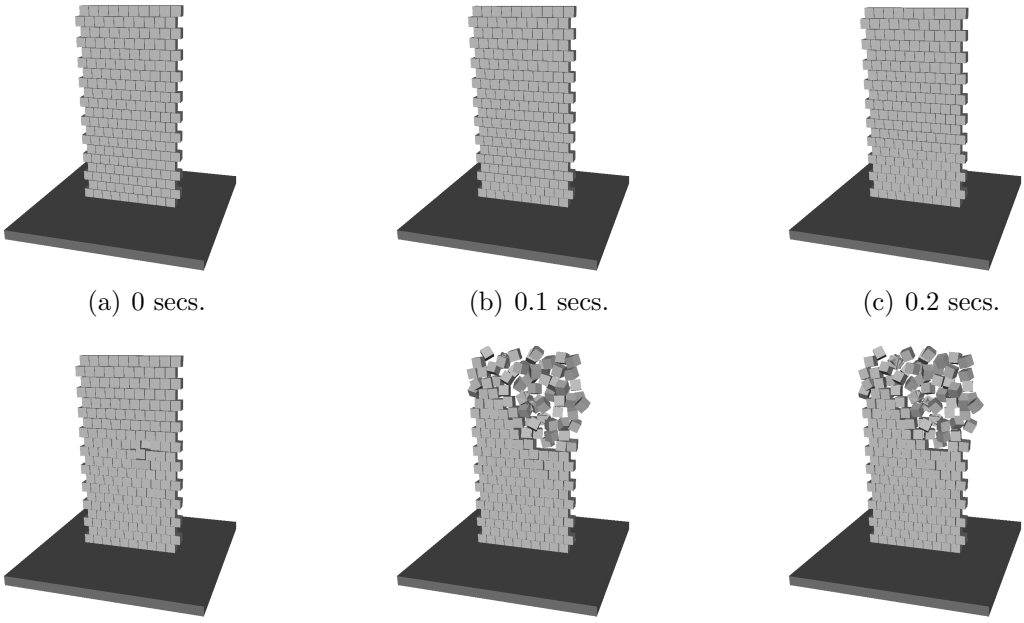
We have observed that using a sphere-tree BVH on signed distance maps gives a very



(a) 0 secs. (b) 1 secs. (c) 2 secs.

(d) 3 secs. (e) 4 secs.

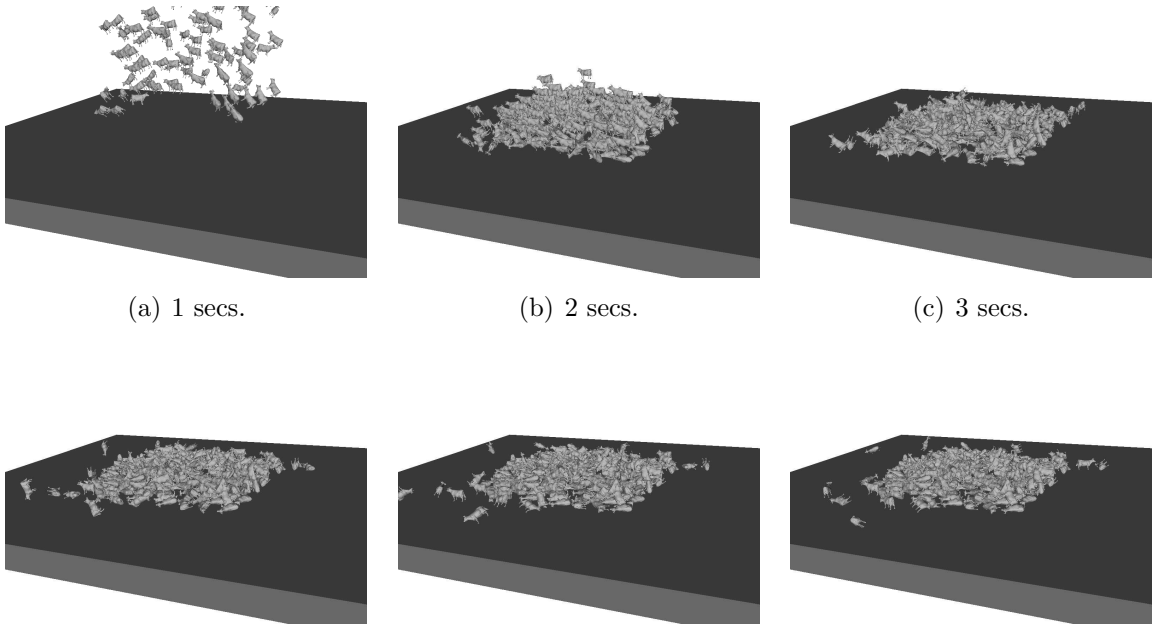
Figure 6.36: Still-frames from a wall configuration using the brute approach.



(a) 0 secs. (b) 0.1 secs. (c) 0.2 secs.

(d) 0.3 secs. (e) 0.4 secs. (f) 0.5 secs.

Figure 6.37: Still-frames from a wall configuration using the brute approach. Boxes are initially displaced by a small distance and during the explicit time-stepping the boxes will slightly penetrate. Due to the local property of the signed distance maps, unfortunate contact normals are generated pointing in a horizontal direction causing a blow-up.



(a) 1 secs. (b) 2 secs. (c) 3 secs.

(d) 4 secs. (e) 5 secs. (f) 6 secs.

Figure 6.38: Still-frames from cow pile configuration using sphere-trees.

small, almost insignificant, performance benefit in most cases. There are several reasons that causes this behavior.

In most of our test-cases, the objects are densely stacked, implying that objects are in close proximity. Thus, by the nature of the configuration, there is a lot of contact points. BVHs will thus traverse many paths in their hierarchy all the way down to the leaf nodes, in order to generate these contact points, implying that the performance gain of using BVHs is almost insignificant.

From theory [67], we obtain the following insight into the nature of the performance of using BVHs. For close proximities, BVH queries are likely to be $O(n \log n)$, since the number of contact points is of order n, and the depth of the tree is of order $\log n$, the brute force method is simply $O(n)$. The cost of a BV overlap test is less than looking up a sample point in the signed distance map. Thus, the BVH can beat the brute-force method if the number of contact points is sufficiently low, and/or enough spheres containing a good ratio of contact points can be pruned. If none of these properties are present, the brute force method is likely to outperform the BVH approach.

As the sampling density is increased, so is the performance gain of using BVHs as in the case of the cow pile shown in Figure 6.38. However, it is unlikely that interactive or real-time applications would wish to increase the sampling density, since a low sampling count is attractive due to the faster performance. On the other hand, off-line simulations such as the ones used in movie-production could clearly benefit from increasing the sampling density and applying sphere-trees.

The sampling density is also directly related to the accuracy of the collision detection. Theoretically speaking, if one lets the sampling resolution go to zero, the collision detection will be exact. For plausible simulations, a coarse sampling resolution is more than sufficient. In conclusion, implementing BVHs may be overkill for applications where

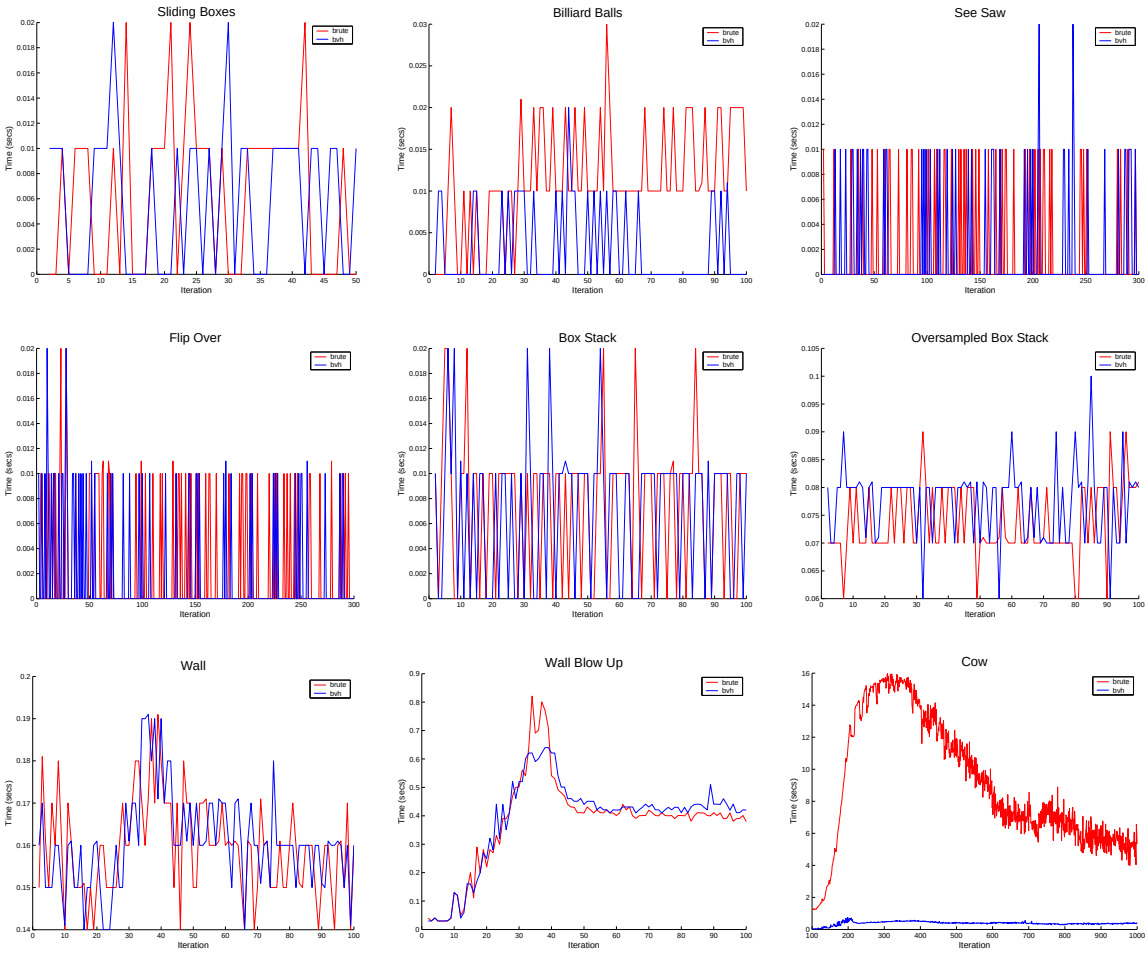


Figure 6.39: Frame time plots comparison.

inaccurate plausible results are sufficient. Here, a brute force method would be a better choice.

Besides, increasing the sampling density may not always be the best idea. If too many contact points are generated the system becomes widely over-determined, and the numerics can blow up the simulation. In our experience this may occur in densely stacked environments, such as the brick-wall example in Figure 6.37.

In summary we can conclude

- Signed distance maps take up a lot of memory,
- They generate a large number of contact points,
- They are robust, fault tolerant, and capable of handling severe penetrations,
- They are easily implemented,
- They provide a local solution to handling penetrating objects.

The first two statements make signed distance maps less attractive for real-time applications. The last two statements are attractive properties for real-time applications such as animation tools and computer games.

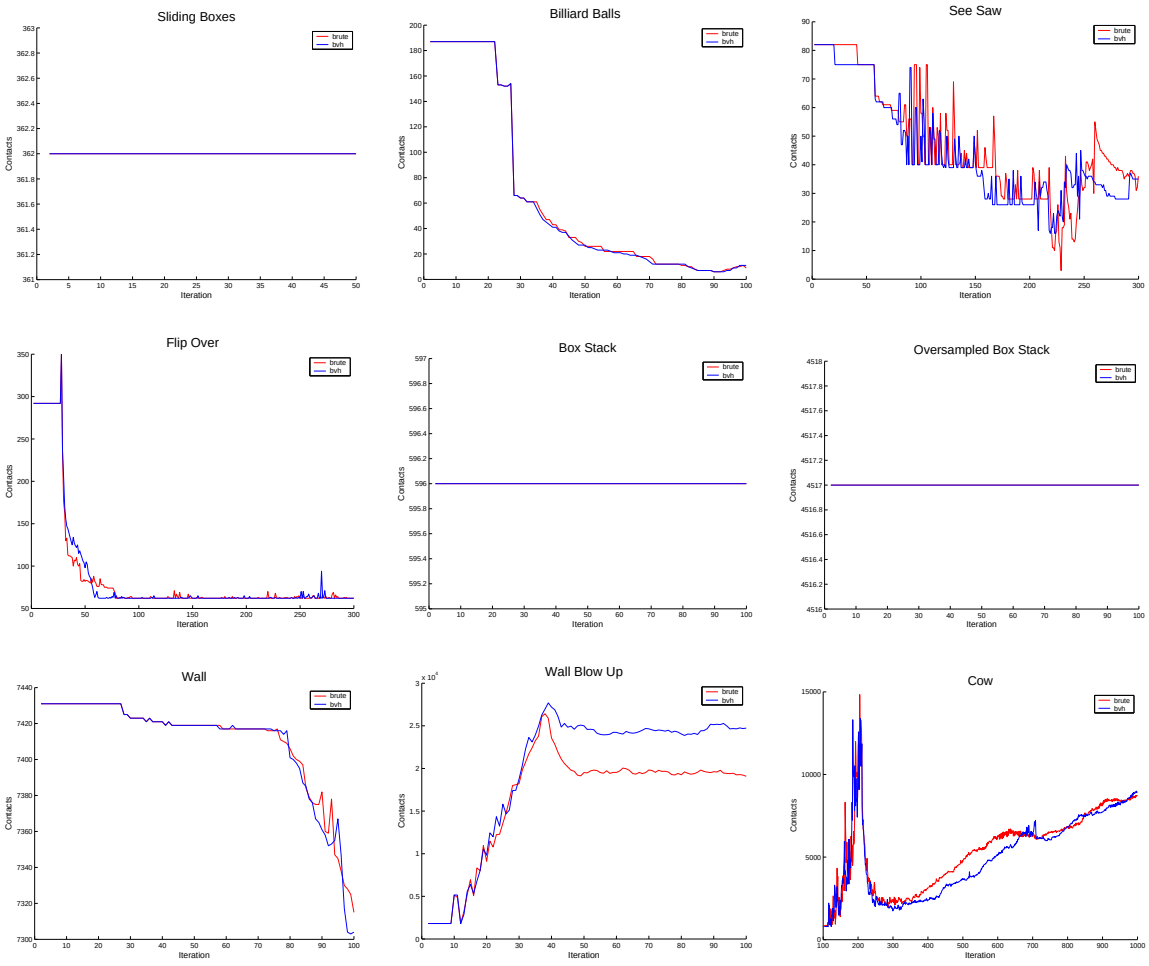


Figure 6.40: Contact point count plots comparison.

The fifth statement can cause severe artifacts in a simulation. For instance, if a smaller box is initially placed inside a larger box, However these cases are not very common. Thus, this is not prohibitive for their use.

6.3 Box Box Collision

We have shown previously how the simulation method can recover from faulty behavior in a stable and robust manner, as described in Section 4.16.1.1. However, it was implicitly assumed that the contact generation created ideal perfect contact points. The creation of these contact points is termed contact determination or contact generation [26, 19, 108]. It is a purely geometrical problem, but it is far from trivial or simple. In this work, we have tried to use signed distance maps, as described in Section 6.2, as an easy solution for contact generation, but they failed miserably on the following accounts:

- They are computationally expensive compared to primitive geometry testing, making them attractive for only highly complex geometries.
- They generate a huge number of contact points.

- The quality of contact normals and penetration depths depend on the resolution of the signed distance map. In fact this can cause a blow-up of the simulation (see Figure 6.27) where contact normals go from upward to sideward for two almost aligned boxes with over-sampled edges.

Box primitives are often the preferred choice in many respects, especially in real-time applications such as computer games. There are several reasons for this popularity. Box primitives or oriented bounding boxes (OBBs), as they are called in the literature, are complex enough geometries for providing fast convergence towards the surface of an object [67]. This is why they are often the preferred choice for bounding volume hierarchies (BVHs). This implies that relatively complex shapes can be approximated relatively easy with a low number of boxes. Yet, boxes are simple enough geometries to perform reasonable fast collision detection. As a consequence, basically all third-person shooting computer games on the market today allow the player to shoot or push boxes around in a submerged world.

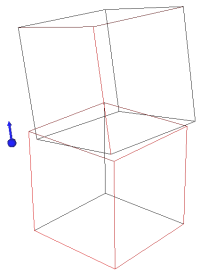
Following this path of thinking we will concentrate on good contact point generation for box-box primitives. We will show how the most widely used algorithm completely fails in certain cases and then we will repair these deficiencies to accommodate, what we think is a better alternative for contact generation.

In our opinion there are several aspects that may cause a simple minded contact generation algorithm to fail in the sense that the simulation yields an unwanted behavior, or a worse faulty state.

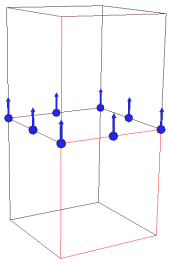
Firstly, simulation errors tend to sneak in to the positioning of the geometry causing small penetrations and misalignments of the geometries. Secondly, tunneling effects may even occur, resulting in geometries being deeply penetrating or even being teleported completely inside another geometry. Thirdly, there is the risk of an end user setting up initially badly placed geometries. Each of these three aspects are more the common case than the exception in everyday simulations. Thus, a usable contact generation algorithm must be stable and robust towards penetrations of geometries and imprecision in their positioning.

The most popular and widely used box-box overlap test is the separation axis test method [66]. It is currently known to be the fastest method in terms of the number of FLOPS required to determine the overlap state of two boxes. A contact generation approach has been developed, based on using the separation axis with the largest overlap, i.e. the axis along which the two boxes should be pushed the least in order to not penetrate any longer. To our knowledge, this approach does not seem to be well-described in the literature, but newsgroups [1] have touched the topic more than once. Also, some open source projects [112] offer implementations based on this idea. There are also examples of commercial software [111] that successfully deal with box-box testing. However, we can only guess at the algorithms and methods used herein.

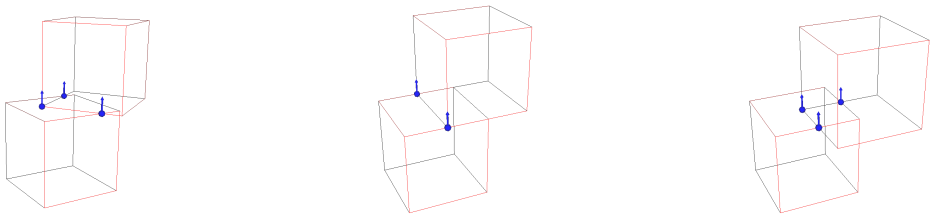
We will use the implementation in [112] as our working example, in order to study the problems with this approach. Henceforth, we refer to the specific implementation in [112] as the old box-box method, in order not to confuse it with our improvements. In [112] a single contact point is generated for edge-edge cases, using the mean point of the two closest points between the two lines running parallel with the two edges. For face-cases, the separation axis is equal to a face normal of one of the boxes, and the closest face of the other box is found and projected onto the box with the separation axis. A simple 2D



(a) Edge-edge case is picked instead of face-face case, generating a faulty contact point outside the contact region.



(b) Contact state of corners are ignored during 2D intersection testing, causing edge-edge crossings to generate contact points.



(c) Missing corners, due to badly (or too lazy) 2D intersection testing.

Figure 6.41: Contact generation using the old box-box test.

intersection test is now performed between the projected face and the face corresponding to the separation axis. Intersection points are reported as contact points.

There are mainly three problems with this approach. Firstly, in some cases an edge-edge case is picked when a face-case should have been used. Secondly, the method exhibits jittering of contact points, and thirdly, some cases are missing crucial contact points. Figure 6.41 supports these statements. Here we have applied the method from [112] and drawn the computed contact points.

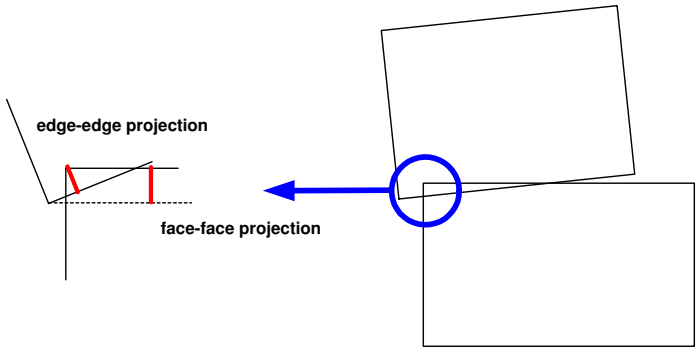


Figure 6.42: 2D projected view of two boxes illustrating when an edge-edge case is picked over face-case.

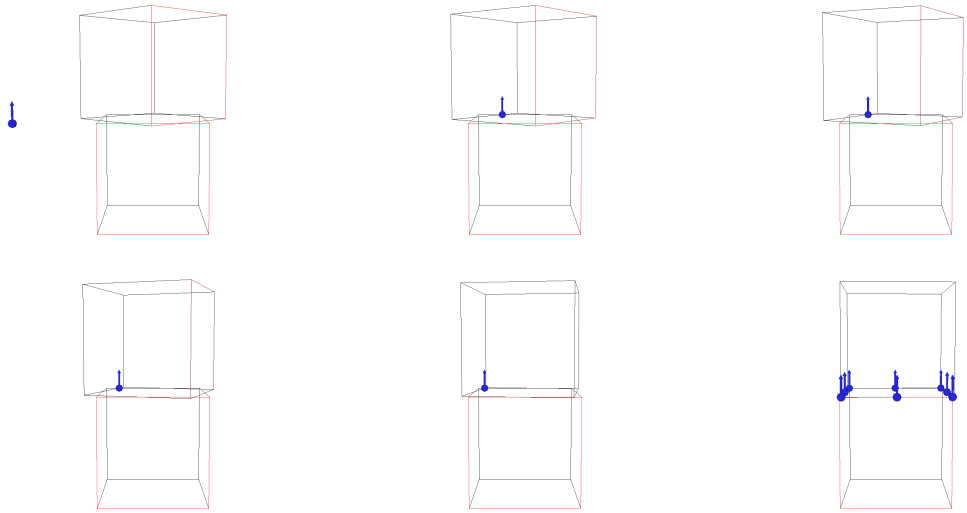


Figure 6.43: Animated box-box collision sequence using the old box-box method.

Figure 6.42 illustrates an analysis of the cause to the faulty edge-edge case. The top box in the figure is slightly tilted. This is shown in an exaggerated fashion. However, notice that the corner point outside the lower box causes the face-projection distance to become larger than the edge-edge projection distance. The scenario shown in the figure often occurs in practice due to small precision errors in computing the relative orientation of the two boxes. This causes the method to see boxes as being slightly tilted, and the edge-edge case wins the projection distance race against the face cases.

We repair the faulty edge-edge case by making sure to truncate precision errors in the rotation matrices of the two boxes. Thus, slight tilting due to imprecision is projected back into a face-case. Secondly, we require that all the four end-points of the edges in the edge-edge case lie outside the boxes. If not, the edge-edge case is dropped and the face-case with minimum overlap is used instead. Figure 6.43 and Figure 6.44 show a sequence of images from an animation sequence comparing the improvement with the old box-box method. As seen from the figures our extensions repair the faulty edge-edge case.

We have also extended the old box-box method to deal with the problems of missing corner contact points and edge-edge crossings generating superfluous contact points.

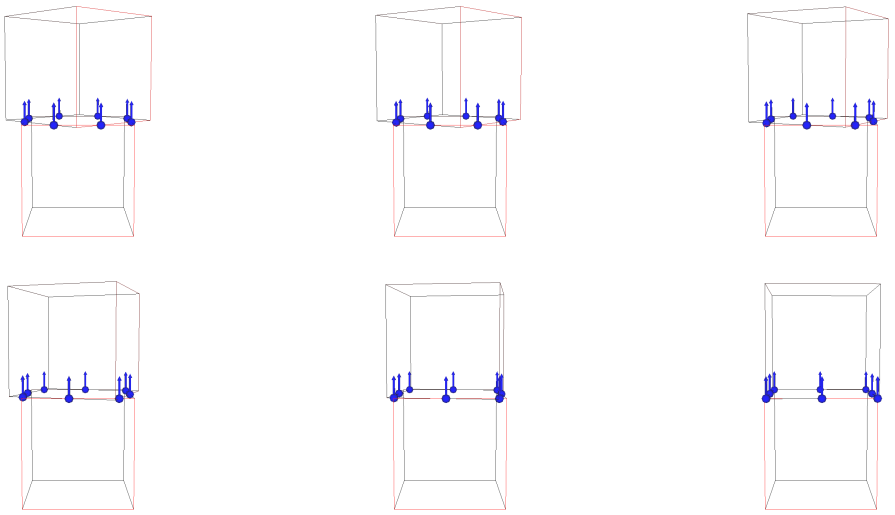


Figure 6.44: Animated box-box collision sequence using the improved box-box method.

These problems are illustrated in Figure 6.41(c) and Figure 6.41(b). The extension is simply to make sure that the most restrictive cases are handled before the more general ones. This means that we first detect whether any of the corners, four from each box, generates a contact point, and if so, the corner is flagged as a contact point. After having taken care of the corners we proceed to the edge-edge crossings. No contact points are generated for edge-edge intersection points if both end-points, i.e. the box corners, were previously flagged as contact points. Further more, it is required that the end-points of one edge must lie on opposite sides of the other edge.

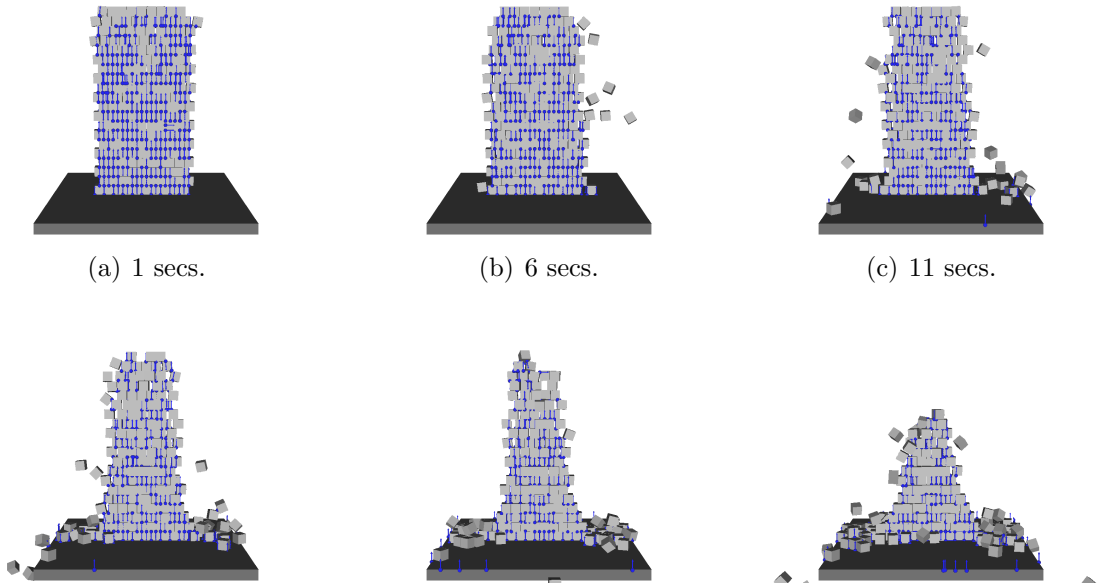
To verify the usefulness of the improved box-box method, a simulation comparison has been performed. A 30 second long simulation is done of a brick wall containing 200 brick cubes. The simulator from Section 6.4 was used with a time-step size of 0.01 seconds. Figure 6.45 shows still-frames from the simulation using the old box-box method, and Figure 6.46 shows corresponding still-frames from the same simulation using the improved box-box method.

The results obtained in Figure 6.45 are less than satisfactory, and obviously using the old box-box method in densely stacked configurations can seriously harm the quality of the simulation. In our opinion ideal and perfect contact points should be generated such that:

- If one slightly moves the geometries, contact points should follow at-least "piece-wise" continuous paths. These paths may merge or split during the movement, and we refer to this as non-flicking contact points.

- In a similar fashion the contact normals at the contact points should not change widely when the geometries are moved slightly. The contact normals should tend to be continuous. We refer to this as non-flicking contact normals.

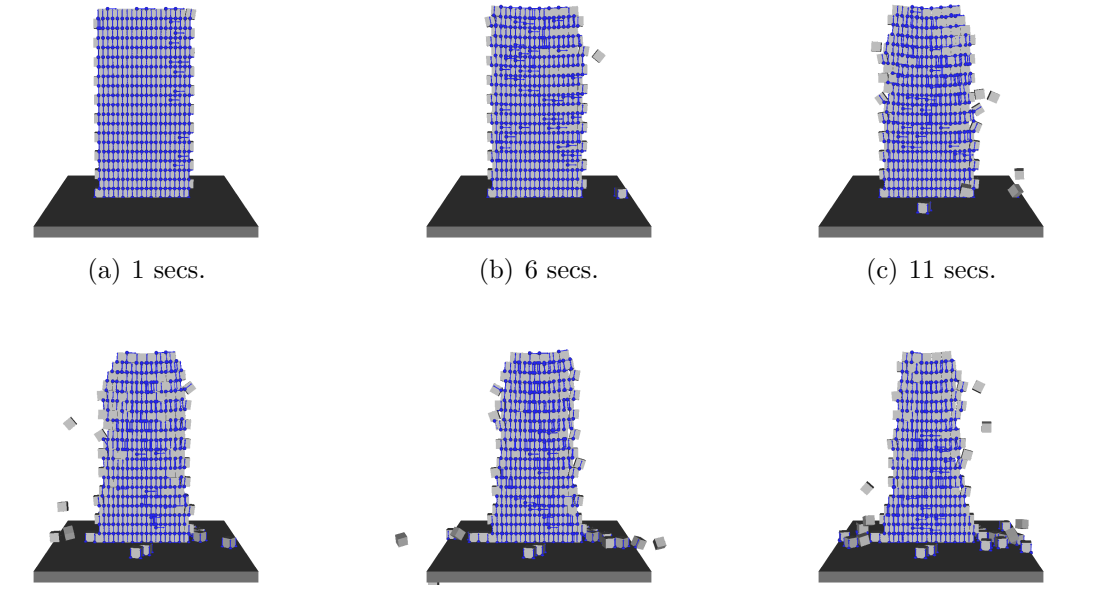
- Lastly, redundant contact points should be avoided, since they cause over-determined systems of constraints. This may cause the numerical methods used in a simulator



(a) 1 secs. (b) 6 secs. (c) 11 secs.

(d) 16 secs. (e) 21 secs. (f) 26 secs.

Figure 6.45: Wall simulation using the old box-box method. Blue arrows show contact points and contact normals.



(a) 1 secs. (b) 6 secs. (c) 11 secs.

(d) 16 secs. (e) 21 secs. (f) 26 secs.

Figure 6.46: Wall simulation using the improved box-box method. Blue arrows show contact points and contact normals.

to fail computing a valid solution for the constraint forces.

Clearly, the old box-box method suffers from both the flickering problems, causing boxes to jitter slightly during simulation. The effect is similar to what happens if one were subjecting the wall to a lot of small rapidly occurring earthquakes. Notice how the contact points drawn in Figure 6.46 are much more regular than in Figure 6.45. This is the major reason why the improved method works better.

The three requirements of contact point generation are based on our real-world intuition. For touching contact points in the real-world both the contact point and the contact normal would be continuous. Unfortunately, computer graphics geometries, boxes or polygons have discontinuities on the surfaces. At these discontinuous surface points, contact normal continuity is deemed to fail. Also, penetrations make it difficult to compute continuous contact points. Thus, in practice we must settle for less than the ideal.

In our opinion the entire subject of contact generation lacks attention and there is a lot to be done in this field. The subject is almost not described in existing literature. Works in computational geometry and collision detection tends to focus more on intersection testing than contact generation.

From our work it is clear that stable, robust and versatile contact generation is important for the quality of physical based animation. Thus, any new progress in this field is of great practical usage.

6.4 Velocity Based Complementarity Formulation with Shock-Propagation

For computer animation it is sufficient to use an explicit time-stepping scheme. Taking a first order Euler step of the equations of motion yields the time-stepping scheme:

$$\vec{b} = J\left(\vec{u}^{\,t} + h\mathbf{M}^{-1}\vec{F}_{\text{ext}}\right), \tag{6.75a}$$

$$\mathbf{A} = J\mathbf{M}^{-1}J^T, \tag{6.75b}$$

$$\vec{\lambda} = \mathbf{lcp}\ \left(\mathbf{A}, \vec{b}\right), \tag{6.75c}$$

$$\vec{u}^{\,t+1} = \vec{u}^{\,t} + \mathbf{M}^{-1}J^T\vec{\lambda} + h\mathbf{M}^{-1}\vec{F}_{\text{ext}}, \tag{6.75d}$$

$$\vec{s}^{\,t+1} = \vec{s}^{\,t} + h\vec{u}^{\,t}. \tag{6.75e}$$

Here, we have used the notation from Chapter 4 in a little abstract way since the generalized position vector, $\vec{s}$, is not of the same dimension as the generalized velocity vector, $\vec{u}$.

Unfortunately, this scheme is bad for taking large time-steps of stacked configurations. Let us study a small example to see why. Imagine two balls on a plane, all in touching contact but both balls have a downward velocity. When doing the velocity update with the classical Euler-scheme in equation (6.75), the position update will use the un-constrained velocities. Thus, in the next time-step the lower ball will penetrate the plane. Due to the large time-stepping, this error can be significant. If error-correction is used, the lower ball will be pushed up into the upper ball. The entire mess could have been avoided by using

a slightly modified version of the classical Euler-scheme with the simple difference that the constrained velocities are used during the position update. That is,

$$\vec{b} = J\left(\vec{u}^{\,t} + h\mathbf{M}^{-1}\vec{F}_{\text{ext}}\right), \tag{6.76a}$$

$$\mathbf{A} = J\mathbf{M}^{-1}J^T, \tag{6.76b}$$

$$\vec{\lambda} = \ \mathbf{lcp}\ \left(\mathbf{A}, \vec{b}\right), \tag{6.76c}$$

$$\vec{u}^{\,t+1} = \vec{u}^{\,t} + \mathbf{M}^{-1}J^T\vec{\lambda} + h\mathbf{M}^{-1}\vec{F}_{\text{ext}}, \tag{6.76d}$$

$$\vec{s}^{\,t+1} = \vec{s}^{\,t} + h\vec{u}^{\,t+1}. \tag{6.76e}$$

The time-stepping scheme in equation (6.76) was successfully applied to all the simulations shown in Figures 6.12-6.23.

The major problem with the simulations in Section 6.1 was that the iterative method could converge too slowly, introducing significant simulation errors into the simulation. To counter these simulation errors, error-correction by projection was applied, as described in Section 4.16. The results showed a clear improvement, but the iterative method still required several 100 iterations to yield visual acceptable results. In this section we will try to do even better by adopting shock-propagation [70] to a velocity based complementarity formulation.

6.4.1 Review of Shock-Propagation

The novelty of the work in [70] is to split the numerical integration of the equation of motion into two separate phases causing a separation of the collision resolving from the contact handling. That is

- Collision resolving.
- Advance the velocities by doing a velocity update.
- Contact handling.
- Advance the positions by doing a position update.

In [70] a different approach is taken to collision resolving. A fixed number of iterations is performed over all the contact points, and the same is done for contact handling. Thus there might still be colliding contacts after the contact handling. To correct the errors, shock-propagation is applied before doing the position update.

In order to perform the shock-propagation, a contact graph is built and contact points are processed in a order corresponding to their placement in the contact graph. The contact graph is used to analyze if objects are stacked on top of each other. Afterwards, contact points are organized into disjoint sets representing the stack layers in a stack. These layers are processed in a bottom-to-top fashion setting lower objects in a layer to be fixed. That is, the lower objects will act as though they have infinite mass and become un-movable by the upper objects in the layer.

An alternative method for computing stack layers is presented in Section 6.4.2 which can be used with the contact graph data structure described in Chapter 5. The contact

```
algorithm shock-propagation(algorithm A)
  compute contact graph
  for each stack layer in bottom up order
    fixate bottom-most objects of layer
    apply algorithm A to layer
    un-fixate bottom-most objects of layer
  next layer
end algorithm
```

Figure 6.47: Pseudo-code version of the general shock-propagation algorithm.

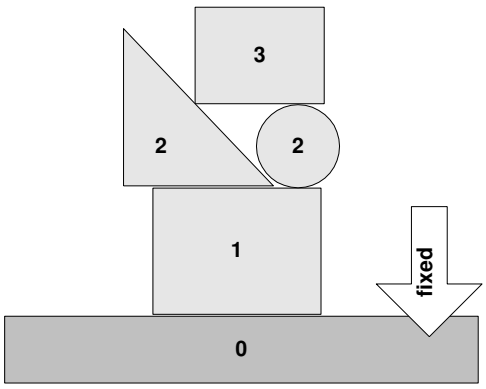


Figure 6.48: Simple stacked objects annotated with stack height.

graph built in [70] is computed differently and the interested reader should look in the reference for more details.

Figure 6.47 shows a pseudo-code version of our generalized shock-propagation algorithm, which we will apply in Section 6.4.3.

6.4.2 Computing Stack Layers

Contact graphs are easily computed as described in Chapter 5. A contact group is a subset of objects in the configuration, all in mutual contact with each other. Edges are created between objects if they are in contact, and contact points are stored directly in these edges. Thus, we want to analyze a contact group for its stack structure and if possible compute stack layers of the contact group.

A stack is defined as a set of objects being supported by one or more fixed objects. A cup on top of a table is in a sense a stack. The table is the fixed body and the cup is being supported by the table. Objects in a stack can be assigned a number indicating how far away they are from the fixed object supporting them. This number is an indication of the height of the object in the stack. Thus, all fixed objects in a configuration have a stack height of zero. Non-fixed objects in direct contact with the fixed objects have a stack height of one. Non-fixed objects in direct contact with objects with stack height one, but not in contact with any fixed objects have a stack height of two. A simple example is shown in Figure 6.48. This definition of stack height does not give a unique sense of an up and down direction as is commonly known from the real world. This is illustrated in

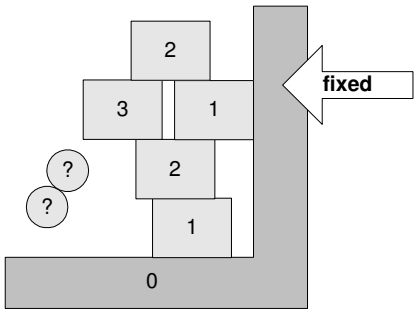


Figure 6.49: Non simple stacked objects annotated with stack height. Free floating objects are marked with a question mark.

```
Queue Q
for each body in Group do
  if body is fixed then
    height(body)=0
    Q.push(body)
    visit(body) = true
  else
    height(body) = infinity
    visit(body) = false
next body
```

Figure 6.50: Initialization of stack analysis algorithm. All bodies in a contact group are traversed, fixed bodies are identified and are then added to a queue for further processing.

Figure 6.49. Notice the position of the object with largest stack height. An object is said to be closer to the bottom of the stack compared to another object if the stack height of the object is lower than the other object. Similarly, the bottom-most objects are those having the lowest stack height. These are the fixed objects.

A free floating object is special, since it is not in contact with any other objects. However, one may even have an entire group of bodies, all in mutual contact with each other, but none in contact with a fixed object. In these cases it does not make sense to talk about assigning a stack height to the objects. Instead, the convention can be used to assign these kind of objects an infinite stack height to distinguish them from objects that are part of a stack. A negative value could also be used, but is not an efficient choice for the algorithm presented in this section.

The stack height of objects is easily computed by doing a breadth-first-traversal on each contact group. Initially, the stack height of all objects is set to infinity unless they are fixed objects, in which case their stack height is set to zero. Also, all fixed objects are pushed onto a queue. This queue will be used by the breadth-first-traversal. The initialization steps are shown in Figure 6.50.

After the initialization, the breadth-first-traversal will pop an object, A, from the queue and iterate over all incident contact graph edges to object A. For each edge, it is tested if the object, B, at the other end of the edge has been visited by the traversal before. If not, this object is pushed onto the queue. The height h_B of the object B is also

computed as

$$h_B = \min(h_B, h_A + 1). \tag{6.77}$$

That is, either a shorter path to object B is already known, in which case h_B is left unchanged, or it is shorter to get to B, by going from A. The cost of taking this part is one more than the cost of getting to object A.

During the traversal, a stack layer index is computed for the edges as they are being visited. A stack layer is defined by two succeeding stack heights, such as 0 and 1. These two stack heights define a subset of objects in the contact group. That is, stack layer 0 is defined as all objects with stack height 0 and stack height 1, and all edges between these objects. Stack layer 1 is defined by all objects with stack height 1 and stack height 2, and all edges between these objects, and so on. This means that an edge between an object with height i and another object with height $i+1$, is given stack layer index i.

Notice, there is some subtlety with edges between objects with the same object height. As an example if we for stack layer i have an edge between two objects both with stack height $i+1$, then the edge belongs to stack layer i. Ideally, an edge between two objects with height i should also be added to stack layer i. However, this is not done. The reason for this is that stack layers are processed in a bottom up fashion. Thus, contact points belonging to the edge between objects at height i have been taken care of when stack layer $i-1$ was processed.

Figure 6.51 shows pseudo-code for assigning stack heights to objects and stack layer indices to edges.

After having assigned stack heights to objects and stack layer indices to edges, it is a simple matter to traverse the edges and assign them to their respective layers they belong to.

Objects are a little special. Given an object, A, at stack height i, one must traverse the edges and examine the stack heights of the objects at the other end. If an object B with stack height $i-1$ is found, then object A is safely added to stack layer $i-1$. If an object C is found with stack height $i+1$ then object A is added to stack layer i. Object A can only belong to stack layer $i-1$ and i. This means that as soon as two other objects have been found, indicating that object A should be in these two stack layers, one can stop traversing the remaining incident edges of object A.

Figure 6.52 shows the pseudo-code for building the stack layers.

6.4.3 Adopting Shock-Propagation

The initial intention of shock propagation is to fix simulation errors. We can thus apply an algorithm such as the projection error correction explained in Section 4.12 and Section 4.16 to each stack layer in a bottom-up fashion. This has the advantage of being able to completely fix penetration errors. In comparison with equation (6.76) the time-stepping of the error-correction can be written as

$$\mathbf{A} = J\mathbf{M}^{-1}J^T, \tag{6.78a}$$

$$\vec{\lambda} = \mathbf{lcp}\left(\mathbf{A}, \vec{d}_{\text{penetration}}\right), \tag{6.78b}$$

$$\vec{s}^{\,t+1} = \vec{s}^{\,t} + \mathbf{M}^{-1}J^T\vec{\lambda}, \tag{6.78c}$$

```
List edges
timestamp = timestamp + 1
height = 0;
while Q not empty
  A = pop(Q)
  for each edge on A do
    B = body on edge that is not A
    if not visit(B) then
      Q.push(B)
      visit(B) = true
    end if
    height(B) = min( height(B), height(A) + 1 )
    if height(B) = height(A) and height(B) not 0 then
      layer(edge) = height(B) -1
   else
      layer(edge) = min( height(B), height(A) )
   height = max(height,layer(edge))
   if not timestamp(edge) = timestamp then
     timestamp(edge) = timestamp
     edges.push(edge)
   end if
  next edge
end while
```

Figure 6.51: A breadth-first-traversal is performed assigning a stack height to each body equal to the number of edges on the minimum path to any fixed body. Edges of the contact group are collected into a list for further processing.

where $\vec{d}_{\text{penetration}}$ is a vector of penetration depths. The notation is still a little abstract, since J is not the same as the one used in equation (6.76). It only contains normal constraints at the contact points.

Letting **error-correction** () denote a time-step by equations (6.78), then Figure 6.53 shows results of a simulation using **error-correction** () and Figure 6.54 shows simulation results using **shock-propagation** (**error-correction** ()). In both figures an iterative Gauss-Seidel LCP solver was applied using only 5 iterations.

Comparing the results from Figure 6.53 and Figure 6.54 it is clear that shock-propagation yields superior results. In fact, it can be seen that only 5 iterations is enough for the iterative LCP solver, when using shock-propagation. Without shock-propagation the error-correction does not even fix all penetration errors within the same number of frames. This is not shown in Figure 6.53. However, it is seen that the final penetration free positions will not even result in a nice stack grid of balls.

Figure 6.54 shows that shock-propagation completely corrects all penetration errors in the first layer during the first frame-computation. In the second frame all penetration errors in the second layer are corrected and so on. For the specific example used, it thus takes 5 frame computations to completely correct all penetrations errors. This property is due to the way our collision detection engine interacts with the shock-propagation algorithm. If, during the shock-propagation, the collision detection engine could re-evaluate all penetration depths of the contact points in a stack layer prior to applying the error-correction

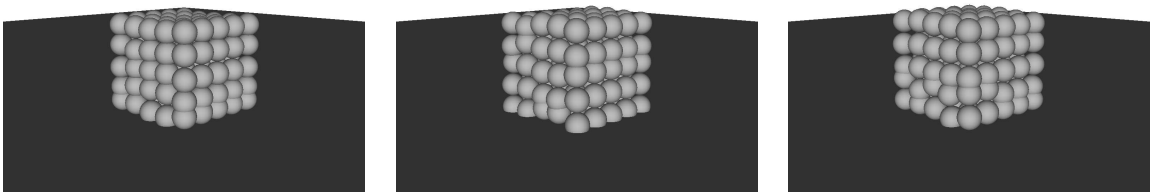
```
Group layers[height +1]
for each edge in edges do
   idx  = layer(edge)
   add contacts(edge) to layers(idx)
next edge

for each body A in Group do
  if height(A)=infinity then
    continue
  end if
  in_lower = false
  in_upper = false
  for each edge on A do
    B = other body on edge
    if height(B) > height(A) then
      in_upper = true
    end if
    if height(B) < height(A) then
      in_lower = true
    end if
    if in_upper and in_lower then
      break
  next edge
  if in_upper then
    layers[height(A)].push(body)
  end if
  if in_lower then
    layers[height(A) - 1].push(body)
  end if
next body
return layers
```

Figure 6.52: Building stack layers by processing all edges and bodies by examining their stack height and layer indices.

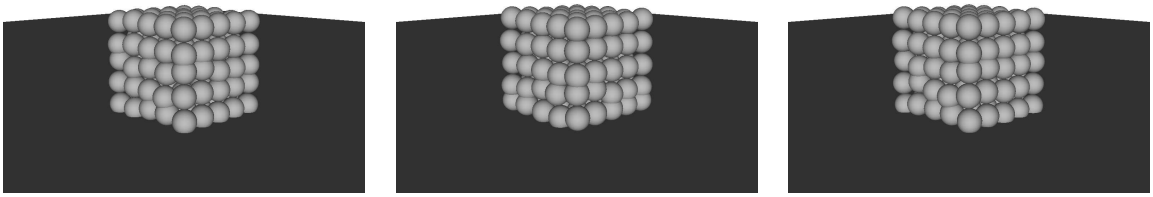
algorithm to the stack layer, then all penetration errors would have been corrected in the first frame computation. This specific interaction with the collision detection engine can cause spurious rippling effects during simulation as discussed in Section 6.4.5.

It should be noted that the shock-propagation that occurs from the bottom most layer of a stack to the top, has trouble fixing errors for "cyclic" configurations as illustrated in Figure 6.55. In fact, the simulation done in Figure 6.30 suffers from the exact problem shown in Figure 6.55. However, it is difficult to see from the still images in Figure 6.30. However, the problem is seen in an animation as high-frequency oscillating spheres lying just below the top-most spheres.

This is clearly an unwanted side-effect, and definitely an area of further work. We suggest that one should take the direction of gravity or whatever external force that is acting on the objects into account when computing stack-heights. Thus, when a down-hill edge is seen going to an object with larger stack-height, the stack height of the up-hill object should be one higher than the down-hill object, regardless of whether the up-hill object is in direct contact with a fixed object. This idea would result in the balls on the

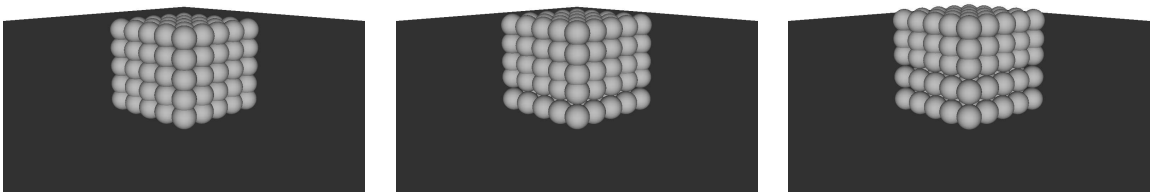


(a) frame 1. (b) frame 2. (c) frame 3.

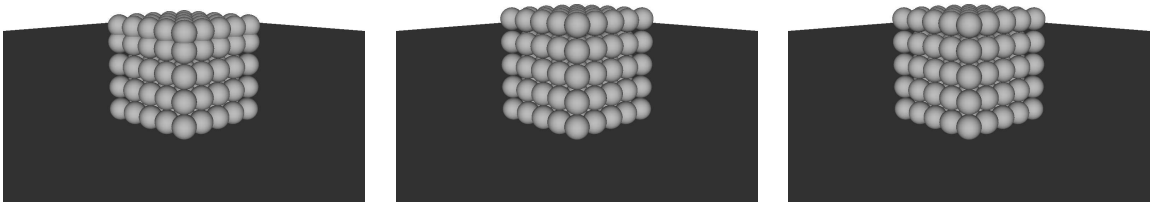


(d) frame 4. (e) frame 5. (f) frame 6.

Figure 6.53: A grid stack of 125 balls with severe penetrations using 5 iterations with Gauss Seidel and no shock-propagation.

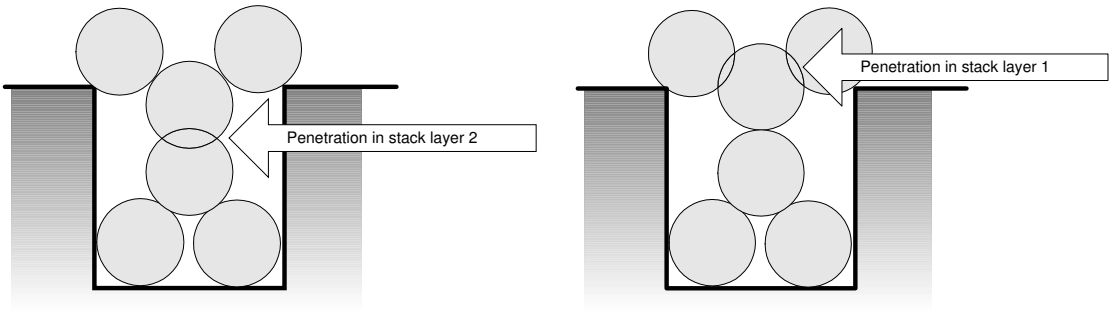


(a) frame 1. (b) frame 2. (c) frame 3.



(d) frame 4. (e) frame 5. (f) frame 6.

Figure 6.54: A grid stack of 125 balls with severe penetrations using 5 iterations with Gauss Seidel and shock-propagation.



(a) Shock propagation results in Figure 6.55(b) (b) Shock propagation results in Figure 6.55(a)

Figure 6.55: Errors can not be corrected by shock propagation if there are cyclic dependencies.

```
velocity-based-shock-propagation(f,h)
  collision detection at time t
  dynamics(f*h)
  shock-propagration( dynamics( (1-f)*h ), error-correction() )
  t = t + h
```

Figure 6.56: Pseudo-code of the velocity based shock-propagation algorithm. f denotes the weighting of the dynamics vs. the shock-propagation.

ridge of the canyon in Figure 6.55 to have a stack height four and not one. However, it is not exactly obvious how to perform the analysis we just described in a fool-proof way, which is why we leave the idea for future work.

The dynamics of objects described by equations (6.76) can also be used in combination with shock-propagation. Letting a time-step taken by equations (6.76) be denoted by **dynamics** (h), where h is the size of the time-step, we then propose the time-stepping scheme shown in Figure 6.56.

We have introduced a weighting of the dynamics versus the shock-propagation, where the weighting is given by f, where $0 \leq f \leq 1$. The weighting is not some wild idea, but was necessitated through experiments which indicated that the weighting was directly related to the amount of allowed simulation error. The exact relation between the weighting factor and simulation error is discussed in detail in Section 6.4.5 and Section 6.5. In comparison with the new time-integration method in [70] which we have restated in Section 6.4.1 for convenience, our scheme is similar to the new time-integration method if f is set to 1.

The scheme in Figure 6.56 has been applied to all the simulations shown in Figure 3.7, 3.8, 3.9, 3.10, 6.30, 6.31, 6.32, 6.33, 6.34, 6.35, 6.36, 6.38, 6.46, 6.67, 6.68, 6.70, 6.71, and Figure 6.72 using only 10 iterations for the iterative LCP solver solving the dynamics and 5 iterations for the LCP solver solving the error correction. In most cases we have used f-values in the range $0 \leq f \leq 0.01$.

6.4.4 Weight Feeling Problem

Shock-propagation was used for an impulse based method in [70]. An important step of the time-integration method herein was to let objects feel the weight of other objects before applying the shock-propagation algorithm. The first time-step **dynamics** (fh) in our scheme is responsible for this weight-feeling aspect. In general there are two cases for when objects need to feel the weight of each other. There is the case of an object impacting with a stack and then there is the case of a system not in equilibrium, but where all objects are initially in resting contact. The see-saw in Figure 6.33 is a perfect example of the latter.

Our time-stepping scheme is capable of handling the weight-feeling of impacting objects regardless of the value of f. In practice, the ideal choice to counter simulation errors is to set f equal to zero. In this case, the first time-step reduces to a simple simultaneous resolving of initial impacts using Newton's collision law. The shock-propagation works most efficiently under these conditions, immediately eliminating any simulation errors from the first zero-size time-step.

Unfortunately, the case of systems initially not in equilibrium can not feel the weight, since there is no impacting objects. Increasing the f-value slightly takes care of the problem, but this comes with a cost, with the shock-propagation becoming less efficient in eliminating simulation errors. Although large-scale simulations can be performed successfully and without noticeable artifacts, the small simulation errors that are not eliminated seems to propagate through the stack severely damaging the effectiveness of a sleepy policy. This is described in detail in Section 6.5.

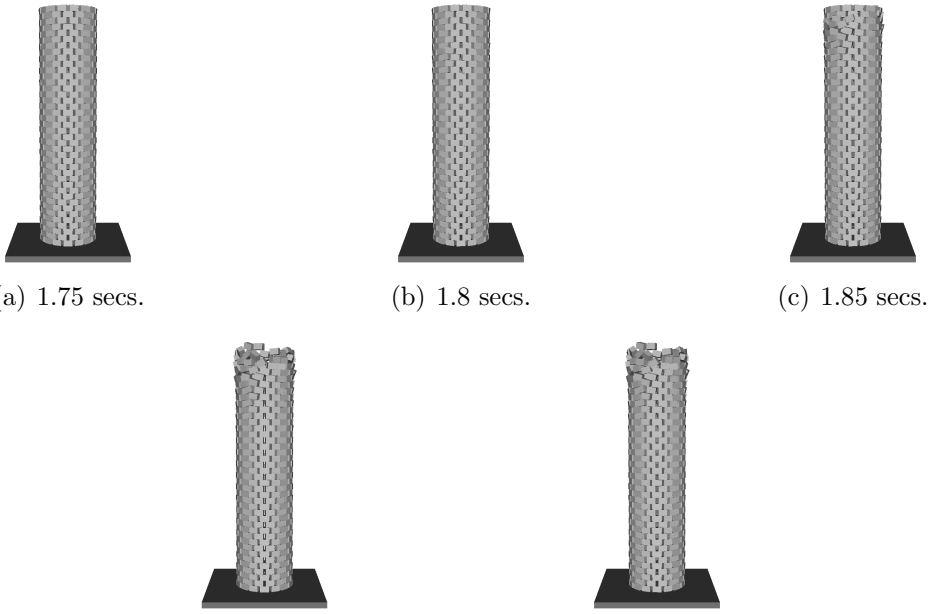
Theoretically, as we let the number of iterations used in the iterative LCP solver go to infinity, or if we exchange the iterative LCP solver with a direct solver, then the simulation errors will diminish, reducing the need for shock-propagation. When we let the weighting factor go towards one, we allow the simulation errors from the iterative LCP solver in the first time-step to gain the upper hand over the shock-propagation. Of course, we achieve the goal of being able to handle configurations initially not in equilibrium.

To counter the artifacts of the more dominant simulation errors we could increase the number of allowed iterations in the iterative LCP solver. We have not gone down this alley, because we wanted to stay in the realm of high performance time-stepping. Thus, performance requirements prohibit us from wasting computation time on doing too many iterations in the iterative LCP solver. We have thus set 10 iterations as the maximum limit for the dynamics, and 5 iterations as the maximum limit for the error correction. As can be seen from our results in Figure 3.7, 3.8, 3.9, 3.10, 6.30, 6.31, 6.32, 6.33, 6.34, 6.35, 6.36, 6.38, 6.46, 6.67, 6.68, 6.70, 6.71, and Figure 6.72 these aggressive limits are sufficient for handling rather complex large scale simulations.

Another avenue for possible future work would be to apply multi-grid methods for solving the LCPs. This may be advantageous for configurations with 10,000-100,000 objects or more.

6.4.5 Rippling Effect

During our simulation test, a rippling shock effect has been observed. At first sight, the effect looks similar to a blow up in the simulation due to some energy that has been built



(a) 1.75 secs. (b) 1.8 secs. (c) 1.85 secs.

(d) 1.9 secs. (e) 1.95 secs.

Figure 6.57: Rippling effect seen in a 640 brick tower simulation using $f = 0.025$.

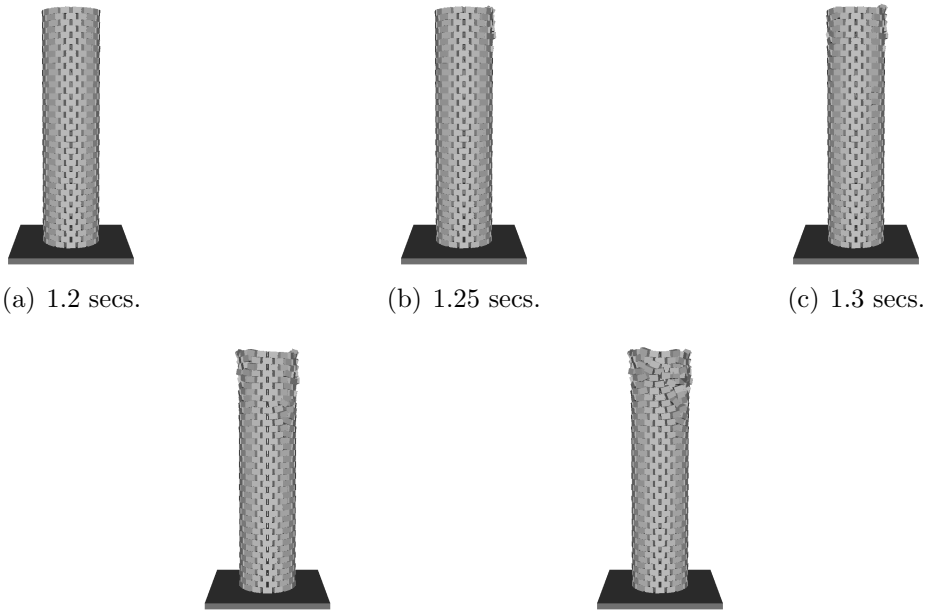
up. The rippling effect is related to the value of the weighting factor f. Figure 6.57 and Figure 6.58 illustrate the rippling effect as seen with different weighting values.

The figures show that for large weight values, the rippling effect appears to start earlier in the simulation and appears more violent. In fact, setting $f = 0$ causes the rippling effect to disappear totally. The cause of this kind of rippling shown in Figure 6.57 and Figure 6.58 is due to numerical errors from the iterative LCP solver. The error is built up during simulation, at the point where the error-correction kicks in. However, due to the specific way our collision detection engine interacts with the shock-propagation algorithm, the errors become worse as they are projected upwards through the tower. This is shown in Figure 6.59. Notice how the penetration depths increase in succeeding frames.

The rippling effect just explained is equivalent to the behavior of the error-correction, and a possible solution to the problem would be to re-evaluate contact points during the shock-propagation. This will prevent penetration errors accumulating. Unfortunately, our current collision detection engine is not easily changed to support this functionality. Instead, we have compromised with performance and added an extra error-correction step. This means that the pseudo-code in Figure 6.56 is changed into the version shown in Figure 6.60.

The final error-correction done in Figure 6.60 has the benefit of quickly distributing the penetration error to nearby objects. The drawback of this is the performance penalty in doing a second collision detection query per time-step. The range of objects that the penetration error is distributed to depends on the number of iterations used in the iterative LCP solver. In our simulations we have only used 5 iterations.

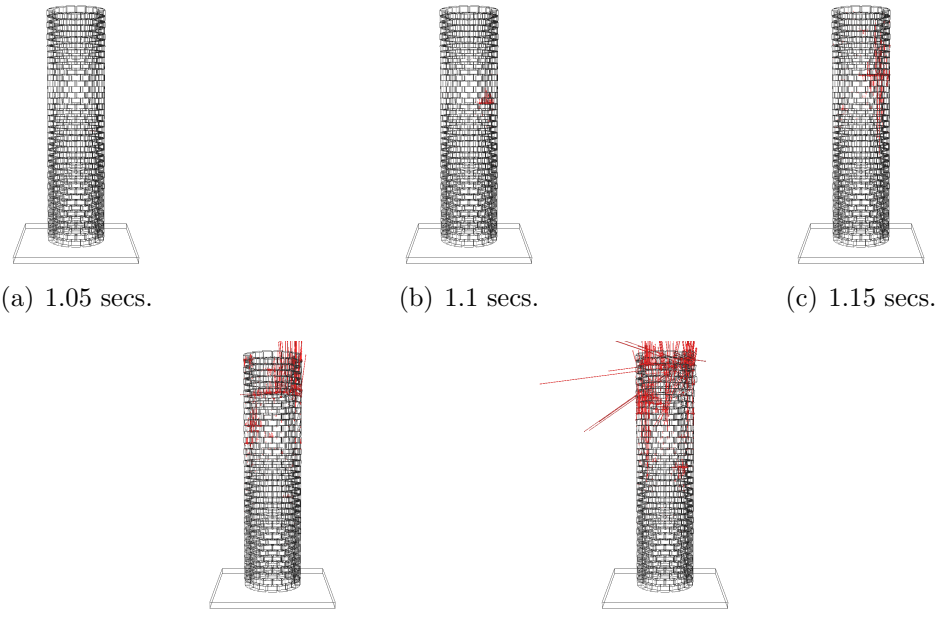
The rippling effect can also come from making a bad analysis of stack layers. Typically, what happens is that some stacked configuration falls down due to some other interaction. Thus, objects at the top-most stack height tumble down, and during their fall they



(a) 1.2 secs. (b) 1.25 secs. (c) 1.3 secs.

(d) 1.35 secs. (e) 1.4 secs.

Figure 6.58: Rippling effect seen in a 640 brick tower simulation using $f = 0.05$.



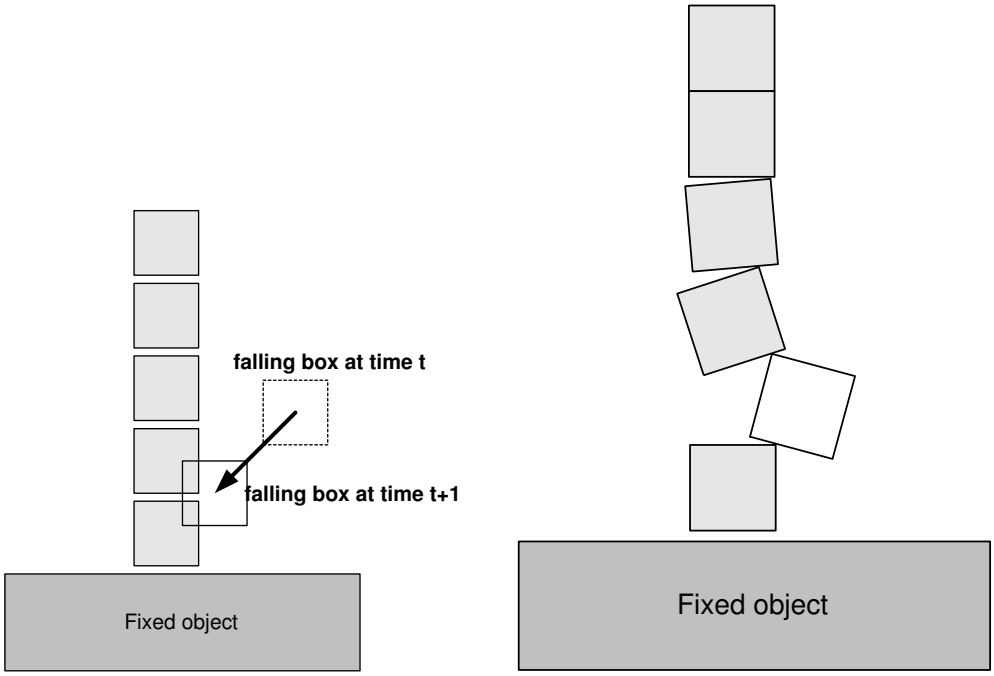
(a) 1.05 secs. (b) 1.1 secs. (c) 1.15 secs.

(d) 1.2 secs. (e) 1.25 secs.

Figure 6.59: Penetration errors causing a rippling effect in a 640 brick tower simulation using $f = 0.05$. Penetration depths are drawn as red arrows, multiplied by a factor of 50 for better visualization.

```
modified-velocity-based-shock-propagation(f,h)
  collision detection at time t
  dynamics(f*h)
  shock-propagration( dynamics( (1-f)*h ), error-correction() )
  collision detection at time t + h
  error-correction()
  t = t + h
```

Figure 6.60: Pseudo-code of the modified velocity based shock-propagation algorithm which adds robustness against rippling.



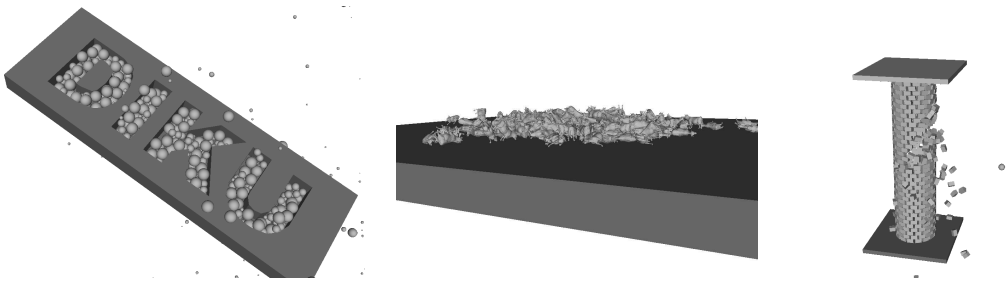
(a) High speed colliding object penetrate lower stack layer.

(b) Shock-propagation causes rippling effect.

Figure 6.61: Rippling effect caused by high speed moving objects, changing their stack height to a lower layer.

gain speed due to gravity. On their path down, they may hit other objects at lower stack layers. Due to their high speed and the nature of the explicit time-stepping, deep penetrations may occur at these lower stack layers. The shock-propagation sees the falling and penetrating object as being at the same stack layer as the objects it is colliding with. The error-correction will therefore project all objects at higher stack-layer upwards to correct the penetration. This projection is then seen as a rippling effect in the simulation.

One way to circumvent the problem may be not to allow the stack height of an object to decrease during a simulation. Thus, a top-most object continues to be at the top stack-height, even though it is falling down and colliding with objects at lover stack layers. We have not tried this solution to the rippling problem and leave this for future



(a) 1000 balls of various size falling onto an inclined plane with engravings.

(b) 250 cows falling into a pile.

(c) Roof falling onto tower of 640 bricks, followed by a canon ball collision.

Figure 6.62: Three test configurations simulated using the modified velocity based shock-propagation algorithm. Simulation results of these configurations can be seen in Figure 6.30, Figure 6.38, and Figure 6.72.

work. Notice that an implicit time-stepping method may also be able to solve the problem since it can see the collision before the penetration occurs. We have not tried this solution either, because the collision detection engine is not easily changed to re-evaluate generated contact points which is necessary because the implicit method needs to detect contacts at time $t + \Delta t$. This is done by using predicted positions at time $t + \Delta t$ for generating contact points. Afterwards, the generated contacts is re-evaluated at time t. We have not implemented this option in our current collision detection engine. Another solution would be to built the contact graphs as done in [70]. Their method will not suffer from the rippling effect we have described here. The collision detection engine we have used does not allow for easy adaption of the contact graph used in [70].

6.4.6 Results

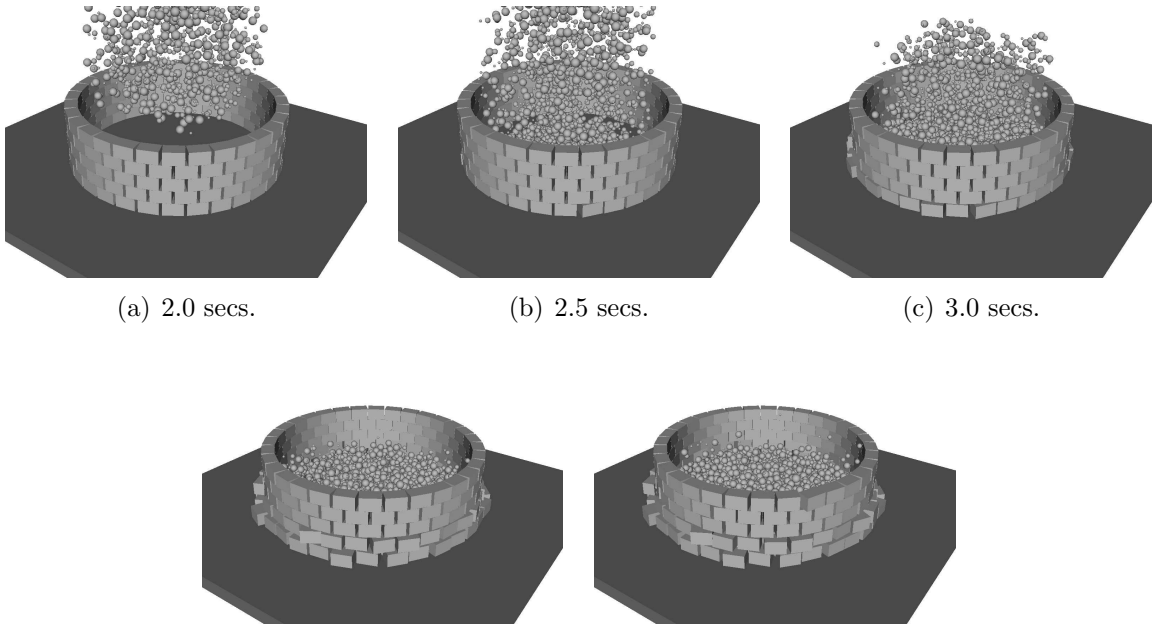
We have applied the modified velocity based shock-propagation algorithm in Figure 6.60 to the test configurations shown in Figure 6.62 and Figure 6.63.

During simulation we measured the total frame time, and the time used for collision detection and the total number of generated contact points. Detailed results are shown in Figure 6.64. Notice the dissimilarity in the two kinds of respective curves. This indicates that all four configurations really behave differently.

The algorithm we have described for computing the actual time-stepping, i.e. frame time minus time used on collision detection, should be linear in the number of contact points. Figure 6.65 shows that this is in fact the case.

For completeness we have also plotted the time spent on collision detection. This is shown in Figure 6.66. These plots appear to have linear behavior, although in different directions. We believe that horizontal lines or lines with negative slope is a consequence of the caching schemes applied to the contact graph edges (see Chapter 5). The piecewise linear curves with positive slope, is an indication that the broad-phase collision detection algorithm quickly prunes away unnecessary tests, such that time is only spent on doing collision detection for objects in close proximity. The scattering of the cow-plot could indicate that the pruning capability of the sphere-trees could be improved.

It should be noted that the simulation in [70] ranges from 500-1000 objects, and



(a) 2.0 secs. (b) 2.5 secs. (c) 3.0 secs.

(d) 3.5 secs. (e) 4.0 secs.

Figure 6.63: Massive number of balls falling into a box silo, total number of objects is 3000.

frame times go from 5 minutes to 7 minutes. Comparing our frame times of similar sized configurations indicate a speed-up of a few hundred orders. However, there is still room for improvement in our implementation [113]. We do lose some performance due to the generality of the framework where-in we have implemented the algorithm shown in Figure 6.60.

6.5 Sleepy Policy

A sleepy object is an object that does not move, nor does its current state imply that it intends to move in the immediate future. There are various heuristics for determining when an object is sleepy, which we will briefly review below. The benefit of a sleepy policy is seen from the observation that if all objects in intermediate contact with each other are sleepy then this entire group of objects is non-moving, and there is no need to compute their motion. Thus, one can save computational resources this way, by simply doing nothing for sleepy objects.

The most simple heuristic to determine sleepiness is to track the position and orientation of an object [24]. If these are unchanged over a specified number of iterations then the object can be turned sleepy. This approach is not very successful in practice. Since the simulation is subject to numerical imprecision, the computed positions and orientations are rarely exactly the same over succeeding frames, even for non-moving objects. Thus, the approach necessitates threshold testing. This adds a scale-problem to the simulation, because large objects might be handled correctly, but small objects of the same scale as the threshold may be turned sleepy, even though they are moving. Another artifact comes from really slow moving objects. These could be turned sleepy if their motion is smaller

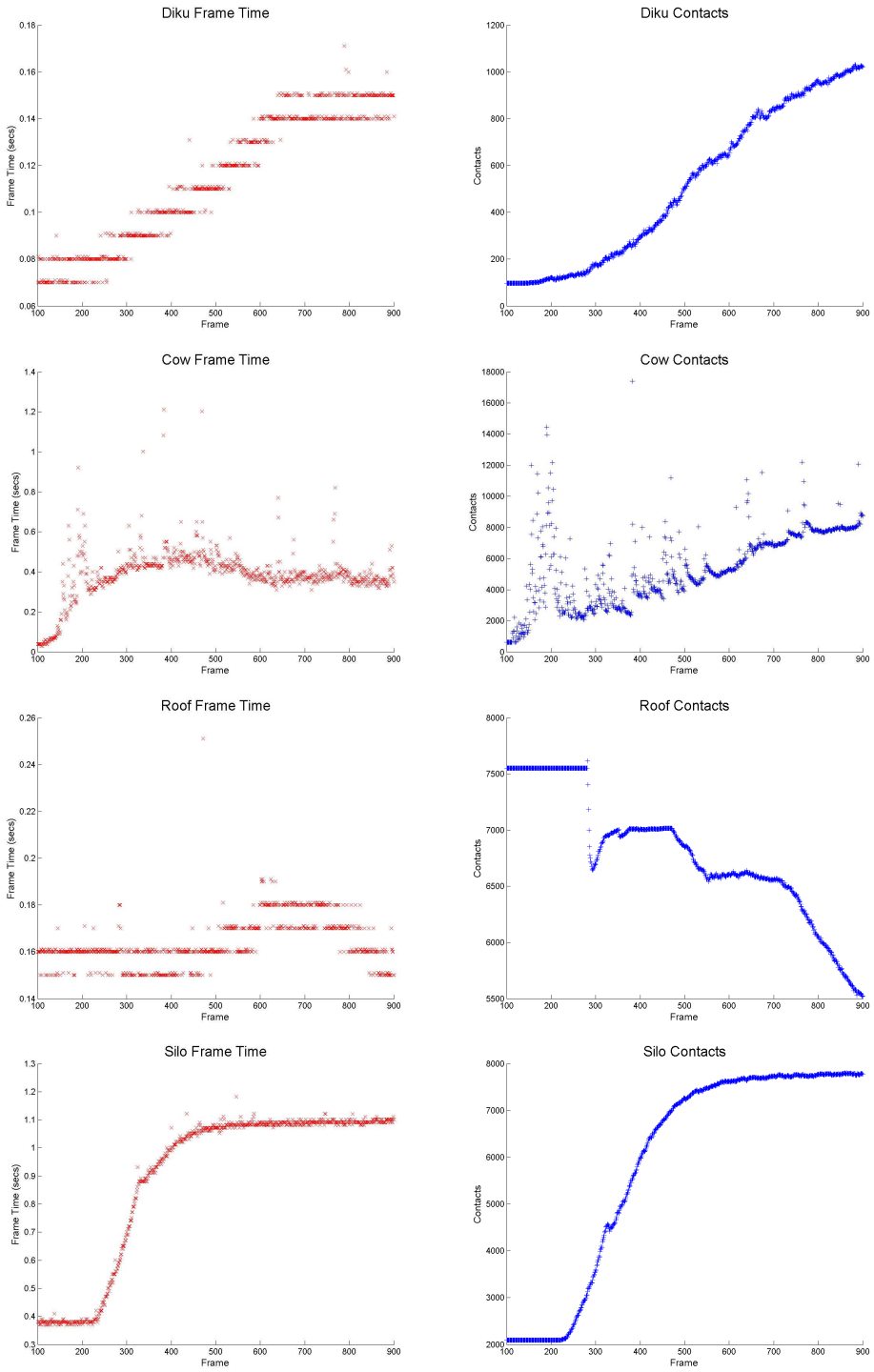


Figure 6.64: Total frame times and generated contact points as functions of frame number.

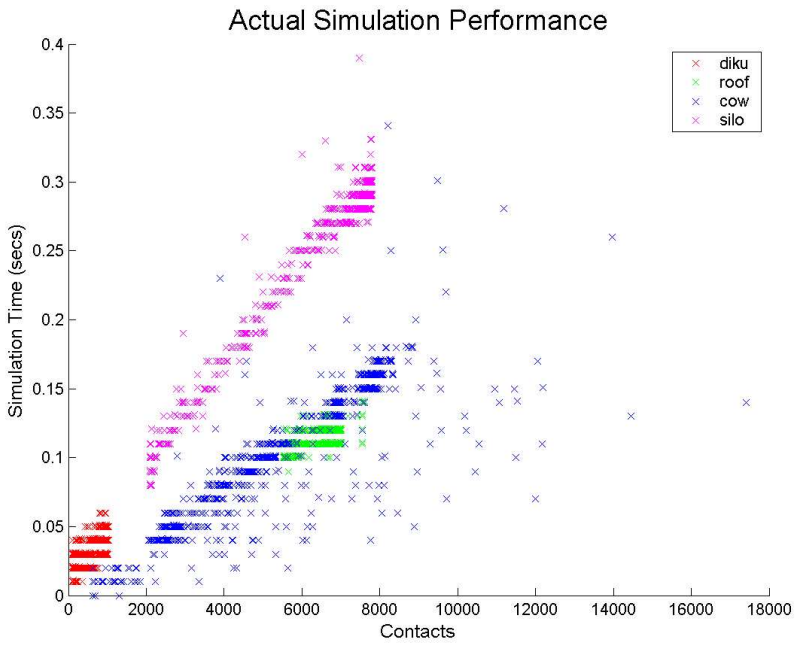


Figure 6.65: Total time spent on time-stepping as function of the number of contact points. That is the total frame time minus total time spent on collision detection.

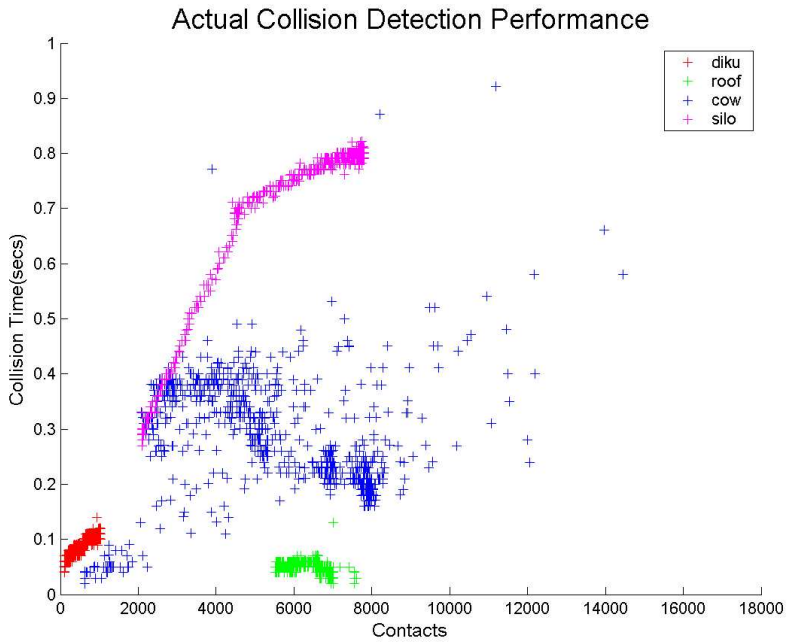


Figure 6.66: Total time spent on collision detection as a function of the number of contact points. That is to say, the total frame time minus the total time spent on time-stepping.

than the threshold. The classical example of the last artifact is a ball thrown upwards in a gravity field. Here, it can be seen that the ball suddenly hangs still in the air, when it reaches the top of its parabola-trajectory.

Another often used heuristic is to track the linear and angular velocities of an object [98, 112]. If the norms of the velocities are below some user specified threshold then an object can be turned sleepy. The major drawback of this is the need for handling linear and angular motion separately. The method overcomes the scale-problems of the position-based approach, but slow moving objects may still be set sleepy. To circumvent this problem, many people require that the velocity norms should be lower than a threshold over a number of succeeding iterations, before an object can be set sleepy. The number of iterations is usually determined by the order of the integration method, i.e. the time-stepping method, used to solve the equations of motion. Generally speaking, one wants more iterations than the order of the integration method.

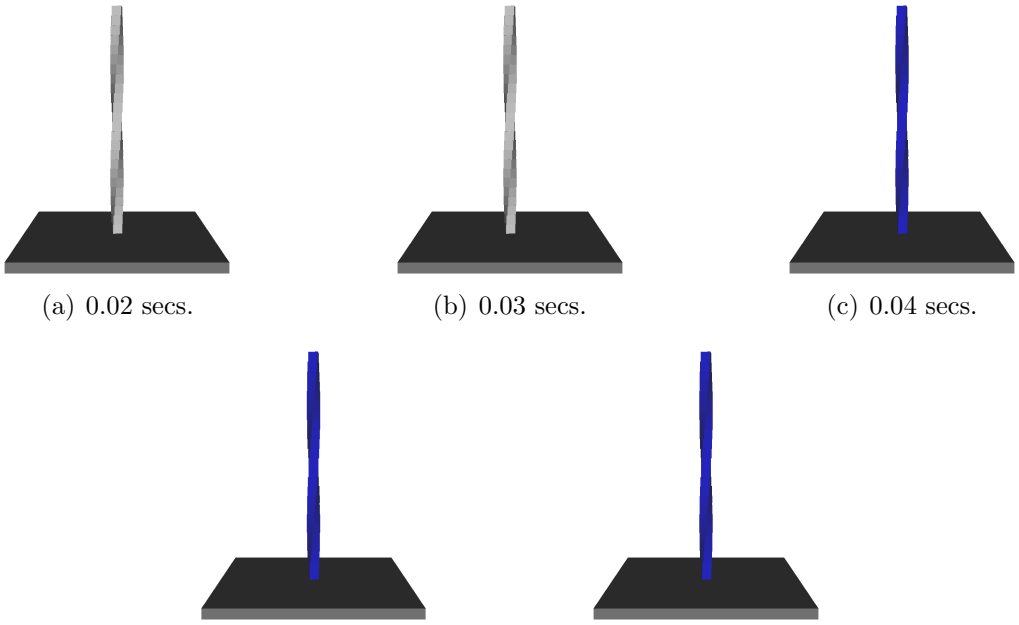
Note that sometimes the positional approach will fail to set objects to sleepy if the simulation method uses some projection method for doing error-correction. The velocity-based approach does not suffer from this drawback.

In [96, 130, 131] the total kinetic energy was used to determine when objects should be set to sleepy. The genius of this is that a natural kinematic weighting occurs of linear and rotational motion, combining them into one measure. This differs from previous approaches which use two threshold tests, one for the norm of the linear velocity and one for the norm of the angular velocity. Using the kinetic energy allows an end-user to set a single threshold value, implying that any object with a kinetic energy less than this threshold is set to sleepy. Thus, the kinetic energy approach overcomes the scale-problem, and the problem of treating linear and angular motion separately. However, it may suffer from the problem of slow-moving objects, which necessitates tracking over several iterations. We will suggest several extensions of the kinetic energy approach. These are:

- Object dependent threshold values. Kinetic energy is scaled by the inertia/mass properties of an object, so it makes sense that the absolute threshold test takes this into account. If not, objects with different mass properties will be treated differently. We simply scale a user specified threshold by the mass of an object.

- The use of both an absolute and a relative test. That is, we require that the kinetic energy of an object to be less than the absolute threshold over a number of iterations. The number of iterations is determined by the order of the integration method. The relative test consists of requiring that the total change in kinetic energy over the same number of iterations is non-decreasing. Only then is an object set to be sleepy. This means that if an object picks up kinetic energy during simulation then it will definitely not stay sleepy, even though the increase in kinetic energy is changing very slowly.

Combined, all these rules work fairly well in practice. In the limiting case of the absolute threshold going to zero and the number of iterations going to infinity, this heuristic will always give a correct answer to whether an object is moving or not. In practice however, it is not usable to have a very low absolute threshold or be tracking to many iterations. The aim is very quickly to determine sleepy objects, so the simulation method can stop



(a) 0.02 secs. (b) 0.03 secs. (c) 0.04 secs.

(d) 0.05 secs. (e) 0.06 secs.

Figure 6.67: Box-stack simulation with a low restitution coefficient of 0.1. Blue objects indicate sleepy objects. Observe that in the fifth frame, i.e. 0.04 seconds, all objects are sleepy.

simulating groups of sleepy objects. Due to this, the heuristic will sometimes fail for certain configurations. A good example is the see-saw configuration from Figure 6.33. Here the plank is moving very slowly, especially at its turn-around points. A future direction of work may be to incorporate the rate of change in kinetic energy in the sleepy heuristic to overcome the problems with configurations such as the see-saw.

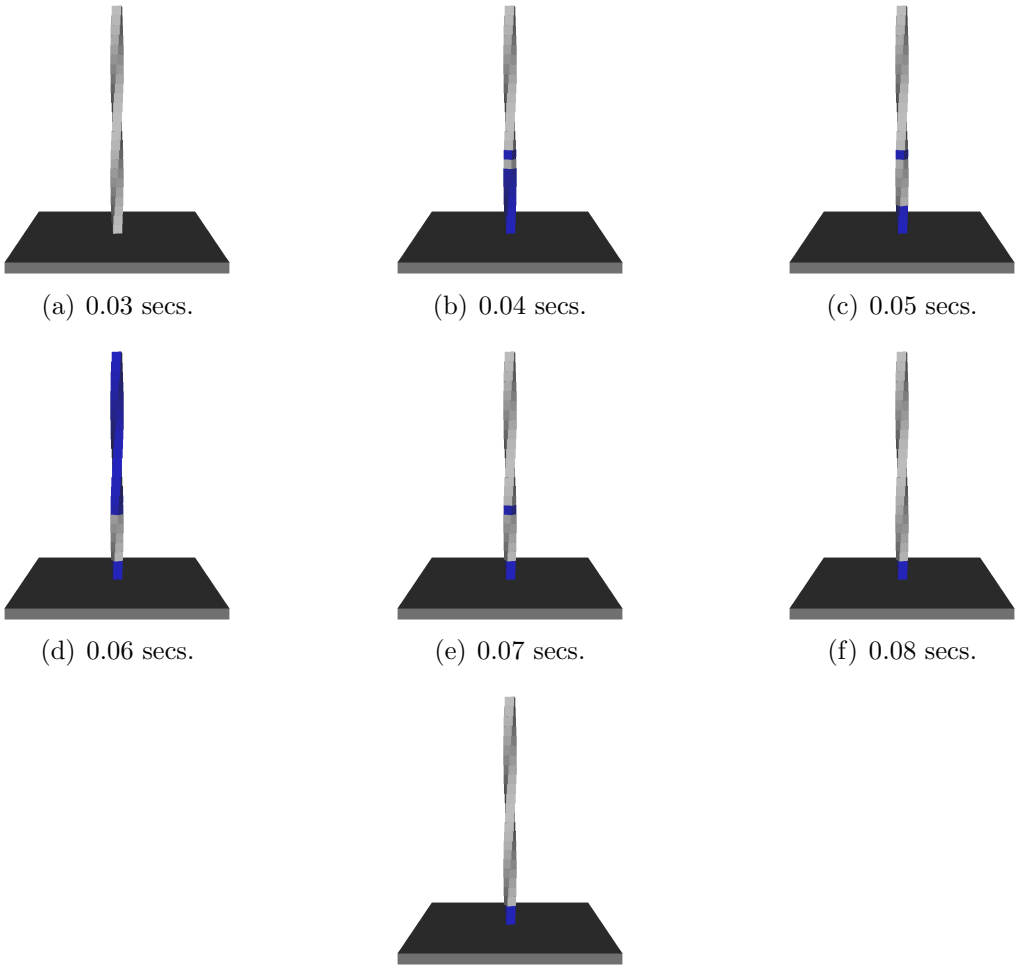
Although it seems attractive to apply a sleepy policy in the hope of obtaining real-time simulation, this is not a profitable approach. In real-time applications the user interacts with the world in a dynamic way, and he may thus put everything in motion. If the real-time performance relies on a sleepy policy to obtain its performance then the application will come to a halt, and eventually the user will not use it.

In our opinion, sleepy policies are far better suitable for controlled animations or off-line simulations such as those used for movie production. Of course, they can be used in real-time applications such as computer games, but they should not be added to ensure real-time performance of the computer game. They should be added simply to save computations in a greedy manner, or allowing for other moving objects in the configuration to use the computational resources saved by sleepy objects in order to get a more accurate simulation.

The physical properties and the chosen time-stepping scheme have a great impact on the effectiveness of the sleepy policy.

As a rule of thumb, the sleepy policy seems to work better for smaller restitution values than for larger ones. This is seen in Figure 6.67 and Figure 6.68.

The behavior can be understood by an analogy. Think of restitution as an insulation coefficient of simulation errors. What happens when restitution becomes large is that



(a) 0.03 secs. (b) 0.04 secs. (c) 0.05 secs.

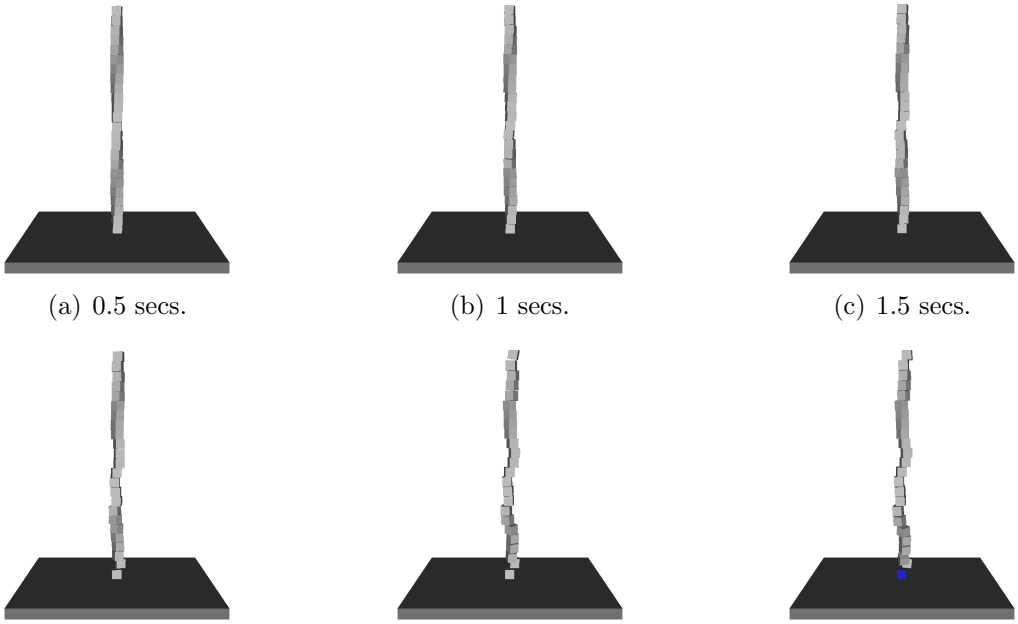
(d) 0.06 secs. (e) 0.07 secs. (f) 0.08 secs.

(g) 0.09 secs.

Figure 6.68: Box-stack simulation with a medium restitution coefficient of 0.4. Blue objects indicate sleepy objects. The large coefficient of restitution causes simulation errors to be propagated between neighboring boxes. It takes 28 frames (not shown) before all the boxes turn sleepy.

simulation errors in velocities can propagate through the configuration. A water rippling pattern of sleepy and non-sleepy objects is often seen as the simulation progresses. The effect can be damped by lowering the restitution. However, this will never solve the problem.

The problem of flickering sleepy patterns is often caused by the time-stepping scheme used, or the numerical method for dealing with the constraints, or some combination of both. In our specific case we use a an iterative LCP solver, which initially causes errors in the constrained velocities, i.e. the time-stepping method given by



(a) 0.5 secs. (b) 1 secs. (c) 1.5 secs.

(d) 2 secs. (e) 2.5 secs. (f) 3 secs.

Figure 6.69: Box-stack simulation with high restitution coefficient of 1.0. Blue objects indicate sleepy objects. Notice that the combination of large restitution and simulation errors causes the box stack to blow-up.

$$\vec{b} = J\left(\vec{u}^{\,t} + h\mathbf{M}^{-1}\vec{F}_{\text{ext}}\right), \tag{6.79a}$$

$$\mathbf{A} = J\mathbf{M}^{-1}J^T, \tag{6.79b}$$

$$\vec{\lambda} = \mathbf{lcp}\left(\mathbf{A}, \vec{b}\right), \tag{6.79c}$$

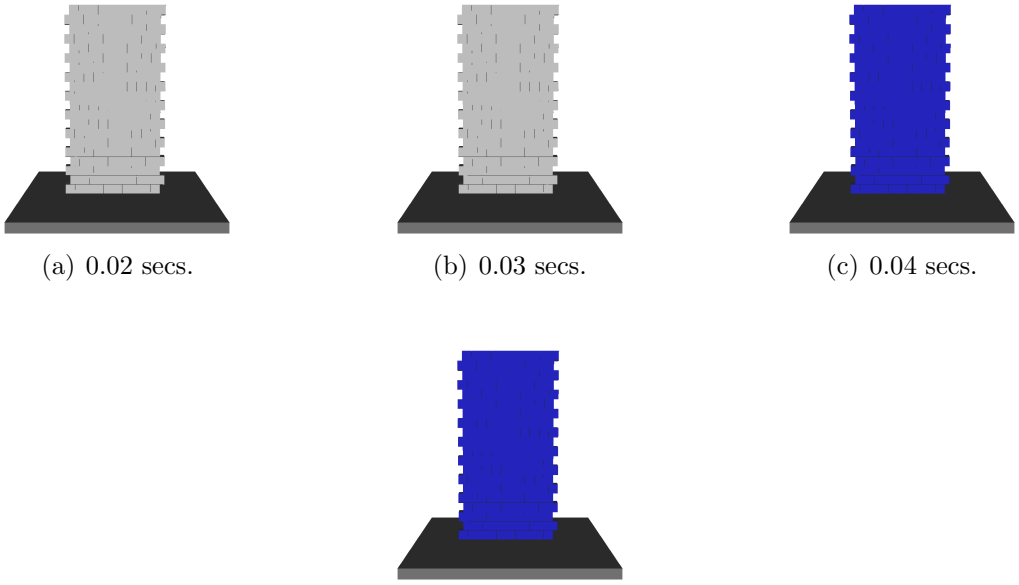
$$\vec{u}^{\,t+1} = \vec{u}^{\,t} + \mathbf{M}^{-1}J^T\vec{\lambda} + h\mathbf{M}^{-1}\vec{F}_{\text{ext}}, \tag{6.79d}$$

$$\vec{s}^{\,t+1} = \vec{s}^{\,t} + h\vec{u}^{\,t+1}, \tag{6.79e}$$

will have small errors in the constrained velocity $\vec{u}^{\,t+1}$. These errors will cause objects to be colliding in the succeeding time-step. If a large restitution coefficient is used, the simulation errors are propagated to nearby objects. This is the explanation for the rippling/flicking pattern of sleeping and non-sleeping bodies. Notice that we have slightly misused the notation in equation (6.79e), since $\vec{s}^{\,t}$ and $\vec{u}^{\,t+1}$ are of different dimensions.

Here, it should be noted that large restitution coefficients can blow up the simulation for the very same reason. Simulation errors are seen as collisions, so if errors are large and the restitution coefficient is large, a box stack can almost explode. This is shown in Figure 6.69.

If we could solve the problem exactly, there would not be any errors, and objects would quickly fall to sleep. To overcome the problems of the iterative LCP solver, we have extended the time-stepping to use error-correction by projection and shock-propagation as described in Section 6.4. To simplify the discussion, here our time stepping method can be described as taking a time-step weighted by some f-value, where $0 \leq f \leq 1$, followed by a shock-propagation weighted by a value of $(1 - f)$.



(a) 0.02 secs. (b) 0.03 secs. (c) 0.04 secs.

(d) 0.05 secs.

Figure 6.70: Wall simulation with $f = 0$. Friction was 0.25 and restitution was 0.4. Simulation time-step was 0.01 seconds. Blue objects indicate sleepy objects. Notice that all wall bricks are quickly turned sleepy, even the upper left most brick. This is due to using an over-aggressive sleepy threshold.

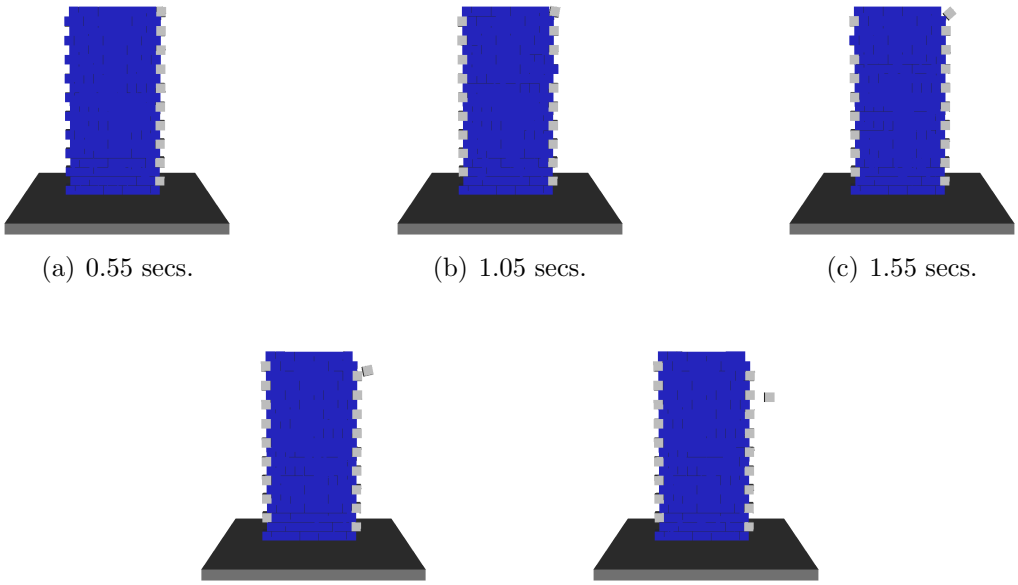
Intuitively, the shock-propagation works as a perfect "directional pre-conditioner" for any stacked configuration. As f goes to zero, it does a better and better job of killing simulation errors caused by the first time-step. The limiting case of f going to zero, yields perfect behavior. Objects in large stacks quickly settle into a sleepy state as shown in Figure 6.70.

Unfortunately, the limiting case of f going to zero has problems with systems not in equilibrium, as described in Section 6.4.4. Thus, one wants to have f larger than zero. As f goes to one, the simulation errors are allowed to become more dominant, causing object velocities to oscillate more and more. Thus, objects have trouble staying sleepy due to the relative test for the total kinetic energy change to be non-positive. In practice, it is not difficult to pick a sensible f-value, as shown in Figure 6.71. The major drawback is of course that this approach adds an element of parameter tuning to one's simulation.

Figure 6.72 shows results from a more interesting simulation. Notice how objects change their sleepy state on impact.

An alternative to the parameter tuning is to let the number of iterations used in the iterative LCP solver become over-wildly large, or simply use a direct LCP solver instead. Then the simulation errors will nearly vanish and f can be set to one without any problems.

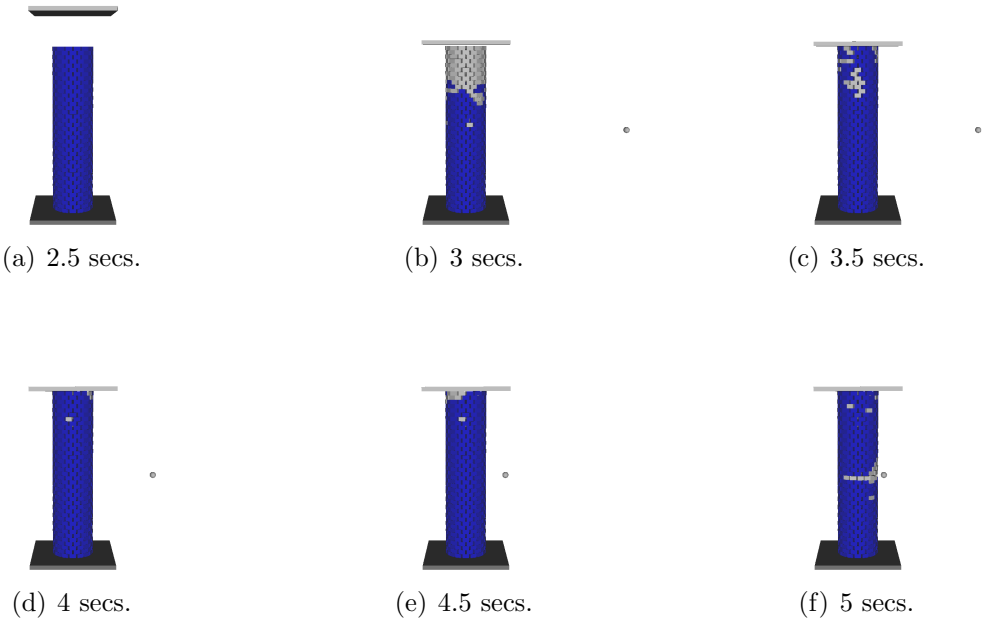
We have demonstrated the specific nature of the connection between physical properties and the simulation errors. The interaction is quite complex and it does put some limits on what values constitute a good simulation result. Clearly, the use of iterative LCP solvers dominate these aspects. Better methods for solving LCPs or merely iter-



(a) 0.55 secs. (b) 1.05 secs. (c) 1.55 secs.

(d) 2.05 secs. (e) 2.55 secs.

Figure 6.71: Wall simulation with $f = 0.125$. The wall contains 200 bricks of dimension $1m \times 1m \times 1m$. Friction was 0.25 and restitution was 0.4. Simulation time-step was 0.01 seconds. Blue objects indicate sleepy objects. Notice how internal bricks of the wall are turned sleepy, whereas the "tooth" along the side of the wall is non-sleepy, as expected.



(a) 2.5 secs. (b) 3 secs. (c) 3.5 secs.

(d) 4 secs. (e) 4.5 secs. (f) 5 secs.

Figure 6.72: Tower simulation with $f = 0.01$. The tower contains 640 bricks of dimension $1.5m \times 1m \times 1m$. Friction was 0.25 and restitution was 0.1. Simulation time-step was 0.01 seconds. Blue objects indicate sleepy objects. Notice how objects change their sleepy state on impact.

ative methods with better convergence would eliminate many of the problems we have addressed both in this section and the previous section.

Chapter 7

Conclusion

This dissertation has presented state-of-the art techniques for building robust, stable, and versatile multibody dynamics simulators, which can be used in the domain of computer animation.

We consider the major contribution of this dissertation to be a new practical approach for constraint based simulation. The main ingredients of this new approach are:

- A velocity based complementarity formulation.
- An iterative LCP solver using no more than 10 iterations.
- Adjustment of friction limits inside the LCP solver.
- A time-stepping scheme based on a weighting of an explicit time-stepping scheme followed by a shock-propagation step.
- A sleepy policy using both an absolute test and a relative test to determine sleepiness.
- Error correction by kinematic projection (i.e. first order physics).

All of the above fits nicely together, and a constraint based simulation is easily built, which allows for large time-steps of the order of 0.01 seconds and tolerance towards large penetration errors.

Shock-propagation has difficulties with feeling the weight of systems initially at rest but not in equilibrium, and rippling error correction caused by unfortunate stack layer analysis. The latter problem is related to our specific choice of algorithm for analyzing stack layers, and has nothing to do with the simulation method itself as explained in Section 6.4.5.

The first problem, however, is related to the accuracy of the numerical scheme, and a detailed discussion is given in Section 6.4.4 on how to deal with this problem. It is not an artifact caused by the time-stepping scheme, but a result of using iterative LCP solvers, which will yield inaccurate solutions.

In comparison with existing methods in the literature, the new approach for constraint based simulation is on the order of 100 times faster, and it is based on a mathematically well-posed model, which guarantees existence of a solution. The existence of a solution

is not guaranteed for impulse-based simulations or acceleration based complementarity formulations.

Penetrations are dealt with in a consistent manner with the dynamics formulation. Previously, large time-stepping meant on the order of 0.001 seconds. The simulation results in this dissertation clearly show that this domain has shifted an order. Today, large time-stepping is on the order of 0.01 seconds, comparable in size to the frame-rate of most real-time applications.

Minor contributions in this dissertation are considered to be:

- A general model for scripted bodies allowing animators to set up arbitrary paths in a simulation.
- A conceptual module design usuable as an educational tool.
- A formal description of contact graphs and several examples of speed-up methods relying on contact graphs.
- An elaborate presentation of constraint based simulation, and detailed discussions on difficulties such as time-stepping and error correction.

Finally, we have presented several ideas for further work, some of these include:

- Extending spline driven scripted motion to include rotational motion.
- Improving the stability of the numerics of spline driven scripted motion, when the derivatives vanish.
- Improving convergence rates of iterative LCP solvers, for instance, by using pre-conditioning, or using other methods for solving the LCP such as multi-grid solvers.
- More robust contact generation for general geometries.
- Better methods for analyzing stack layers for shock propagation, especially the case of cyclic dependencies.

7.1 The Future

Presently, we have constraint based simulators that scale linear in the number of contact points. From a time complexity viewpoint, we cannot do much better, unless we have a dynamical model that is stated in the object space [124]. Such a model may give hope of a simulation method that scales linear with the number of objects. Since the number of objects is most likely going to be less than the number of contact points, we may hope for a faster approach for large scale simulation. Unfortunately, there does not exist object space models that include friction.

It is our best estimate that constraint based methods such as the one presented in this dissertation will be prevalent for a long time. The near future will most likely contain a lot of studies of applying various numerical schemes to the specific problem of multibody dynamics. Convergence rate is currently a very hot topic in the middle-ware physics

industry [107, 89]. The ideas range from splitting methods, pre-conditioning to multi-grid methods.

Another avenue is to exploit the brute force computational resources of the more and more powerful emerging graphics processing units (GPUs). 64000 iterations (see Figure 6.23) on a GPU is really nothing to be worried about. We may even see emerging specialized computer hardware for physics-based simulation, due to the gaming industry's increasing interest in applying physics-based animation. The usage of GPUs will require a basic change in the GPU hardware to make read-backs from the GPU memory less costly. However, if the gaming industry keep on trying to go in that direction, it may well happen. The down-side of this aspect is that console-games probably will dominate, and the GPU generations currently evolve faster than consoles.

In essence, the future (as the past) will be about doing even larger scale real-time simulations.

We also feel that the future holds a keen interest for more elaborate friction models than Coulomb friction. Anisotropic friction, rolling, and spinning friction are missing in most of today simulators, even the one presented in this dissertation. Such a contribution to the graphics community will definitely be welcome.

Another aspect of multibody simulation, which we have not treated in this dissertation, is the question of control. We have presented a new tool for combining animated objects with dynamic simulation. Here, the simulation is used to create secondary motion in the animation. Techniques for creating primary motion using the multibody dynamic simulation may be something we will see more of in the future. It is already a hot topic in the graphics community [120], however current methods are time-consuming and not something you want to apply real-time in for instance a computer game.

7.2 Seven Rules of Thumb

To summarize the lessons learned during our own work with multibody dynamics simulation, we will end this dissertation with some rules of thumb for building multibody dynamics simulators in the hope that these rules will help others to avoid pitfalls:

- Use velocity based complementarity formulations, anything else is right down thoughtless. It is the only dynamical model that is well-posed regardless of the specific configuration you wish to simulate. All other paradigms have flaws in the sense that they can break or slow down given a particular configuration. It may well be that your numerical solution adds "noise" to this nice mathematically model, but you will still benefit from the mathematical nice properties in the end.

- Use error correction by projection. You may as well recognize that penetrations are a way of life. So instead of trying to get rid of them, accept that they exist and fix them instead. Don't use stabilization. It will just add a penalty force element to your simulator. You are better off without.

- Iterative methods may be very inaccurate, but they definitely provide you with a performance advantage that is hard to overlook. They are also easy to implement compared to direct methods, such as Lemke's Algorithm [109] or Dantzig Pivoting [16]. These will only give you a headache.

- Watch out for bad contact point generation. Most times when your simulations blow-up, it is due to some special case you have forgotten about in your contact generation algorithm. Start by making sure your contact generation works in all cases, or use simple geometries such as spheres and planes, where you really have to make an effort not to make the contact generation work.

- Get stacking to work. This is the most difficult thing to simulate. All the fancy stuff with things blowing up or tumbling down is easy. Almost any kind of simulator will give you good results and performance for such kind of simulations. However stable stacking is the challenging part. If your simulator can deal with it, it can deal with anything. Start out by adding friction and building stacks of balls right away. When these things work, it is childs-play to extend the simulator with more complex geometries.

- If you want to have high performance, use fixed time-stepping and forget about anything else. If your model can not handle fixed time-stepping, you may have the wrong model, or you did not read the first rule of thumb.

- Finally, keep in mind that you are working on a numerical model of a mathematical model that describes a physical model, which is a limited approximation of the real-world[1]. Thus, when you see something unexpected in your simulations it may be one of two things besides an implementation error: either you do not understand your numerical model, and it returns the correct results you asked it about, or your understanding or intuition of the real world is wrong. The solution to the first case is to remodel your model, so it returns what you expect. The second case is not a problem, but is due to your own lack of knowledge about the real-world.

[1]Notice that your simulator is a third generation model of the real world, each generation loses some detail about the real world.

Bibliography

[1] comp.graphics.algorithms. Newsgroup.

[2] Rigid Body Simulation Course 2002. Department of Computer Science, University of Copenhagen, course no. 178, http://www.diku.dk/undervisning/2002e/178.

[3] 3DFacto. Visual Configuration Software Provider. http://www.3dfacto.com/.

[4] M. Anitescu and F.A. Potra. Formulating dynamic multi-rigid-body contact problems with friction as solvable linear complementary problems. *Nonlinear Dynamics*, 14:231–247, 1997.

[5] M. Anitescu and F.A. Potra. A time-stepping method for stiff multibody dynamics with contact and friction. *International J. Numer. Methods Engineering*, 55(7):753–784, 2002.

[6] Mihai Anitescu, James F. Cremer, and Florian A. Potra. Formulating 3d contact dynamics problems. Reports on Computational Mathematics No 80/1995, Department of Mathematics, The University of Iowa, 1995.

[7] Mihai Anitescu, James F. Cremer, and Florian A. Potra. Properties of complementary formulations for contact problems with friction. Reports on Computational Mathematics No 83/1995, Department of Mathematics, The University of Iowa, 1995.

[8] Mihai Anitescu and Florian A. Potra. Formulating dynamic multi-rigid-body contact problems with friction as solvable linear complementary problems. Reports on Computational Mathematics No 93/1996, Department of Mathematics, The University of Iowa, 1996.

[9] Mihai Anitescu, Florian A. Potra, and David E. Stewart. Time-stepping for three-dimensional rigid body dynamics. *Comp. Methods Appl. Mech. Engineering*, 1998.

[10] William W. Armstrong and Mark W. Green. The dynamics of articulated rigid bodies for purposes of animation. *The Visual Computer*, 1(4):231–240, 1985.

[11] David Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. *Computer Graphics*, 23(3):223–232, 1989.

[12] David Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. *Computer Graphics*, 24(4):19–28, 1990.

[13] David Baraff. Coping with friction for non-penetrating rigid body simulation. *Computer Graphics*, 25(4):31–40, 1991.

[14] David Baraff. *Dynamic simulation of non-penetrating Rigid Bodies*. PhD thesis, Cornell University, 1992.

[15] David Baraff. Non-penetrating rigid body simulation. *State of the Art Reports of EUROGRAPHICS'93, Eurographics Technical Report Series*, 1993.

[16] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. *Computer Graphics*, 28(Annual Conference Series):23–34, 1994.

[17] David Baraff. Interactive simulation of solid rigid bodies. *IEEE Computer Graphics and Applications*, 15(3):63–75, May 1995.

[18] David Baraff. Physical based modeling: Rigid body simulation. ONLINE SIGGRAPH 2001 COURSE NOTES, Pixar Animation Studios, 2001. `http://www-2.cs.cmu.edu/~baraff/sigcourse/`.

[19] David Baraff, Andrew Witkin, John Anderson, and Michael Kass. Physically based modeling. Siggraph Course Notes, 2003.

[20] David Baraff, Andrew Witkin, and Michael Kass. Untangling cloth. *ACM Transactions on Graphics*, 22(3):862–870, 2003.

[21] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, 1994. `http://www.netlib.org/templates/Templates.html`.

[22] B.A. Barsky, N. Badler, and D. Zeltzer, editors. *Making Them Move: Mechanics Control and Animation of Articulated Figures*. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Morgan Kaufman Publishers, 1991.

[23] R. Barzel and A.H. Barr. A modeling system based on dynamic constraints. In *Computer Graphics*, volume 22, pages 179–187, 1988.

[24] Ronen Barzel. *Physically-based Modelling for Computer Graphics, a structured approach*. Academic Press, 1992.

[25] Ronen Barzel, John F. Hughes, and Daniel N. Wood. Plausible motion simulation for computer graphics animation. In *Proceedings of the Eurographics Workshop, Computer Animation and Simulation*, pages 183–197, 1996.

[26] William J. Bouma and George Vaněček, Jr. Modeling contacts in a physically based simulation. In *SMA '93: Proceedings of the Second Symposium on Solid Modeling and Applications*, pages 409–418, 1993.

[27] Gareth Bradshaw and Carol O'Sullivan. Adaptive medial-axis approximation for sphere-tree construction. *ACM Transactions on Graphics*, 23(1), January 2004.

[28] R. Bridson, R. Fedkiw, and J. Anderson. Robust treatment of collisions, contact and friction for cloth animation. *Proceedings of ACM SIGGRAPH*, 21(3):594–603, 2002.

[29] M. Buck and E. Schömer. Interactive rigid body manipulation with obstacle contacts. *Journal of Visualization and Computer Animation*, 9:243–257, 1998.

[30] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. Brooks/Cole Publishing Company, 6th edition, 1997.

[31] S. Cameron. Collision detection by four–dimensional intersection testing. *IEEE Transaction on Robotics and Automation*, 6(3):291–302, 1990.

[32] Stephen Cameron. Enhancing GJK: computing minimum and penetration distances between convex polyhedra. *Int. Conf. Robotics & Automation*, April 1997.

[33] S.L. Campbell, L.R. Petzold, and K.E. Brenan. *Numerical Solution of Initial-Value Problems in Differential Algebraic Equations*, volume 14 of *Classics in Applied Mathematics*. Society for Industrial & Applied Mathematics, 1996.

[34] Mark Carlson, Peter J. Mucha, and Greg Turk. Rigid fluid: animating the interplay between rigid bodies and fluid. *ACM Trans. Graph.*, 23(3):377–384, 2004.

[35] Anindya Chatterjee and Andy Ruina. A new algebraic rigid body collision law based on impulse space considerations. *Journal of Applied Mechanics*, 1998.

[36] M. B. Cline and D. K. Pai. Post-stabilization for rigid body simulation with contact and constraints. In *Proceedings of the IEEE Intl. Conf. on Robotics and Autom.*, 2003.

[37] J. D. Cohen, M. K. Ponamgi, D. Manocha, and M. C. Lin. Interactive and exact collision detection for large-scaled enviroments. Technical Report TR94-005, Department of Computer Science, University of N. Carolina, Chapel Hill, 1994. http://www.cs.unc.edu/ dm/collision.html.

[38] Murilo G. Coutinho. *Dynamic Simulations of Multibody Systems*. Springer-Verlag, 2001.

[39] John J. Craig. *Introduction to Robotics, mechanics and control*. Addision-Wesley Publishing Company, Inc, 2nd edition, 1986.

[40] M. Damsgaard. *Analysis of Rigid and Flexible Multibody Systems using Object-Oriented Programming*. PhD thesis, Institute of Mechanical Engineering Aalborg University, September 1999. Special Report No. 41.

[41] John Dingliana and Carol O'Sullivan. Graceful degradation of collision handling in physically based animation. *Computer Graphics Forum*, 19(3), 2000.

[42] David Eberly. Intersection of objects with linear and angular velocities using oriented bounding boxes. Online Paper. Magic Software, Inc. http://www.magic-software.com.

[43] Stephan A. Ehmann and Ming C. Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. In A. Chalmers and T.-M. Rhyne, editors, *EG 2001 Proceedings*, volume 20(3), pages 500–510. Blackwell Publishing, 2001.

[44] Douglas Enright, Stephen Marschner, and Ronald Fedkiw. Animation and rendering of complex water surfaces. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 736–744. ACM Press, 2002.

[45] Kenny Erleben. En introducerende lærebog i dynamisk simulation af stive legemer. Master's thesis, Department of Computer Science, University of Copenhagen, May 2001. number 00-09-1, `ftp://ftp.diku.dk/diku/image/erleben.00-09-01.pdf`.

[46] Kenny Erleben. An introduction to approximating heterogeneous bounding volume hierarchies. Technical Report DIKU 02/04, Department of Computer Science, University of Copenhagen, 2002. `http://www.diku.dk/publikationer/tekniske.rapporter/2002/02-04.pdf`.

[47] Kenny Erleben. A simple plane patcher algorithm. (submitted to journal), 2003.

[48] Kenny Erleben. Contact graphs in multibody dynamics simulation. Technical Report DIKU 04/06, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, May 2004. `http://www.diku.dk/publikationer/tekniske.rapporter/2004/04-06.pdf`.

[49] Kenny Erleben and Henrik Dohlmann. Contact graphs in multibody dynamics simulation. In Brian Elmegaard, Jon Sporring, Kenny Erleben, and Kim Sørensen, editors, *SIMS 2004*, pages 307–314, September 2004.

[50] Kenny Erleben and Henrik Dohlmann. The thin shell tetrahedral mesh. In Søren Ingvor Olsen, editor, *DSAGM 2004*, pages 94–102, August, 2004.

[51] Kenny Erleben, Henrik Dohlmann, and Jon Sporring. The adaptive thin shell tetrahedral mesh. (Submitted to Conference), 2004.

[52] Kenny Erleben and Knud Henriksen. B-splines. Technical Report DIKU 02/17, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, August 2002. `http://www.diku.dk/publikationer/tekniske.rapporter/2002/02-17.pdf`.

[53] Kenny Erleben and Knud Henriksen. Scripted bodies and spline-driven animation. Technical Report DIKU 02/18, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, August 2002. `http://www.diku.dk/publikationer/tekniske.rapporter/2002/02-18.pdf`.

[54] Kenny Erleben and Knud Henriksen. Scripted bodies and spline-driven animation. In Jeff Lander, editor, *Graphics Programming Methods*, chapter 1.4, pages 37–50. Charles River Media, Inc., 2003.

[55] Kenny Erleben and Jon Sporring. Collision detection of deformable volumetric meshes. In Jeff Landers, editor, *Graphics Programming Methods*, chapter 1.5, pages 51–68. Charles River Media, 2003.

[56] Kenny Erleben and Jon Sporring. Review of a general modular based design for rigid body simulators. (unpublished paper), 2003.

[57] Anthony C. Fang and Nancy S. Pollard. Efficient synthesis of physically valid human motion. *ACM Transactions on Graphics (TOG)*, 22(3):417–426, 2003.

[58] Raanan Fattal and Dani Lischinski. Target-driven smoke animation. *ACM Trans. Graph.*, 23(3):441–448, 2004.

[59] Roy Featherstone. *Robot Dynamics Algorithms*. Kluwer Academic Publishers, second printing edition, 1998.

[60] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 15–22. ACM Press / ACM SIGGRAPH, 2001.

[61] Bryan E. Feldman, James F. O'Brien, and Okan Arikan. Animating suspended particle explosions. *ACM Trans. Graph.*, 22(3):708–715, 2003.

[62] Nick Foster and Ronald Fedkiw. Practical animations of liquids. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 23–30. ACM Press / ACM SIGGRAPH, 2001.

[63] Ollie Frank and Johnson Thomas. *Illusion of Life: Disney Animation*. Hyperion Press, 1995.

[64] Tolga G. Goktekin, Adam W. Bargteil, and James F. O'Brien. A method for animating viscoelastic fluids. *ACM Trans. Graph.*, 23(3):463–468, 2004.

[65] Herbert Goldstein, Charles P. Poole, Charles P. Jr. Poole, and John L. Safko. *Classical Mechanics*. Prentice Hall, 3rd edition, January 2002.

[66] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics*, 30(Annual Conference Series):171–180, 1996.

[67] Stefan Gottschalk. *Collision Queries using Oriented Bounding Boxes*. PhD thesis, Department of Computer Science, University of N. Carolina, Chapel Hill, 2000.

[68] N.K. Govindaraju, S. Redon, M.C. Lin, and D. Manocha. Cullide : Interactive collision detection between complex models in large environments using graphics hardware. *ACM SIGGRAPH/Eurographics Graphics Hardware*, 2003.

[69] Jens Gravesen. The length of bezier curves. *Graphics Gems V*, pages 199–205, 1995.

[70] E. Guendelman, R. Bridson, and R. Fedkiw. Nonconvex rigid bodies with stacking. *ACM Transaction on Graphics, Proceedings of ACM SIGGRAPH*, 2003.

[71] J. K. Hahn. Realistic animation of rigid bodies. In *Computer Graphics*, volume 22, pages 299–308, 1988.

[72] Gary D. Hart and Mihai Anitescu. A hard-constraint time-stepping approach for rigid multibody dynamics with joints, contact, and friction. In *Proceedings of the 2003 conference on Diversity in computing*, pages 34–41. ACM Press, 2003.

[73] D. H. House and D. Breen, editors. *Cloth Modeling and Animation*. A K Peters, 2000.

[74] P. Hubbard. Real-time collision detection and time-critical computing. *Proc. 1 st Workshop on Simulation and Interaction in Virtual Environments*, July 1995.

[75] P. M. Hubbard. Interactive collision detection. In *Proceedings of the IEEE Symposium on Research Frontiers in Virtual Reality*, pages 24–32, 1993.

[76] Philip M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, 1995.

[77] Philip M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, 1996.

[78] John Hughes and Ronen Barzel. Siggraph 2003 course, plausible simulation. SIGGRAPH 2003 Conference Select CD-ROM and "Full Conference DVD, 2003.

[79] Thomas Jakobsen. IO Interactive. Personal Communication.

[80] Jeppe Johansen. Senior Researcher, M.Sc.Eng., Ph.D, Wind Energy Department, Risøe National Laboratory. Personal Communication.

[81] A. Joukhadar, A. Scheuer, and Ch. Laugier. Fast contact detection between moving deformable polyhedra. In *IEEE-RSJ Int. Conference on Intelligent Robots and Systems, Vol. 3*, pages 1810–1815, Kyongju (KR), October 1999.

[82] Karma. Middleware Physics Software Provider. MathEngine Karma, http://www.mathengine.com/karma/.

[83] Y.J. Kim, M.C. Lin, and D. Manocha. Deep: Dual-space expansion for estimating penetration depth between convex polytopes. *IEEE International Conference on Robotics and Automation*, May 2002.

[84] Y.J. Kim, M.C. Lin, and D. Manocha. Incremental penetration depth estimation between convex polytopes using dual-space expansion. *IEEE Transactions on Visualization and Computer Graphics*, 2003.

[85] D. Kleppner and R. J. Kolenkow. *An Introduction to mechanics*. McGraw-Hill Book Co., international edition, 1978.

[86] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, /1998.

[87] P. R. Kraus and V. Kumar. Compliant contact models for rigid body collisions. In *IEEE International Conference on Robotics and Automation*, Albuquerque, 1997.

[88] S. Krishnan, A. Pattekar, M. Lin, and D. Manocha. Spherical shell: A higher order bounding volume for fast proximity queries. In *Proc. of Third International Workshop on Algorithmic Foundations of Robotics*, pages 122–136, 1998.

[89] Claude Lacoursiere. Splitting methods for dry frictional contact problems in rigid multibody systems: Preliminary performance results. In Mark Ollila, editor, *The Annual SIGRAD Conference*, number 10 in Linkøping Electronic Conference Proceedings, November 2003.

[90] Eric Larsen, Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha. Fast proximity queries with swept sphere volumes. Technical Report TR99-018, Department of Computer Science, University of N. Carolina, Chapel Hill, 1999.

[91] John Lasseter. Principles of traditional animation applied to 3d computer animation. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 35–44. ACM Press, 1987.

[92] C. Lennerz, E. Schömer, and T. Warken. A framework for collision detection and response. *11th European Simulation Symposium*, pages 309–314, 1999.

[93] Kristoffer Møllerhøj. Simulering af rigid-body systemer med coulomb friktion ved brug af en implicit tidsskridts metode. Master's thesis, Department of Computer Science, University of Copenhagen (DIKU), February 2004. NO. 03-04-17.

[94] P. Löstedt. Mecahnical systems of rigid bodies subject to unilateral constraints. *SIAM Journal of Applied Mathermatics*, 42(2):281–296, 1982.

[95] Antoine McNamara, Adrien Treuille, Zoran Popovic, and Jos Stam. Keyframe control of smoke simulations. *Proceedings of SIGGRAPH 2003*, 2003.

[96] Victor J. Milenkovic and Harald Schmidl. Optimization-based animation. *SIGGRAPH Conference*, 2001.

[97] V.J. Milenkovic. Rotational polygon overlap minimization and compaction. *Computational Geometry. Theory and Applications*, 10(4):305–318, 1998.

[98] Brian Mirtich. Hybrid simulation: Combining constraints and impulses. *Proceedings of First Workshop on Simulation and Interaction in Virtual Environments*, July 1995.

[99] Brian. Mirtich. Hybrid simulation: combining constraints and impulses. *Tech. rep., Department of Computer Science, University of California, Berkeley.*, 1996.

[100] Brian Mirtich. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD thesis, University of California, Berkeley, December 1996.

[101] Brian Mirtich. Rigid body contact: Collision detection to force computation. Technical Report TR-98-01, MERL, March 1998.

[102] Brian Mirtich. V-clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics*, 17(3):177–208, July 1998.

[103] Brian Mirtich. Timewarp rigid body simulation. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 193–200. ACM Press/Addison-Wesley Publishing Co., 2000.

[104] Brian Mirtich and John F. Canny. Impulse-based simulation of rigid bodies. In *Symposium on Interactive 3D Graphics*, pages 181–188, 217, 1995.

[105] Neil Molino, Zhaosheng Bao, and Ron Fedkiw. A virtual node algorithm for changing mesh topology during simulation. *ACM Trans. Graph.*, 23(3):385–392, 2004.

[106] M. Moore and J. Wilhelms. Collision detection and response for computer animation. In *Computer Graphics*, volume 22, pages 289–298, 1988.

[107] Adam Moravanszky. A path to practical rigid body dynamics. Talk at the Annual CISP Workshop in Copenhagen, May 2004.

[108] Adam Moravanszky and Pierre Terdiman. *Game Programming Gems 4*, chapter Fast Contact Reduction for Dynamics Simulation, pages 253–264. Charles River Media, March 2004.

[109] Katta G. Murty. *Linear Complementarity, Linear and Nonlinear Programming*. Helderman-Verlag, 1988. This book is now available for download. `http://ioe.engin.umich.edu/people/fac/books/murty/linear_complementarity_webbook/`.

[110] Åke Nordlund. Associate Professor, Niels Bohr Institute, University of Copenhagen. Personal Communication.

[111] NovodeX. Middle-ware Physics Software Provider. NovodeX Physics SDK v2, http://www.novodex.com/.

[112] ODE. Opensource Project, Multibody Dynamics Software. Open Dynamics Engine.

[113] OpenTissue. Opensource Project, Physical based Animation and Surgery Simulation. http://www.opentissue.org.

[114] C. O'Sullivan and J. Dingliana. Real-time collision detection and response using sphere-trees. pages 83–92, 1999.

[115] Path. PATH CPNET Software, http://www.cs.wisc.edu/cpnet/cpnetsoftware/.

[116] Camilla Pedersen, Kenny Erleben, and Jon Sporring. Ballet balance strategies. In Brian Elmegaard, Jon Sporring, Kenny Erleben, and Kim Sørensen, editors, *SIMS 2004*, pages 323–330, September 2004.

[117] F. Pfeiffer and M. Wösle. Dynamics of multibody systems containing dependent unilateral constraints with friction. *Journal of Vibration and Control*, 2:161–192, 1996.

[118] J.C. Platt and A.H. Barr. Constraint methods for flexible bodies. In *Computer Graphics*, volume 22, pages 279–288, 1988.

[119] Jovan Popovic, Steven M. Seitz, and Michael Erdmann. Motion sketching for control of rigid-body simulations. *ACM Transactions on Graphics*, 22(4):1034–1054, 2003.

[120] Jovan Popović, Steven M. Seitz, Michael Erdmann, Zoran Popović, and Andrew Witkin. Interactive manipulation of rigid body simulations. *Proceedings of SIGGRAPH 2000*, pages 209–218, July 2000. ISBN 1-58113-208-5.

[121] N. Rasmussen, D. Nguyen, W. Geiger, and R. Fedkiw. Smoke simulation for large scale phenomena. In John Hart, editor, *ACM SIGGRAPH Transactions on Graphics*, volume 22, July 2003.

[122] Stephane Redon. Continuous collision detection for rigid and articulated bodies. To appear in ACM SIGGRAPH Course Notes, 2004, 2004.

[123] Stephane Redon. Fast continuous collision detection and handling for desktop virtual prototyping. *Virtual Reality Journal*, 2004. Accepted for publication.

[124] Stephane Redon, Abderrahmane Kheddar, and Sabine Coquillart. Gauss least constraints principle and rigid body simulations. In *Proceedings of IEEE International Conference on Robotics and Automation*, 2003.

[125] Stephane Redon, Young J. Kim, Ming C. Lin, and Dinesh Manocha. Fast continuous collision detection for articulated models. *Proceedings of ACM Symposium on Solid Modeling and Applications*, 2004. To appear.

[126] Stephane Redon, Young J. Kim, Ming C. Lin, Dinesh Manocha, and Jim Templeman. Interactive and continuous collision detection for avatars in virtual environments. *IEEE International Conference on Virtual RealityProceedings*, 2004.

[127] Anne Mette K. Sœrensen. Director, Research Department, Danish Meteorological Institute. Personal Communication.

[128] Sten Rettrup. Associate Professor, Department of Chemistry, University of Copenhagen. Personal Communication.

[129] J. Sauer and E. Schömer. A constraint-based approach to rigid body dynamics for virtual reality applications. *ACM Symposium on Virtual Reality Software and Technology*, pages 153–161, 1998.

[130] Harald Schmidl. *Optimization-based animation*. PhD thesis, Univeristy of Miami, May 2002.

[131] Harald Schmidl and Victor J. Milenkovic. A fast impulsive contact suite for rigid body simulation. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):189–197, 2004.

[132] Jonathan R Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, 1994.

[133] Kerawit Somchaipeng, Kenny Erleben, and Jon Sporring. A multi-scale singularity bounding volume hierarchy. Technical Report DIKU-04/08, Department of Computer Science, University of Copenhagen, 2004. `http://www.diku.dk/publikationer/tekniske.rapporter/2004/04-08.pdf`.

[134] Kerawit Somchaipeng, Kenny Erleben, and Jon Sporring. A multi-scale singularity bounding volume hierarchy. (Submitted to Conference), 2004.

[135] Peng Song, Jong-Shi Pang, and Vijay Kumar. A semi-implicit time-stepping model for frictional compliant contact problems. Submitted to International Journal of Numerical Methods for Engineering. Avaible at `http://www.mts.jhu.edu/~pang/dcm_kps.pdf`, May 2003.

[136] Abderrahmane Kheddar Stephane Redon and Sabine Coquillart. Fast continuous collision detection between rigid bodies. *Computer Graphics Forum (Eurographics 2002 Proceedings)*, 21(3), 2002.

[137] David Stewart. Convergence rate of iterative LCP solvers. Personal Communication, October 2004.

[138] David E. Stewart. Rigid-body dynamics with friction and impact. *SIAM Review*, 42(1):3–39, 2000.

[139] D.E. Stewart and J.C. Trinkle. Dynamics, friction, and complementarity problems. In *International Conference on Complementarity Problems*, Nov 1995.

[140] D.E. Stewart and J.C. Trinkle. An implicit time-stepping scheme for rigid body dynamics with inelastic collisions and coulomb friction. *International Journal of Numerical Methods in Engineering*, 1996.

[141] D.E. Stewart and J.C. Trinkle. An implicit time-stepping scheme for rigid body dynamics with coulomb friction. *IEEE International Conference on Robotics and Automation*, pages 162–169, 2000.

[142] K. Sundaraj and C. Laugier. Fast contact localisation of moving deformable polyhedras. In *IEEE Int. Conference on Control, Automation, Robotics and Vision*, Singapore (SG), December 2000.

[143] J. Teran, S. Blemker, V. Ng Thow Hing, and R. Fedkiw. Finite volume methods for the simulation of skeletal muscle. In D. Breen and M. Lin, editors, *ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA)*, pages 68–74, 2003.

[144] D. Terzopoulos, J.C. Platt, A.H. Barr, and K. Fleischer. Elastic deformable models. In *Computer Graphics*, volume 21, pages 205–214, 1987.

[145] Jeff Trinkle, Jong-Shi Pang, Sandra Sudarsky, and Grace Lo. On dynamic multi-rigid-body contact problems with coulomb friction. Technical Report TR95-003, Texas A&M University, Department of Computer Science., 3, 1995.

[146] Jeff C. Trinkle, James Tzitzoutis, and Jong-Shi Pang. Dynamic multi-rigid-body systems with concurrent distributed contacts: Theory and examples. *Philosophical Trans. on Mathematical, Physical, and Engineering Sciences*, 359(1789):2575–2593, December 2001.

[147] G. van den Bergen. Proximity queries and penetration depth computation on 3d game objects. *Game Developers Conference*, 2001.

[148] Gino van den Bergen. A fast and robust GJK implementation for collision detection of convex objects. *Journal of Graphics Tools: JGT*, 4(2):7–25, 1999.

[149] Brian Vinter. Associate Professor, Department of Mathematics and Computer Science, University of Odense. Personal Communication.

[150] CMLabs vortex. Phsycis Simulation Software. http://www.cm-labs.com/products/vortex/.

[151] Emo Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, LNCS. Springer, 1991.

[152] Jane Wilhelms and Allen Van Gelder. Fast and easy reach-cone joint limits. *Journal of graphics tools*, 6(2):27–41, 2001.

[153] Andrew Witkin and Michael Kass. Spacetime constraints. *ACM SIGGRAPH, Computer Graphics*, 22(4):159–168, 1988.

[154] M. Wösle and F. Pfeiffer. Dynamics of multibody systems with unilateral constraints. *International Journal of Bifurcation and Chaos*, 9(3):473–478, 1999.

[155] G. Zachmann. Rapid collision detection by dynamically aligned dop-trees. In *Proc. of IEEE, VRAIS'98*, Atlanta, March 1998.

[156] O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method: The Basis*, volume 1. Butterworth-Heinemann, 5th edition, 2000.