A Study on Real-time Multibody Simulation with Contacts

Carl Johan Gribel
Numerical Analysis
Lund Institute of Technology

Supervisor: Prof. Claus Führer

November 11, 2009

Abstract

The so called contact problem, referring to the resolution of geometric contacts in a dynamic simulation, is an interesting, multifaceted one. Consider the simulation of a pile of bodies: most constellations of bodies in contact are, directly or indirectly, dependent on many other such constellations, forming an entangled web of bodies in collective, yet fragile, equilibrium.

This thesis will present an impulse-based method for solving multibody systems with contact; in its theoretical formulation as well as in its implementation. Contact resolution and behavior, including friction and elasticity, will be setup using kinematic constraints. Much the same way, it turns out, as general joints, incentivizing a choice of constraint solver method adapted for both. Each necessary simulation component, from collision detection to visualization and user interaction, will subsequently be detailed and combined into a simple, yet fast and robust simulation framework.

Contents

1 Introduction

Dropping a toy block to the floor, one expects it to accelerate through the air for a short while, before landing violently. The impact will most likely make the block spin and bounce off in a random direction. After a while, past some chaotic tumbling, the block will eventually settle, surrendering to friction. Easy enough in real life, but how can this behavior be re-created in a simulation?

Interactive multibody simulation, from model to computer application, spans several disciplines. The free motion of bodies in a mechanical system is expressed by Newton's second law: $Ma = f$. This formulation, called a *many-body system*, is sufficient for systems with forces only. When adding body geometry, and the notion of geometric overlap; *collisions* or *contacts*; motion is no longer free but *constrained*. The constrained system, now a *multibody system*, include additional *Lagrange Multipliers*, λ: $Ma = f + J^T\lambda$, where J is a constraint matrix.

The issue of solving the multibody system equations eventually turns out to be a *Linear Complementarity Problem* (LCP), which can be solved in different ways. In this thesis, an iterative method is used, in large based on the merits of iterative methods in interactive, performance sensitive environments.

Then there is the collision detection process, in which collisions are detected and data necessary to resolve them is gathered. Collision detection is by itself a vast subject, rooted in computational geometry. Being one of the most computationally expensive aspects of the simulation (naïvely, all bodies need to be tested against all other bodies), it often requires supplementary algorithms reducing this cost in different ways.

Finally, in order to produce and display the end results visually in a credible manner, every part of the simulation must operate in symbiosis; model, method and program implementation.

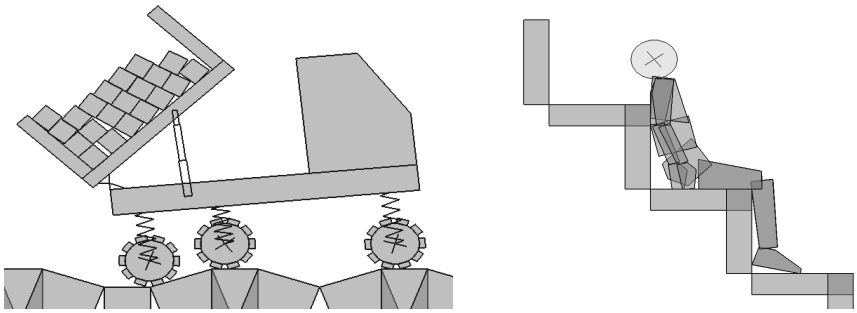


Figure 1.1 A truck model (*left*) with an articulated bed and suspended wheels, and a humanoid, "ragdoll" model (*right*), with body parts connected by joints. Videos demonstrating both models can be viewed online [Grib08]. Humanoid model courtesy of Mark Bayazit [Byaz08]

1.1 Scope of the Work

In this thesis, the following key directions have been made.

- **2D**. Working in two instead of three dimensions simplify several aspects: rotations have one instead of three degrees of freedom; there are no Coriolis forces; friction is simpler etc. However, 2D still reveals most relevant dynamic effects. It also simplifies visualization and user interaction.

- **Real-time and Interactivity**. The simulation is expected to run in real-time while simultaneously being subject to external input and system manipulations. These criteria pose an extra challenge to the extent that, referring back to the block-dropping scenario, humans possess an inherently well developed perception of how mechanics ought to behave: they know right from wrong when they see it, thus, on one hand, demanding a high fidelity simulation. On the other hand, however, this perception is based on heuristics and patterns, not n:th-decimal accuracy, and so cutting some selected corners, in the name of performance, can often be considered acceptable.

- **Discrete time, fixed timestep**. This is a consequence of how the system equations are derived and solved, which will be apparent in due time.

- **Rigid bodies**. Only non-deformable bodies are used.

- **Geometric primitives: polygon, circle**. These primitives are simple enough to represent, and to manage during collision detection, yet flexible enough to combine into composite geometries.

1.2 Chapter Overview

The constrained system equations will be introduced and derived into a velocity form LCP in chapter 2. The constraint equations and several types of joints will be presented in 2.2-2.6. The contact model follows in 2.7, explaining restitution, friction and other contact-related issues. The constraint solver is presented in 2.9.

In chapter 3 the various phases of collision detection is explained, as well as its role in supplying the solver with contact data.

Chapter 4 details the implementation process; tools, classes and implementation-specific aspects of the simulation. In chapter 5, two test cases are presented.

2 Kinematic Constraints

A mechanical system under the rule of Newton's second law: $f = ma$, is considered a *free system*. Despite the fact that internal or external forces may effectively impose limits on the motion of the system, all states are indeed still analytically viable. In a constrained system however, motion is restricted according to kinematic relations called *constraints*, which are central in multibody dynamics. They are for instance used to model *joints*, which can be seen as idealizations of real-world mechanical devices such as hinges and sliders. In the context of this thesis, constraints are also used to model more subtle applications such as motors and friction.

A classical example of a constrained system is the simple pendulum: a bob hanging in a mass-less rod, oscillating back and forth under the influence of gravitational force. The constraint relation in this case states that the length of the rod must remain the same throughout the simulation.

2.1 Constraint Solver Overview

Several methods for solving constrained systems are available. One strategy is to consider the kinematic topology of the system, and to express the motion of each body with respect to its interconnections using *relative coordinates* [ESFu98]. Such a system retains the form of an ordinary differential equation since no additional equations are needed. A drawback is that the mass matrix may become position dependent and non-diagonal. Another strategy is to use *explicit forces*, such as springs, to uphold the constraints. This approach potentially raises stability issues, since constraint accuracy require stiff springs, which might in some cases make numerical integration less efficient.

In *implicit* or *Lagrange multiplier-based* methods, the constraint relations are instead defined separately in the form of *constraint equations*, which are satisfied by additional *constraint forces* acting on the system. By the introduction of algebraic equations, the system no longer has the form of an ordinary differential equation - but of a differential algebraic equation (DAE).

Since these methods enforce constraints essentially by applying forces in order to alter accelerations, they are said to be *force-based*. *Impulse-based methods*, on the other hand, alter velocities directly through the use of impulses. To make this possible, the system equations needs some rewriting.

But let's start by formalising from the original system equations. Consider an N-body system with K holonomic constraints $C = g(p)$ and $3N$ constraint forces f_c. The constrained motion then has the form

$$M\dot{v} = f_{ext}(t, p, v) + f_c(p, \lambda_f) \tag{1}$$
$$0 = g(p) \tag{2}$$

According to *D'Alembert's principle of virtual work* it can be shown that the manifold of free motion defined by the constraint equations is orthogonal to the constraint forces [ESFu98]: $f_c(p, \lambda_f) = \frac{d}{dp} g(p)^T \lambda_f$, and we define

$$J(p) := \frac{d}{dp} g(p) \tag{3}$$

as the *constraint matrix* or *constraint Jacobian*, $J \in \mathbb{R}^{K \times 3N}$. We now have

$$M\dot{v} = f_{ext}(t, p, v) + J(p)^T \lambda_f \tag{4}$$
$$0 = g(p) \tag{5}$$

Solving this system is a matter of finding the K unknown *Lagrange multipliers* λ_f, representing the magnitudes of the constraint forces $J(p)^T \lambda_f$. This procedure is described further by [Bara96].

Now, in order to use an impulse-based solution we need to express (3) in terms of velocities instead of accelerations [StTr96]. This is performed by numerical integration. Assuming a fixed timestep h, explicit Euler-integration approximates the acceleration $\dot{v}$ in terms of velocities at the end and beginning of the timestep:

$$\dot{v}^{(t)} \approx \frac{v^{(t+h)} - v^{(t)}}{h} \tag{6}$$

The time integral of the system in (4) thus reads

$$M(v^{(t+h)} - v^{(t)}) = \left(h \cdot f_{ext} + h \cdot J^T \lambda_f\right)^{(t)} \tag{7}$$

Here, we recognize that under the duration of a timestep, the constraint forces effectively turn into *constraint impulses* $P_c(p, \lambda_f) = h \cdot J(p)^T \lambda_f = h \cdot f_c(p, \lambda_f)$. By introducing as impulse magnitude λ, related to the force magnitude by $\lambda = h \cdot \lambda_f$, these impulses read

$$P_c(p, \lambda) = J(p)^T \lambda \tag{8}$$

And the system in (7) evolves into

$$M(v^{(t+h)} - v^{(t)}) = \left(h \cdot f_{ext} + J^T \lambda\right)^{(t)} \tag{9}$$

The constraint equations in (5) are expressed at velocity level by time derivation. The chain rule leads to

$$\dot{C}(p,v) = \frac{d}{dt}g(p) = \frac{\partial}{\partial p}g(p)\frac{\partial}{\partial t}p = J(p)v \tag{10}$$

From the standpoint of discrete time, the constraint equation is to be satisfied at the end of each timestep. Hence

$$\dot{C}^{(t+h)} = Jv^{(t+h)} \tag{11}$$

Combining (9) and (11), and rewriting to matrix form [Bara96] [AnPo97], we arrive at

$$\begin{pmatrix} 0 \\ \dot{C} \end{pmatrix} = \begin{pmatrix} M & -J^T \\ J & 0 \end{pmatrix} \begin{pmatrix} v^{(t+h)} \\ \lambda \end{pmatrix} + \begin{pmatrix} -Mv^{(t)} - h \cdot f_{ext} \\ 0 \end{pmatrix} \tag{12}$$

This is the core form of the discrete system equations in the context of this thesis. It turns out, as will be further elaborated during the coming sections, that this can be turned into a linear complementary problem (LCP). What's missing in its current form is a few additional condition, one of which permits for limits to be applied to the constraint impulses λ. This is necessary for some type of constraints, such as non-penetration contact and friction. We express these limits in general as

$$lo_k \leq \lambda_k \leq hi_k \tag{13}$$

It is important to note that impulse-based solvers operate solely on the velocity level $\dot{C}$ of the constraints. The mechanism by which the position level C is satisfied will be presented in section 2.8.

2.2 The Constraint Equations

The broad and somewhat diverse use of constraints makes the constraint equation an important topic to understand in order to grasp other, more abstract aspects of the simulation such as joint design and the solver algorithm. This section will explain the main shapes and characteristics of the constraint equations, yet without revealing their actual algebraic content, which will be covered in the next section.

To reiterate, constraint equations express kinematic relationships between bodies. They come in an assortment of categories, however this thesis will focus on *holonomic bilateral* constraints, $C(p) = 0$ and *holonomic unilateral* constraints $C(p) \geq 0$. Unilateral constraints are used for instance in non-penetration contacts, where constraint impulses are allowed to push bodies apart but not together. Going forward, we state that each individual constraint equation affect **exactly two bodies**.

To determine whether a constraint equation is satisfied or not, it is simply evaluated. A constraint that is completely and exactly satisfied evaluates to zero at all available levels: $C_k = \dot{C}_k = \ddot{C}_k = 0$. Since the impulse-based solver used in this thesis operates on velocity level only, the velocity constraint equations (10) are of foremost interest. That is

$$\dot{C}_k = J_k v = 0 \tag{14}$$

To picture the form of the constraint matrix J, consider the elements j_{kn} of K velocity constraints in an N-body system:

$$\begin{aligned} j_{11}v_1 + \ldots + j_{1N}v_N &= 0 \\ j_{21}v_1 + \ldots + j_{1N}v_N &= 0 \\ &\vdots \\ j_{K1}v_1 + \ldots + j_{KN}v_N &= 0 \end{aligned}$$

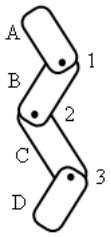
By defining $J = \begin{pmatrix} j_{11} & \cdots & j_{1N} \\ j_{21} & \cdots & j_{2N} \\ \vdots & \ddots & \vdots \\ j_{K1} & \cdots & j_{KN} \end{pmatrix}$ and $\begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{pmatrix}$,

the matrix form reads $\dot{C} = Jv = 0$. The elements of J constitute algebraic expressions, but since we happen to know that only two bodies are affected by each constraint, all but two elements will be zero. As a consequence, the element arrangement of J often exhibit distinguished patterns. Some of which are preferred, and some less so, by the solver. An overview of matrix patterns for different system configurations is presented in figure 2.2.1-2.2.3.
Another consequence of this conclusion is that a more compact, two-body form constraint equation can be used. Hence, a constraint k, acting on bodies i and j, can therefore be expressed

$$\dot{C}_k = j_{ki}v_i + j_{kj}v_j = \begin{pmatrix} j_{ki} & j_{kj} \end{pmatrix} \begin{pmatrix} v_i \\ v_j \end{pmatrix} = J_k v_{ij} \tag{15}$$

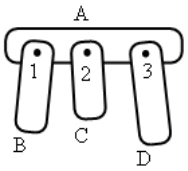
Before arriving at the final form of the constraint equation, an additional term d will be included. This term will prove useful in several upcoming situations. Appearing on the velocity level, it is consequently time-dependent. In conclusion, a velocity-level constraint k, acting on bodies i and j, has the general form

$$\dot{C}_k = J_k v_{ij} + d_k \tag{16}$$



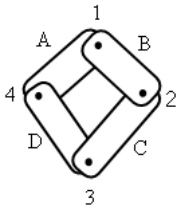
$$J = \left(\begin{array}{cccc|c} A & B & C & D & \\ j_{1A} & j_{1B} & & & 1 \\ & j_{2B} & j_{2C} & & 2 \\ & & j_{3C} & j_{3D} & 3 \end{array} \right)$$

Figure 2.2.1 Constraint matrix topology for a sparse system.



$$J = \left(\begin{array}{cccc|c} A & B & C & D & \\ j_{1A} & j_{1B} & & & 1 \\ j_{2A} & & j_{2C} & & 2 \\ j_{3A} & & & j_{3D} & 3 \end{array} \right)$$

Figure 2.2.2 Constraint matrix topology for a dense system.



$$J = \left(\begin{array}{cccc|c} A & B & C & D & \\ j_{1A} & j_{1B} & & & 1 \\ & j_{2B} & j_{2C} & & 2 \\ & & j_{3C} & j_{3D} & 3 \\ j_{4A} & & & j_{4D} & 4 \end{array} \right)$$

Figure 2.2.3 Constraint matrix topology for a cyclic system.

2.3 Distance Joint

A distance joint proclaims a fixed distance L between two points. It can be symbolized by a rigid, massless rod of length L. It is specified by:

Anchor points: r_i, r_j
Joint vector:

$$u = \frac{(p_j + r_j) - (p_i + r_i)}{||(p_j + r_j) - (p_i + r_i)||}$$

Position constraint:

$$C_k = ||(p_j + r_j) - (p_i + r_i)|| - L \tag{17}$$

Velocity constraint:

$$\begin{aligned}\dot{C}_k &= u \cdot (\dot{r}_j - \dot{r}_i) \\ &= u \cdot (v_j + w_j \times r_j - v_i - w_i \times r_i) \\ &= \begin{pmatrix} -u \\ -r_i \times u \\ u \\ r_j \times u \end{pmatrix}^T \begin{pmatrix} v_i \\ w_i \\ v_j \\ w_j \end{pmatrix} \\ &= J_k v_{ij} = 0 \end{aligned} \tag{18}$$

2.4 Hinge Joint

A hinge joint fixes two points to each other, effectively being a distance joint of zero length (although such a distance joint is not recommended). It can be symbolized by a pin, attaching two bodies together in one point. It is specified by:

Anchor points: r_i, r_j
Position constraint:

$$C_k = (p_j + r_j) - (p_i + r_i) = 0 \tag{19}$$

Velocity constraint:

$$\begin{aligned}\dot{C}_k &= \dot{r}_j - \dot{r}_i \\ &= v_j + w_j \times r_j - v_i - w_i \times r_i \\ &= \begin{pmatrix} -\mathbf{1}^2 \\ -\text{skew}(r_i) \\ \mathbf{1}^2 \\ \text{skew}(r_j) \end{pmatrix}^T \begin{pmatrix} v_i \\ w_i \\ v_j \\ w_j \end{pmatrix} \\ &= J_k v_{ij} = 0 \end{aligned} \tag{20}$$

2.5 Slider Joint

A slider joint allow motion only along a predefined reference direction. Its function resembles that of a cylinder with a piston, but allowing angular motion as well (unlike the cylinder). The slider joint can be used for instance in wheel suspensions, permitting the wheel to move only up and down relative the chassis. It is specified by:

Anchor points: r_i, r_j
Reference direction: l
Joint vector: $d = (p_i + r_i) - (p_j + r_j)$
Position constraint:

$$C_k = l^T d = 0 \tag{21}$$

Velocity constraint:

$$\begin{aligned}
\dot{C}_k &= \frac{d}{dt}\left(l^T d\right) \\
&= (w_i \times l) \cdot d + l \cdot (v_j + w_j \times r_j - v_i - w_i \times r_i) \\
&= \begin{pmatrix} -l \\ (d + r_i) \times l \\ l \\ r_i \times l \end{pmatrix}^T \begin{pmatrix} v_i \\ w_i \\ v_j \\ w_j \end{pmatrix} \\
&= J_k v_{ij} = 0
\end{aligned} \tag{22}$$

2.6 Angular Driver

Some constraints act directly to impose or restrict the relative angular velocity of two bodies, regardless of absolute rotation. These are called *drivers*, *motors* or *actuators*, and are often used in combination with torque limits to prevent "infinite" accelerations. A wheel for instance can be made to spin, brake, or even slow down gradually as if it where subject to rolling friction, by imposing a certain angular velocity (possibly zero) between the wheel and the chassis. By using torque limits, the capacity of the engine can be simulated. Without limits, the wheel would reach its target velocity instantaneously.

Consider the form of (16) and a target velocity of w_k, the constraint reads

$$\dot{C}_k = w_j - w_i - w_k = \begin{pmatrix} 0 \\ -1 \\ 0 \\ 1 \end{pmatrix}^T \begin{pmatrix} v_i \\ w_i \\ v_j \\ w_j \end{pmatrix} = J_k v_{ij} - w_k = 0 \tag{23}$$

2.7 Contact Model

In this section it will be explained how constraints can be used to model contact and collision behavior. As with other constraints presented this far, contact constraints

dictate kinematic relationships between pairs of bodies, but unlike them, contact constraints are short-lived (typically a single timestep) entities created and handed to the constraint solver "on the fly" by the collision detection module.
Some new semantics also arise. Anchor points are now referred to as the (single) *contact point*; the joint vector is referred to as the *contact normal*; positional error is referred to as *penetration depth*. Furthermore, while of less formal importance, a *contact* can be said to represent the set of data relevant to the contact constraint; bodies, contact points, contact normal etc; while a *collision* can be said to represent the event causing it.

The contact model is set up by two constraint relations, both of which focus on the relative velocity of the bodies: the first constraint relation eliminates its normal component in order to avoid penetration, and the second manipulates its tangent component in order to simulate friction.
To begin, let us consider the data at hand and then set up a few useful relations. The data for a given contact k, provided by the collision detection module, consist of

- Colliding bodies: i, j.
- Contact point: p_c.
- Contact normal: n_c.
- Penetration depth: δ_c. Assumed here: $\delta_c = 0$.

Henceforth, implying that all results are related to this k:th contact, k will be omitted for the sake of readability.
Contact point p_c expressed in the local frame for each respective body (previously referred to as anchor points)

$$r_i = p_c - p_i$$
$$r_j = p_c - p_j$$

Relative velocity at contact point

$$\dot{r}_{rel} = \dot{r}_j - \dot{r}_i = v_j + w_j \times r_j - v_i - w_i \times r_i$$

Normal component of relative velocity

$$\dot{r}_{rel,n} = n_c \cdot \dot{r}_{rel}$$

2.7.1 Non-Penetration Constraint

The non-penetration constraint resembles the distance joint in that it prevents relative motion along a certain direction, in this case the contact normal n_c. One important difference, however, is that this constraint may only prevent the bodies from moving *toward* each other, not *apart*. Preventing the latter would, potentially, cause colliding but not approaching bodies to stick together. As such, the non-penetration constraint is unilateral.

$$\begin{aligned}
\dot{C}_n &= \dot{r}_{rel,n} \\
&= n_c \cdot (v_j + w_j \times r_j - v_i - w_i \times r_i) \\
&= \begin{pmatrix} -n_c \\ -r_i \times n_c \\ n_c \\ r_j \times n_c \end{pmatrix}^T \begin{pmatrix} v_i \\ w_i \\ v_j \\ w_j \end{pmatrix} \\
&= J_n v_{ij} \geq 0
\end{aligned} \tag{24}$$

The position level of the constraint is, for now, assumed to be satisfied: $C_n = \delta_c = 0$ (no penetration). During simulation, the constraint in (24) will guide the solver to calculate a non-penetration constraint impulse according to (12) and (8), which, when applied to both bodies at the contact point (in their respective frame; r_i and r_j), eliminates all normal relative velocity

$$P_n = {J_n}^T \lambda_n \tag{25}$$

Also, since (24) is unilateral, the impulse magnitude in (25) is subject to the following limits

$$0 \leq \lambda_n \leq \infty$$

Restitution

The non-penetration constraint can conveniently be extended to make the colliding bodies "bounce" off from each other in an elastic manner. For this purpose, the constraint is set up to generate a non-zero, post-collision velocity proportional to the pre-collision velocity $\dot{r}_{rel,n}$. This is governed empirically by Newton's law of impact

$$\dot{r}_{rel,n}^{(t+h)} = -e \cdot \dot{r}_{rel,n}^{(t)} \tag{26}$$

Here, $\dot{r}_{rel,n}$ is superscripted with time-state to distinguish between the velocity before (t) and after $(t+h)$ the collision. The *coefficient of restitution* $e \in [0,1]$ states constitutively the fraction by which the kinetic energy is preserved and, indirectly, how much is lost due e.g. to heat induction and structural deformation. For $e = 0$ the collision is fully inelastic ("beanbag"), for $e = 1$ it is fully elastic ("solid rubber ball").
The post-collision velocity is included in the constraint equation (24) using an additional term

$$\begin{aligned} \dot{C}_n^{(t)} &= \dot{r}_{rel,n}^{(t+h)} + e \cdot \dot{r}_{rel,n}^{(t)} \\ &= J_n v_{ij}^{(t+h)} + e \cdot \dot{r}_{rel,n}^{(t)} \geq 0 \end{aligned} \tag{27}$$

Using this constraint, the end impulse (25) will push the colliding bodies apart in a bouncing manner.

Resting Contact

At low relative velocities, a special state of *resting contact* can be defined. This is especially useful in an implementation aspect, whereas bodies may have a hard time coming to rest due to numerical errors preventing the velocities to reach exactly zero. This can be resolved by treating the collision as fully inelastic when the relative velocity falls below a given tolerance, causing the remainder of the velocity to be eliminated. The coefficient of restitution e is thus set according to whether resting (28) or colliding (29) contact is present

$$e' = \begin{cases} 0 & \text{if } |\dot{r}_{rel,n}| \leq \mathsf{tol}_{vel}, \\ e & \text{otherwise} \end{cases} \tag{28}$$

(29)

2.7.2 Friction Constraint

Unlike real-life objects, which constitute some level of surface roughness that cause resistance during sliding and rolling contact, the rigid bodies in this thesis are, unfortunately one might say, perfectly smooth. To add some realism, a friction-esque behavior can be simulated using constraints in collaboration with Coulomb's law of friction.

First, we need to decide the tangent component of the relative velocity. direction. Since the tangential component of the relative velocity can be calculated as $\dot{r}_{rel,t} = \dot{r}_{rel} - \dot{r}_{rel,n}$, normalization leads to

$$t_c = \frac{\dot{r}_{rel,t}}{\|\dot{r}_{rel,t}\|}$$

The friction constraint is then set up to eliminate all tangential velocity between the bodies in contact

$$\begin{aligned}\dot{C}_t &= \dot{r}_{rel,t} \\ &= t_c \cdot (v_j + w_j \times r_j - v_i - w_i \times r_i) \\ &= \begin{pmatrix} -t_c \\ -r_i \times t_c \\ t_c \\ r_j \times t_c \end{pmatrix}^T \begin{pmatrix} v_i \\ w_i \\ v_j \\ w_j \end{pmatrix} \\ &= J_t v_{ij} = 0 \end{aligned} \tag{30}$$

This constraint will eventually, through the relations in (12) and (8), be used to calculate a friction impulse by the solver

$$P_t = {J_t}^T \lambda_t \tag{31}$$

Just like the non-penetration impulse P_n (25) eliminates the normal velocity component, the friction impulse eliminates the tangent component. But even though this is accurate according to the constraint equation in (30), it an unintended behavior. Some portion of the tangent velocity may well be eligible for removal – but not necessarily all of it. In addition, we desire the ability to tune this behavior in order to model different kind of materials. This is achieved by limiting the friction impulse using *Coulomb's law of friction*, which establishes a relationship between the magnitudes of the tangential and the normal impulses. The normal impulse magnitude in this context is the magnitude λ_n of the previously calculated non-penetration impulse P_n (25).
Coulomb's law distinguishes between *static friction* or *stiction*, and *kinetic friction*. Under static friction, the bodies in contact are "stuck", and should exhibit no relative tangential velocity. Static friction is active when the friction impulse magnitude λ_t is below a threshold $|\lambda_t| \leq \mu_s |\lambda_n|$, where $\mu_s \geq 0$ is the coefficient of static friction. Under kinetic friction, λ_t is instead limited by a fraction μ_k of the normal impulse magnitude λ_n. Here, $\mu_k > \mu_s$ is the coefficient of kinetic friction. The limits to be enforced on the friction impulse under static (32) and kinetic (33) friction, respectively, thus read

$$-\infty \leq \lambda_t \leq \infty \qquad \text{if } |\lambda_t| \leq \mu_s |\lambda_n|, \tag{32}$$

$$-\mu_k |\lambda_n| \leq \lambda_t \leq \mu_k |\lambda_n| \quad \text{otherwise} \tag{33}$$

The final friction impulse is then calculated using (30)

$$P_t = J_t^T \lambda_t \tag{34}$$

Note that under static friction the impulse magnitude is not limited, since all tangential velocity is indeed supposed to be eliminated.
Limits that depend on other simulation variables, such as in (33), are said to be *dependent* [Bara94].

2.8 Constraint Stabilization

Since an impulse-based solver operates exclusively on the velocity level of the constraint equations, the position level (including rotation) needs special attention. Ideally; if the velocity level were to be solved perfectly, and granted that no initial violations at the position level were present; the position level would indeed remain satisfied throughout the simulation. But such are not the conditions, and positions will, unattended, either remain wrong, if initially being so, or "drift" out of place due to errors at the velocity level. This issue is addressed by *constraint stabilization*, also called *error correction*.

In *Baumgarte's stabilization method* [Baum72], the positional error C is reduced gradually over several timesteps by error correcting impulses. This is achieved by settings up a velocity constraint that, based on C, will cause the error to be reduced by some amount at each timestep. The amount of reduction is controlled by a *error reduction parameter*, $\text{ERP} \in [0, 1]$ [Smit04].

$$\dot{C} = -\frac{\text{ERP}}{h} C \tag{35}$$

This is a first order ODE which, analytically, will cause the error to decline over time according to $C(t) = C_0 e^{-\frac{\text{ERP}}{h} t}$, where C_0 is the inital error.
The right-hand side of (35) is then incorporated in the expression for the very constraint k from which the error originates (be it a distance joint, a contact or something else). It can be regarded as an error correcting *bias* on the original constraint [Catt05].

$$\dot{C}_k = J_k v_{ij} + d_k = J_k v_{ij} - \frac{\text{ERP}_k}{h} C_k \tag{36}$$

The main challenge is to find a suitable value for ERP. Too low – and the error will disappear too slowly, causing joints to separate and colliding bodies to interpenetrate longer than desired. Setting it too high may cause the correction to overshoot and create new errors, resulting in oscillating joints and bodies unable to find a resting state of contact. Typically, each class of constraints has its own appropriate ERP value.

2.8.1 Stabilizing Unilateral Constraints

While the stabilization of bilateral constraints is straightforward; just feed a fraction of the position error C back in the velocity constraint (36); the unilateral constraints, such as contacts, are somewhat more tricky. Since it's in their nature to be either "on or off"; permitting some configurations but not others; abrupt, unpleasing behavior can occur when they switch between being active and inactive. One way to work around this issue is to allow some small error, making the stabilization process proceed until the constraint error is small enough, and then stop, leaving the constraint still active.
In the case of contacts, we label this error tolerance the *allowed penetration depth* ϵ_{depth}. Now, instead of using $C = \delta_c$ in (36), we use

$$C_k^{'} = \max\left(\min\left(\delta_c - \epsilon_{depth}, \infty\right), 0\right) \tag{37}$$

This way, the position error will be corrected just short of being completely removed, causing the non-penetration constraint to remain active.

2.9 Impulse-based Constraint Solver

Since unilateral constraints, such as contacts, introduce inequalities in the system equations, they no longer form a traditional linear system, but a *linear complementarity problem* (LCP). This was first formulated by Lötstedt [Löts82]. LCP's in general are discussed at length by Cottle et al. [CoPS92]. Solving an LCP involves using one of two principle techniques; *pivoting methods*, such as Lemke's method [Bara91], which generates a solution – if one exists – in a single computational step; and iterative methods, such as the *Jacobi* and the *Gauss-Seidel method*, which generate results that will converge towards a solution over several iterations steps.
Direct methods are able to produce very accurate results, but cannot guarantee solvability e.g. for systems with conflicting constraints. Iterative methods are typically less accurate, but can guarantee a result (be it a good or a bad one) regardless of the configuration. This is a compelling trait in an interactive simulation. Iterative methods also produce accessible intermediate results which can be used to set up criteria for iteration termination, such as error tolerances or execution time.
In this thesis an iterative solver based on the Gauss-Seidel method, inspired by Erleben [Erle04] and Catto [Catt05], is used. It is fast, easy to implement and, as it turns out, well suited for multibody systems.

To reiterate, the constrained system of (12), with the extended constraint form notion (16), reads

$$\begin{pmatrix} 0 \\ \dot{C} \end{pmatrix} = \begin{pmatrix} M & -J^T \\ J & 0 \end{pmatrix} \begin{pmatrix} v^{(t+h)} \\ \lambda \end{pmatrix} + \begin{pmatrix} -Mv^{(t)} - h \cdot f_{ext} \\ d \end{pmatrix} \tag{38}$$

Eliminating $v^{(t+h)}$ and expressing in terms of λ, the system can be rewritten to

$$JM^{-1}J^T\lambda = -d - J\left(v^{(t)} + h \cdot M^{-1}f_{ext}\right) \tag{39}$$

Setting $A = JM^{-1}J^T$ and $b = -d - J\left(v^{(t)} + h \cdot M^{-1}f_{ext}\right)$, we retrieve in compact form the linear system

$$A\lambda = b \tag{40}$$

Here, it follows that $A \in \mathbb{R}^{K\times K}$ and $\lambda, b \in \mathbb{R}^K$, where K represent, as before, the number of constraints.
In the Gauss-Seidel method, the A-matrix is decomposed into one diagonal, one upper triangular and one lower triangular matrix, all of which contribute to the solution through separate terms. The recursion takes, for some initial guess $\lambda^{(0)} \in R^K$, the form

$$\lambda_i^{(\nu+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}\lambda_j^{(\nu+1)} - \sum_{j>i} a_{ij}\lambda_j^{(\nu)}\right), \quad i = 1, \ldots, K \tag{41}$$

where $\nu = 0, 1, 2, \ldots$ is an iteration counter.
After each iteration, the elements of λ are *clamped* to fit within the limits set by the complementarity conditions of the LCP. That is, for the k:th constraint, limited by $lo_k \leq \lambda_k \leq hi_k$, the clamped value is set by

$$\lambda_k^{'} = \max\left(\min\left(\lambda_k, hi_k\right), lo_k\right)$$

By adding this step to (41), the resulting method is called *projected Gauss-Seidel*.

Now, as is thoroughly discussed in [Erle04], due to the block-wise characteristics of A in a multibody system, where constraints act on bodies in pairs exclusively, the A and b-matrices never need to be computed in their entireties. It is sufficient to use an inner loop to solve the system, computing and applying impulses individually per constraint, without large matrix-operations. This realization is especially valuable from an implementation standpoint, since a sequential, block-wise approach leads to a more compact simulation loop with smaller memory requirements.

3 Collision Detection

In chapter 2, various aspects of contact behavior was discussed, assuming that all necessary data; contact points etc; was readily available. In this chapter, we will find out how this data is gathered by the process of *collision detection*. Collision detection is typically made in a separate simulation module preceeding the constraint solver, feeding it with contact data. It takes as input the (set of) geometry of the simulated bodies, and produces output according to the queries *If* and *Where*. Is there a collision? Where is it?

An important consequence of time-discrete simulation schemes is that bodies advance in "steps", possibly causing collision where the actual time of impact was in-between the timesteps. This will cause geometric overlap or *penetration*, and is, as we have already seen, accounted for in the contact model. Worse is, however, when bodies move fast enough to pass right through each other, avoiding detection altogether. This effect is called *tunneling*, and can be avoided by introducing velocity caps and by keeping bodies from being too small.

Two different types of geometries are used in this thesis and its implementation: *convex polygon* and *circle*. A convex polygon is represented explicitly by n counter-clockwise ordered vertices P_k and normals N_k, $0 \leq k \leq n$. A circle is represented implicitly by the parametric equation $x^2 + y^2 = r^2$, where r is the circle radius.

To represent concave geometries, one can use *convex decomposition* to assemble multiple convex subsets into concave master sets [Eric05]. This is made possible through the use of *compound geometries*, which will be discussed further in chapter 4.

3.1 Narrow Phase Collision Detection

During the *narrow* or *exact phase* of the collision detection, bodies are analyzed in pairs. If they are considered to be colliding, one or multiple sets of contact data; *contacts*; are computed:

- Contact point, p_c: approximation of the point of impact.
- Contact normal, n_c: direction of the collision; impact plane normal.
- Penetration depth, δ_c: the amount of geometric overlap along n_c.

The actual test is carried in large by the *separating axis theorem* (SAT). It states, in 2-dimensional space, that two given convex sets are not intersecting if and only if there exist a line onto which their projections will be disjoint. This line is referred to as the *separating axis*. The separating axis theorem is useful to

us since each separating axis represents a "way out" for the overlapping bodies; a direction along which the bodies can be separated.

Unfortunately, there are usually many, possibly infinitely many, separating axes for a given pair of bodies. Therefore we also need to measure the distance required for the bodies to be separated, and then pick the separating axis with the shortest amount of required distance. This distance is called *separation*, δ, and is $\delta < 0$ when there is overlap along the separating axis. It is a generalization of the penetration depth, since it is defined also when there is no penetration at all (then $\delta > 0$). When considering two bodies i an j, we are thus interested in the *maximum separation* δ^{ij}_{max}, or, simply put, the "shortest way out".

3.1.1 Polygon–Polygon Collision Test

In the polygon–polygon test, all edge normals are possible separating axes, and therefore also δ^{ij}_{max}-candidates. To decide δ for any given normal, it is evaluated against the vertices of the opposite body. This test, for bodies i and j, is outlined like this:

1. Decide the maximum δ from evaluating all vertices of i, against all edges of j, where each test is given by

   $$\delta = \left(P^j_l - P^i_k\right)^T N^i_k$$

   for $0 \leq k \leq n^i$, $0 \leq l \leq n^j$. If a positive separation is encountered, $\delta > 0$, the bodies are per definition disjoint and the test is aborted.

2. Repeat step 1 with bodies reversed: test all vertices of j against all edges of i. Abort test whenever $\delta > 0$.

3. It can now be concluded that the maximum δ from 1-2 is $\delta^{ij}_{max} < 0$. A collision for this pair of bodies is confirmed, and we pick as contact normal, the normal associated with δ^{ij}_{max}.

The next step is to decide all contact points. First, we label the edge from which n_c was taken the *reference edge*, and the polygon it is part of the *reference polygon*. The opposite polygon is then labeled the *incident polygon* [Catt06]. Contact points are then picked using either of the following criteria:

- Vertices from the incident polygon being completely inside the reference polygon.

- Points of intersection between the edges of the incident polygon and the *side planes* of reference edge, having negative separation relative the reference edge.

The penetration depth δ_c for every p_c is then defined as the distance to its closest point on the reference edge. Finally, all sets of p_c, δ_c and n_c (n_c is the same for all contacts of this current collision) are used to form new contact constraints, which will eventually be reported to the simulation framework.

3.1.2 Polygon–Circle Collision Test

A simple way to test polygons and circles is first to classify regions of the polygon, and then decide what region the circle is part of. These regions, called *Voronoi regions*, are related to different *features* of the polygon; in 2-dimensional space these features are either *vertex* or *edge*. The region defined by the edge feature is the space extending the the edge outward, like a corridor, in the normal direction, while the vertex feature regions are the open arcs separating the edge regions.

For a circle at position P_c with radius r_c, and a polygon with n vertices P_k and normals N_k, $0 \leq k \leq n$, the test outline is:

1. **Test if circle is within an edge region**. This is the case if for any vertex k of the polygon, the circle is close enough to the edge:

   $$0 \leq (P_c - P_k)^T N_k \leq r_c,$$

   and is within the side planes of it:

   $$(P_{k+1} - P_k)^T (P_c - P_k) \geq 0,$$
   $$(P_k - P_{k+1})^T (P_c - P_{k+1}) \geq 0$$

   We can then compute the contact data:

   - Contact normal: $n_c = N_k$
   - Contact point: $p_c = P_c - N_k (P_c - P_k)^T N_k$
   - Penetration depth: $\delta_c = r_c - (P_c - P_k)^T N_k$

2. **Test if circle is within a vertex region**. If 1 fails, the circle must be part of one of the vertex regions. This is the case for the k:th vertex at which $\|P_c - P_k\| \leq r_c$. It then follows that

   - Contact normal: $n_c = \dfrac{P_c - P_k}{\|P_c - P_k\|}$
   - Contact point: $p_c = P_k$
   - Penetration depth: $\delta_c = r_c - \|P_c - P_k\|$

3.1.3 Circle–Circle Collision Test

Testing circles for contact is straightforward; it is merely a a matter of testing the distance from the center points. If they are closer than their combined radii, they will be in contact with each other.

Consider two circles at P_i and P_j, with radii r_i and r_j: they collide if $\|P_i - P_j\| \leq r_i + r_j$. Contact data is then computed as:

- Contact normal: $n_c = \dfrac{P_i - P_j}{\|P_i - P_j\|}$.
- Contact point: $p_c = P_j + \dfrac{r_j}{r_i + r_j}(P_i - P_j)$.
- Penetration depth: $\delta_c = r_i + r_j - \|P_i - P_j\|$.

3.2 Collision Culling

Since each narrow phase test is computationally expensive, and the global collision detection problem is of time-complexity $O\left(N^2\right)$, there is a strong incentive to implement algorithms limiting the number of exact tests. Many such algorithms are sprung from two fundamental assumptions: *temporal* and *spatial coherence*. Temporal coherence states that between timesteps, the **world disposition will stay roughly the same**, while spatial coherence states that **most pairs of bodies will not collide at any given moment**.

The *Sweep and Prune* broad phase, collision culling algorithm [Bara92] exploits both assumptions. Its fundamental idea is to keep track of the spatial extent of each body (each geometry, if multiple geometries are present). These extents are kept in sorted *interval lists*, one for each individual dimension. From these lists, it can be decided using very few operation whether the intervals of two given bodies are overlapping or not. Only when they do is it necessary to proceed with further testing. When they don't, they can simply be ignored, saving the entire cost of the narrow phase test.

Now, since the content of the simulated world will typically change continuously, the interval lists will hence become outdated. Constant re-sorting seem unappealing; however, under the assumption of temporal coherence, the lists can still be considered to be "almost" sorted, enabling some well suited sorting algorithms to be applied. A common choice is *insertion sort*, which is optimal for partially sorted sequences ($O(N)$ best case) [TrBW09].

4 Implementation

By implementing the theoretical model it can be used in a variety of new ways: it can be tested, visualized, interacted with, applied to real-world tasks and so on. But while providing new opportunities, implementations often bring new problems as well. Most numerical methods have limitations, or at least limited operating ranges, computers have limited abilities to represent numbers etc. Compromises usually needs to be made between model integrity, accuracy, error tolerance, performance or other characteristics. This is not necessarily a problem however; the important thing is to know the purpose of the implementation, and to know how and when its strengths and weaknesses appear.

The purpose of this implementation is to display the aspects of the presented theory in an interactive, real-time environment.

4.1 Development Tools

The programming language picked for the implementation was C++. While, one might argue, it is somewhat less straightforward to use than other available alternatives, it is a very dynamic and flexible language, providing all necessary functionalities. Being object-oriented, the various content of the multibody system, such as bodies and forces, can be organized in an intuitive way. Some data structures utilize the Standard Template Library (STL).
Graphics were created using hardware support through OpenGL [OpGL92]. The OpenGL Utility Toolkit (GLUT) was used for window management [Kilg98]. All code was developed in Microsoft Visual Studio 2005, Professional Edition.

4.2 Basic Classes

The following section is intended to provide an overview of the classes used in the implementation. They are presented in a pseudo-code, pseudo-UML style, aimed not at covering all details exhaustively, but at outlining their principal content and interrelationships.

Standard data types will be used, such as floating points, `float` (32 bit), integers, `int` (16 or 32 bit, signed or unsigned) and booleans, `bool`. Classes with italicized titles are *abstract base classes*, meaning they are used only through classes derived from them. Derived classes have the title format `[DerivedClass]:[BaseClass]`. Abstract classes are practical when several classes share the same characteristics.

The very basic thing needed is some 2-dimensional algebra. We need to be able to represent vectors and matrices, as well as the adherent arithmetic. This

functionality is provided by the `Vector2d` and `Matrix2d` classes, presented below.

`Vector2d`	
Euclidean coordinates	`float x,y`
Euclidean norm	`float norm()`
Dot product	`Vector2d dot(Vector2d)`
Cross product	`Vector2d cross(Vector2d)`
Projection	`Vector2d project(Vector2d)`
Vector addition	`Vector2d +(Vector2d)`
Vector subtraction	`Vector2d -(Vector2d)`
Vector scaling	`Vector2d *(float)`

`Matrix2d`	
Elements	`float m11,m12,m21,m22`
Inverse	`Matrix2d invert()`
Matrix multiplication	`Matrix2d *(Matrix2d)`
Transformation of vector	`Vector2d *(Vector2d)`

Vectors are commonly used; in representing positions, velocities, forces, normals, polygon edges and much more. They are also associated to several vector operations such as dot- and cross-product. Matrices are used primarily for transformation of vectors. They can be instantiated for instance like this:

```
/* initiate as rotation matrix */
Matrix2d(float phi) {
   m11 = cos(phi); m12 = -sin(phi);
   m21 = sin(phi); m22 = cos(phi);
}
/* initiate as uniform scaling matrix */
Matrix2d(float scale) {
    m11 = scale;  m12 = 0;
    m21 = 0;      m22 = scale;
}
```

Whenever suitable, operations are implemented using operator overloading.
The next key building block is the rigid body class. It contain all data relevant for a simulated rigid body, such as the state vector (position, velocity etc) and constitutive properties such as mass and friction, but also information about its geometry.

RigidBody	
Position	`Vector2d X`
Linear velocity	`Vector2d V`
Force accumulator	`Vector2d F`
Rotation	`float R`
Angular velocity	`float W`
Torque accumulator	`float T`
Mass	`float mass`
Moment of inertia	`float I`
Static & kinetic friction	`float friction_s,friction_k`
Restitution	`float restitution`
Static status	`bool isStatic`
Sleep status	`bool isSleeping`
Geometries	`Geometry[] geometries`
Apply impulse at point	`applyImpulse(Vector2d,Vector2d)`
Apply force at point	`applyForce(Vector2d,Vector2d)`

Here, `Geometry` is an instance of either `CircleGeometry` or `PolygonGeometry` (also derived further into for instance `Box`). Each `Geometry` is convex, but concave bodies can be created by using multiple convex geometries – so called *compound geometries*. However, since the properties of the body apply to the body in its entirety, position (center of mass), mass, moment of inertia and similar entities must be calculated with respect to all included geometries. This can be done manually or by using some of the auxiliary functions included in the implementation, such as `PolyCOM()` and `PolyI()` (both assume uniform mass distribution).
The `isStatic`-flag is used for bodies acting as "ground" or other spatially fixed objects. Static bodies will be treated as infinitely heavy and immovable during simulation.

For simplicity, all springdamper configurations are combined into one single class `SpringDamper`, inheriting from a general, two-body force base class `Force`. The `SpringDamper` class can be setup with various linear and angular coefficients according to the analytical equivalent. Force-classes thus "know" by themself everything they need to know in order to calculate, and apply, the force or torque for a given situation, and they do just that when called upon by the `apply()`-method.

Force (abstract)	
Affected bodies	`RigidBody bodyA,bodyB`
Anchor points	`Vector2d rA,rB`
Calculate force and apply to bodies	`apply()`

SpringDamper:Force	
Linear stiffness & damping	`float K_lin,D_lin`
Angular stiffness & damping	`float K_ang,D_ang`
Resting length	`float L`
Resting angle	`float phi`

Additional, auxiliary forces based on `SpringDamper` are used in the implementation, including `MouseTrackingForce` and `CameraTrackingForce`. The former is used to provide user interaction by attaching a springdamper between some body "grabbed" by the user, and the mouse pointer. The latter enables a camera-tracking effect by attaching the camera, represented in the simulation world by a geometry-less rigid body, to other bodies with a springdamper.

The `constraint` base class contains several functionalities common for most constraint, as well as one crucial component: the `solve()`-method, which instructs the constraint to solve itself given its current state. It will be explained further in section (5). The class has the following appearance:

Constraint (abstract)	
Affected bodies	`RigidBody bodyA,bodyB`
Anchor points	`Vector2d rA,rB`
Error reduction parameter	`float ERP`
Calculate impulse and apply to bodies	`solve()`

Each specific constraint is then represented by its own derived class, two of which are the `DistanceJoint` and the `AngularDriver`. In some cases, some of the functionalities of the base class are redundant; purely angular constraints for instance, such as `AngularDriver`, never use the anchor points `rA` and `rB`.

DistanceJoint:Constraint	
Joint length	`float L`

AngularDriver:Constraint	
Target angular velocity	`float target_vel`
Max torque	`float max_torque`

Contact constraints are, as previously mentioned, somewhat special. They are created, used and subsequently disposed of during each single timestep, and they preside multiple internally dependent constraints related to non-penetration and friction. The anchor points of the base class are not set beforehand, but are instead calculated from the contact point.

Contact:Constraint	
Contact point	`Vector2d c_point`
Contact normal	`Vector2d c_normal`
Penetration depth	`float depth`

Note that contact-related information regarding coefficients of friction and restitution is accessed through the body-references.

Finally, there is the `World`-class, containing bodies, constraints, simulation parameters and everything else necessary for the simulation. The simulation is progressed one timestep by its `step()`-method.

World	
Timestep	`float h`
Bodies	`RigidBody[] bodies`
Forces	`Force[] forces`
Constraints	`Constraint[] constraints`
Collision Detector	`CollisionDetector col_detector`
Solver iterations	`int iterations`
Simulation step	`step()`

For practical reasons, a static geometry-less body acting as background is added by default. This way forces and constraints can be attached to "anywhere", simply by attaching them to the background body with some suitable anchor point.

4.3 Simulation

The simulation is progressed using the `step()`-method of the `World`-class. This method performs the following actions in what is considered the *simulation loop*.

1. **Apply forces**.
   Call `apply()` for all forces in `forces`, accumulating forces in the `F`- and `T`-variables of the bodies of the system ($= f_{ext}$).

2. **Update velocities**.
   Update `V` and `W` using numerical integration: $v \leftarrow v + h \cdot M^{-1} f_{ext}$. This leaves us with the linear system of (39).

3. **Run collision detection**.
   Detect collisions and generate contact constraints.

4. **Solve constraints**.
   Call `solve()` for all constraints, one at a time. Repeat as many times as is set by the `iterations` parameter.
   For each `solve()`-call, the constraint k in question is instructed to solve its dedicated two-body i, j-block of (39). This is done internally in the following steps:

   - Calculate impulse magnitude: $\lambda_k = \left(J_k M_{ij}^{-1} J_k^T\right)^{-1} (J_k v_{ij} + d_k)$
   - Calculate impulse: $P_k = J_k^T \lambda_k$
   - Apply to bodies: $v_{ij} \leftarrow v_{ij} + M^{-1} P_k$

   The k:th constraint is thereby satisfied, but may well be violated again by other other constraints acting on these same bodies. By iterating all constraint over and over however, the global solution will converge gradually.

5. **Update positions**.
   Update `P` and `R` using numerical integration: $p \leftarrow p + h \cdot v$.

4.3.1 Sleeping

In many common simulation scenarios the bodies will eventually settle on the ground. Since they then have near-zero velocities and cause no or few new interactions with other bodies, it is tempting to simply stop simulating them in order to household processing power. This kind of actions are called *sleeping*, *freezing* or *deactivation*, and use some predefined mechanism to stop, fully of partially, the simulation of bodies near rest.

This implementation optionally applies a simple, *velocity threshold*-based sleeping strategy, which suspends step 5 in the simulation loop (4.3) for bodies meeting the sleeping criteria. It works as follows: if for some body, the absolute linear and angular velocities stay below some predefined thresholds, ϵ_{lin_vel} and ϵ_{ang_vel}, for some predefined, minimum amount of time ϵ_{t_min}, the flag `isSleeping` for this body is set to `true`. As soon as any velocity threshold is broken, the flag is set to `false` and the body must fulfill the critera anew.
Since only the position-update step is suspended, the actual gain in performance of this strategy is low. The primary gain, rather, is that small velocities, related to "numerical noise" rather than mechanics, are capped away, providing a smoother, more relaxed resting behavior. As such it can be regarded as a form of crude post-stabilization. The tuning of the time and velocity threshold are very

system-dependent, and in many cases geometry dependent. Large bodies for instance, which can undergo significant movement even from low angular velocities, are especially hard-tuned. In some cases, it might not be worth the effort.

4.4 Collision Detection

The collision detection can be set to use either a crude, $O(N^2)$ algorithm, always testing all bodies against all other bodies, or a broad phase culling algorithm using *Sweep and Prune*. For systems with few bodies, the crude algorithm can well outperform the culling algorithm because of its low need for computational overhead, but as the number of bodies grow, the culling algorithm turns more and more advantageous. Overall, collision detection is the major bottleneck of the simulation. Efficient algorithms and fast low-level vector operations are crucial in order to keep the simulation speed up.

4.4.1 Collision Filtering

Occasionally there is a need to prohibit specific bodies, or groups of bodies, from being tested for collision. One instance is when using a hinge joint with anchor points located within the geometries of the bodies, potentially causing an endless conflict between the joint and the non-penetration constraint, unless a collision test exempt were made for the bodies.
One way to resolve this problem is to use bitmasks. Each body is assigned two bitmasks: `CollisionId`, which identifies the collision group this body is part of, and `CollisionFilter`, which contains information about which other collision groups this body can collide with.
Consider the following sample identification masks:

```
/* create collision id for body BOX by assigning the first bit */
unsigned int BOX = 0x1;
/* create collision id for body BALL by assigning the second bit */
unsigned int BALL = 0x8;
```

Now different filters can be setup according to the desired collision behavior:

```
/* collide with everything (32-bit case) */
unsigned int COLLIDE_ALL = 0xffffffff;
/* collide only with BOX */
unsigned int COLLIDE_1 = BOX;
/* collide only with BOX and BALL */
```

```
unsigned int COLLIDE_2 = BOX | BALL;
/* collide with everything except BOX */
unsigned int COLLIDE_3 = COLLIDE_ALL ^BOX;
```

Here, the ^-symbol refers to the "exclusive or"-operator (XOR). Since each collision group occupies one bit, the number of groups is limited by the memory size of the data type. In this implementation, 32-bit unsigned integers are used, thus allowing a maximum of 32 groups.

What happens internally during the actual collision test, is that the id and the filter of every candidate pair is compared bitwise:

```
if (bodyA.CollisionId & bodyB.CollisionFilter)
   /* proceed collision detection...  */
```

4.5 Rendering and User Interface

At the end of each timestep, selected content of the system is visually rendered. This is considered a *frame* of the simulation. The rate by which frames are drawn is called *frame rate*, measured in frames per second (FPS) or Hertz (Hz). Real-time rendering usually refer to a rate of 30-60 FPS.
Though visualization is a key part of this implementation, the sophistication of it is not. The aim is first and foremost to provide useful information about world disposition, forces, joints and other aspects of the simulation. Rendering focused on program information rather than aesthetics is sometimes called *debug rendering*. When applied in an end-user application, such as a computer game, graphics are, according to most programming models, made in a completely separate abstraction layer, usually with far higher geometric complexity than that of the dynamic simulation.
The main runtime user interaction tool is the mouse, which enables the user to "grab" and move bodies (using a springdamper), to move the camera and to add new bodies, springdampers and joints.

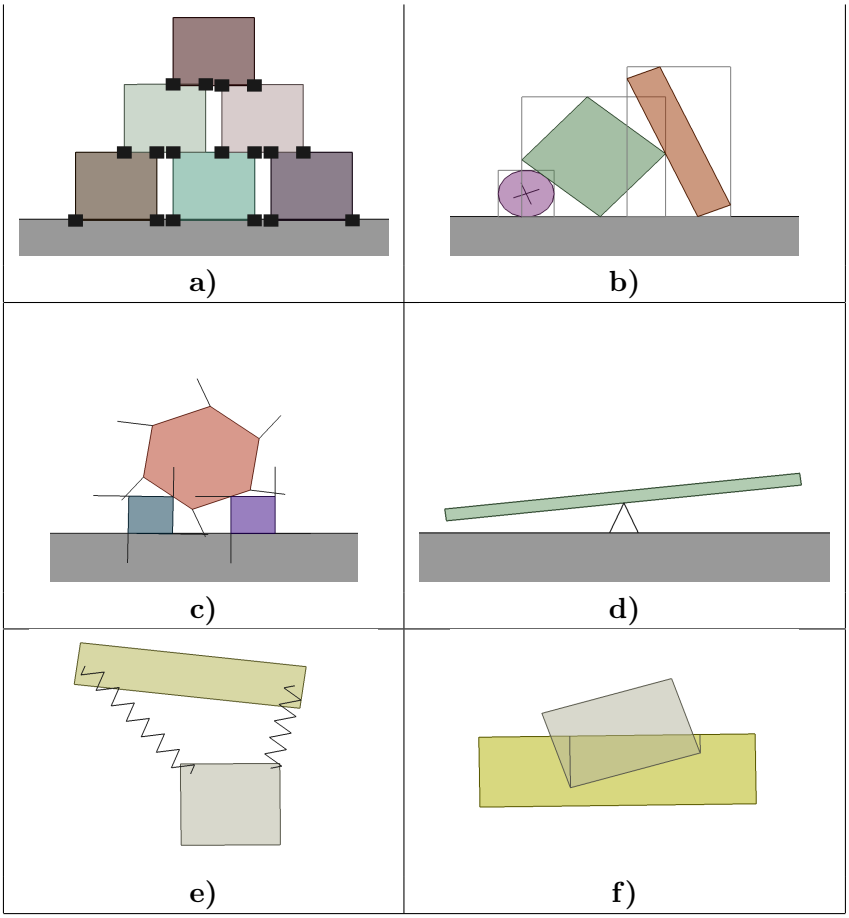


Figure 4.5.1 Rendering samples

a) Contact points
b) Geometry AABB's
c) Edge normals
d) Hinge joint attaching a seesaw to the ground, rendered with "dummy" struts
e) Springdampers
f) Highlight of penetration

5 Testing and Evaluation

5.1 Stacking

Building stacks of aligned, equally sized bodies is a significant challenge (ask any three year-old), putting every aspect of both model and implementation to the test. What makes the task difficult is the fact that errors will propagate up and down through the entire stack, and unlike chain-like multibody structures where the bodies are well connected by constraints, these errors will eventually cause the bodies to misalign enough for the stack to fall over.
The key to the simulation of stacks is first and foremost to keep the solver output consistent: if the system configuration stays roughly the same from one timestep to another, then so should the solution. This is achieved by keeping the input; contact data produced by the collision detection module; as consistent at possible. Second, since errors will nevertheless occur, the goal is to strike a balance between how much error to allow, and how fast to correct it.

Stacks with 20 bodies of dimension 0.5 m and density $\rho = 8.0$ kg/m^2 where tested (sleeping disabled) with different number of solver iterations using the following parameters:

Simulation parameters	
Timestep	$h = 10$ ms
Solver Iterations	10/50/100
Gravitational acceleration	$g = 9.81$ m/s^2
Body Configuration	
Restitution	$e = 0.25$
Friction	$\mu_s = \mu_k = 0.25$
Contact Parameters	
Allowed penetration	$\epsilon_{depth} = 0.002$ m
Velocity for resting contact	$\epsilon_{vel} = 1.0$ m/s
Error reduction parameter	ERP $= 0.4$

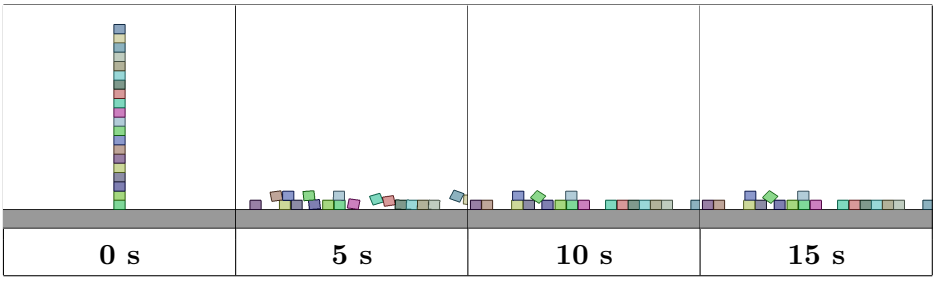


Figure 5.1.1 10 iterations, 0-15 s.

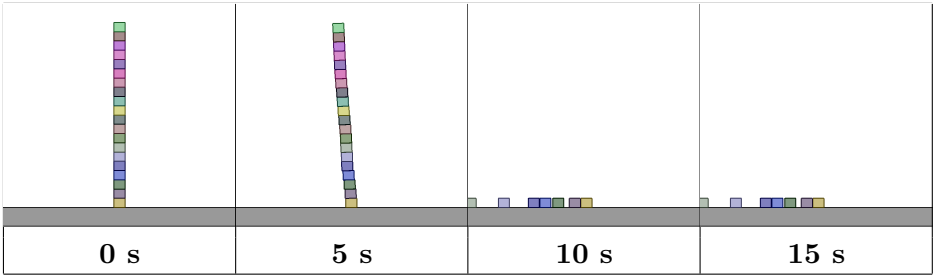


Figure 5.1.2 50 iterations, 0-15 s.

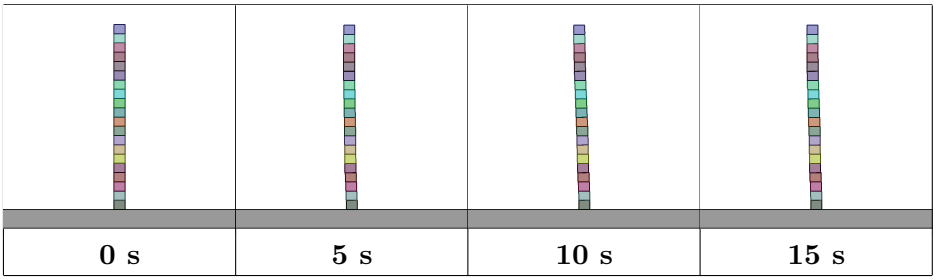


Figure 5.1.3 100 iterations, 0-15 s.

Remarks While falling over quickly at 10 iterations, the stack remains standing for longer and longer at 50 and 100 iterations. The simulation runs at or near real-time performance during all three simulations on a 2 GHz single core CPU, 1 GB RAM computer. During the initial seconds of the simulation, the stack "sways" slightly to the side. At 10 iterations, the stack never recovers from this and falls over, while at 100 and, to some extent, 50 iterations, the sway stops, turns and starts oscillating slowly back and forth. Though all parameters are relevant to this behavior, it can be related particularly to ϵ_{depth} and ERP. The former parameter influences the amplitude of the sway, while the latter influence how fast the sway recovers (its frequency). The objective in this case is to limit this swaying motion enough to keep the stack standing, but in a moderate enough manner to prevent

error stabilization to overshoot and cause sliding.

5.2 Hybrid Joint: Hydraulic Driver

In hydraulic machinery, pressurized fluid is used to transport and transform hydrostatic energy into translational motion. The fluid, typically low-compressible oil, first passes through a hydraulic pump before it is led to an end device located where the desired motion is to take place. One common such device is the *hydraulic cylinder*: a cylindrical barrel with a moving piston.

Under certain assumptions such a device can be modeled by a combination of already presented techniques. The one novelty to account for is the fact that a hydraulic device has different, externally issued operating states, such as "extend piston", "contract piston" and "hold piston". Since each such state requires a separate set of constraints, a state transition framework needs to be implemented.

Real-world to model setup

1. Attachment points: **anchor points $\mathbf{r_i}$, $\mathbf{r_j}$**.
2. **Assume fluid is neither compress- nor expandable**.
3. Limited pump capacity: **max joint force $|\mathbf{f_{max}}|$**.
4. Limited piston velocity (limited fluid flow): **max joint velocity $|\mathbf{v_{max}}|$**.
5. Operating length: **joint length $\mathbf{L_{min}} \leq \|\mathbf{r_i} - \mathbf{r_j}\| \leq \mathbf{L_{max}}$**.

Two constraints are setup to manage this behavior. A length-related constraint C_l and a driver-related constraint $\dot{C}_d$. The driver constraint is straightforward; as long as the device is of legal length, $\dot{C}_d$ is set to the target velocity, with the maximum force as limits. The length constraint is responsible for keeping the device within its legal length, and, importantly, to keep the current length $L = \|r_i - r_j\|$ when the device is on hold. The constraint length, $C_l = L$, therefore must be updated continuously as the piston moves (as long as it moves in a state-consistent direction).

State constraint designation

State	Length constraint		Driver constraint
EXTEND	$L \leq C_l \leq L_{max}$		$\dot{C}_d = v_{max}$,
	`if` $C_l < L$	$\dot{C}_l \geq 0$,	$-\infty \leq \frac{\lambda_d}{h} \leq f_{max}$
		$0 \leq \frac{\lambda_l}{h} \leq \infty$	
	`if` $C_l > L_{max}$	$\dot{C}_l \leq 0$,	
		$-\infty \leq \frac{\lambda_l}{h} \leq 0$	
CONTRACT	$L_{min} \leq C_l \leq L$		$\dot{C}_d = -v_{max}$,
	`if` $C_l < L_{min}$	$\dot{C}_l \geq 0$,	$-f_{max} \leq \frac{\lambda_d}{h} \leq \infty$
		$0 \leq \frac{\lambda_l}{h} \leq \infty$	
	`if` $C_l > L$	$\dot{C}_l \leq 0$,	
		$-\infty \leq \frac{\lambda_l}{h} \leq 0$	
HOLD	$C_l = L$		$\dot{C}_d = 0$,
	`if` $C_l \neq L$	$\dot{C}_l = 0$,	$0 \leq \frac{\lambda_d}{h} \leq 0$
		$-\infty \leq \frac{\lambda_l}{h} \leq \infty$	

The hydraulic driver was tested using a springdamper in a setup shown in figure 4.2.1a-c. The spring force f_{spring} was plotted over time to monitor the actual force generated by the driver for different settings of f_{max}. The springdamper had length $L = 2$ m, $k = 200$ N/m and no damping. Piston velocity was set to $v_{max} = 1$ m/s. The simulation timestep was $h = 10$ ms and the solver used 10 iterations.

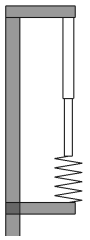
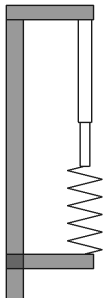


Figure 4.2.1a Driver at HOLD. Spring at rest length.



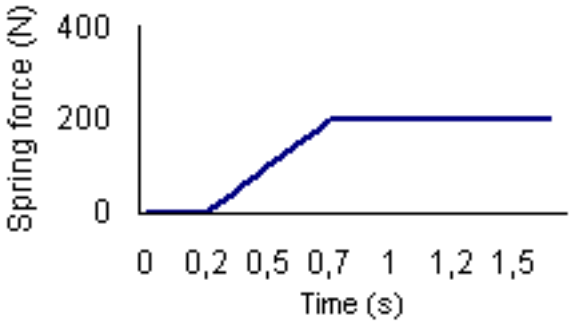
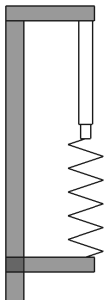


Figure 4.2.1b Driver at CONTRACT, $f_{max} = 200$ Nm.



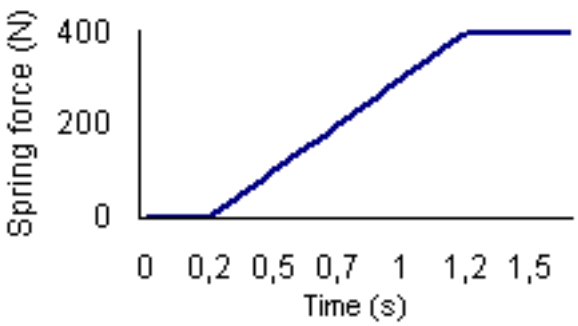


Figure 4.2.1c Driver at CONTRACT, $f_{max} = 400$ Nm.

Remarks The driver appears to deliver what it promises; a linear motion with constant (maximum) velocity and force. As can be seen in the diagrams of figure 4.2.1b-c, the driver manages to extend the spring until the spring force reaches that of f_{max} for the driver, at which point the driver stops. When the driver is subsequently extended, the maximum velocity v_{max} is still held, despite the fact that the spring now acts in the same direction, thus "helping" the driver motion. The driver is rendered to resemble a *telescopic hydraulic cylinder*, a device able to extend in multiple piston stages.

5.3 Conclusion

The thesis presented a study on real-time multibody simulation using an impulse-based system formulation solved by an iterative scheme. It was explained how kinematic constraints can be used as multipurpose tools for various applications such as joints, friction and angular drivers. A contact model was presented, as well as a collision detection module. In addition, an outline of the implementation of the model was presented, including key classes, the solver loop and rendering.

The main advantages of the impulse-based multibody approach used here is that it is fast, lightweight and flexible. By being able to solve constraints block-wise, the solver loop gets very compact, free from heavy matrix operations. Constraints can also be added and removed during simulation. Furthermore, is it convenient to be able to express many key dynamic aspects of the simulation uniformly as constraints; joints, drivers and more; all solved by the same solver.

One weakness is the explicit integrator, which can perform poorly and even cause instability in systems with stiff elements such as springs or contacts with aggressive error-correction. Another weakness is an occasional slow rate of solver convergence; for instance when bodies with large mass-ratios interact. As for collision detection, the performance of the Sweep and Prune broad phase algorithm can dip significantly in certain situations when many bodies move simultaneously (causing disarray in the interval lists).

Most weaknesses can be addressed by making the timestep smaller and increasing the number of iterations used by the solver, but this of course takes its toll on the performance. Other stability-related strategies include adding damping, capping velocities or other system restrictions, depending on the problem.

It turns out that even with a well theoretically founded model, it is still a significant challenge to implement it. Each algorithm and numerical method has its limitations, urging compromise and delicacy. This is especially the case when performance is of high priority. And however robust and sophisticated the implementation, in the end it will nevertheless be a matter of some degree of tuning. As was shown earlier in this chapter, each setting, be it one with stacks or with vehicle models, poses its own unique challenge and calls for its own unique system calibration.

5.3.1 Future Work

Some suggestions for future work:

- **Extending to three dimensions**. This pose new challenges especially regarding rotations, which would then be represented by matrices, or possibly quaternions, instead of scalars. Three new degrees of freedom also means

new possibilities in removing them – meaning new types of joints, such as the Cardan joint.

- **Warm starting**. Instead of disposing of contact points after each timestep, they can be stored and used to provide better initial guesses for the solver at later times (as long as the contact persists) [Erle04].

- **Optimization in general**. Program code is, it seems, ever evolving, always with areas for improvement. In this case, a smarter, more efficient collision detection can be implemented, as well as a faster memory management system. An interesting method to reduce redundant contact points, which are not only costly but cause unwanted over-determined systems, is presented by [Mora04].

Bibliography

[AnPo97] Mihai Anitescu and Florian A. Potra. Formulating Dynamic Multi-Rigid-Body Contact Problems with Friction as Solvable Linear Complementarity Problems. *International Journal for Numerical Methods in Engineering*, Vol. 55, No. 7, 2002.

[Bara91] David Baraff. Coping with friction for non-penetrating rigid body simulation. *Computer Graphics* 25(4): 31-40, 1991.

[Bara92] David Baraff. *Dynamic Simulation of Non-Penetrating Rigid Bodies*, (Ph. D thesis), Computer Science Department, Cornell University, pp. 5256. 1992.

[Bara94] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. *Computer Graphics*, 28 (Annual Conference Series):2334, 1994.

[Bara96] David Baraff. Linear-time dynamics using Lagrange multipliers. *Computer Graphics Proceedings*, Annual Conference Series: 137-146, 1996.

[Baum96] J. Baumgarte. Stabilization of constraints and integrals of motion in dynamical systems. *Computer Methods in Applied Mechanics and Engineering*, (1):116, 1972.

[Byaz08] Mark Bayazit: http://mnbayazit.com (2009-11-05)

[Catt05] Erin Catto. *Iterative Dynamics with Temporal Coherence.* http://www.gphysics.com/downloads (2009-11-05)

[Catt06] Erin Catto. *Fast and Simple Physics using Sequential Impulses.* GDC 2009 presentation. http://www.gphysics.com/downloads (2009-11-05)

[CaPS92] R.W. Cottle, J.S. Pang and R.E. Stone. *The Linear Complementarity Problem.* Academic Press, Inc., 1992.

[Eric05] Christer Ericson. Real-Time Collision Detection. Morgan Kaufmann Publishers, 2005.

[Erle04] Kenny Erleben. *Stable, Robust, and Versatile Multibody Dynamics Animation*. Ph.D. Thesis, University of Copenhagen, 2004.

[ESFu98] Edda Eich Soellner and Claus Führer. *Numerical Methods in Multibody Dynamics*. Teubner, Stuttgart, 1998.

[Kilg98] Mark Kilgard. The OpenGL Utility Toolkit. http://www.opengl.org/resources/libraries/glut (2009-11-05)

[Grib08] Carl Johan Gribel. Multibody simulation demos: http://www.youtube.com/user/robbe21 (2009-11-05)

[Löts82] P. Lötstedt. Mechanical systems of rigid bodies subject to unilateral constraints. *SIAM Journal on Applied Mathematics*, 1982.

[Mora04] Adam Moravanszky and Pierre Terdiman. Contact Reduction for Dynamics Simulation. In *Game Programming Gems 4*, pages 253-263. Charles River Media 2004.

[OpGL92] OpenGL – The Open Graphics Library. http://www.opengl.org (2009-11-05)

[Smit04] Russel Smith. Constraints in rigid body dynamics. In *Game Programming Gems 4*, pages 241-251. Charles River Media 2004.

[StTr96] D. E. Stewart and J. C. Trinkle. An implicit time-stepping scheme for rigid body dynamics with inelastic collisions and coulomb friction. *Internat. J. Numer. Methods Engineering*, 1996.

[TrBW09] Daniel J. Tracy, Samuel R. Buss and Bryan M. Woods. *Efficient Large-Scale Sweep and Prune Methods with AABB Insertion and Removal*. 2009 IEEE Virtual Reality Conference.