Rigid Body Simulation
and
Evolution of Virtual Creatures

Daniël Fontijne

Project Supervisors: dr. Jaap Kaandorp
prof. dr. Peter Sloot

Master's Thesis Artificial Intelligence,
Specialization Autonomous Systems

august 2000

University of Amsterdam,
Faculty of Science,
Section Computational Science

Figures 2.5 and 2.6 copied from [Mir97] with permission of Mitsubishi Electric Information Technology Center America; copyright 1997 Mitsubishi Electric Information Technology Center America.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Aim of the work

The two main goals of the work performed for this thesis were to create a rigid body simulator and to evolve virtual creatures inside the virtual world created by that simulator. Initially the emphasis was on the evolution, but as time passed simulation itself became the major subject.

A rigid body simulator is able to simulate the behavior of a system of rigid (non-deformable) bodies. Examples of rigid bodies or materials whose real-world behavior can be approximated by such a simulator are metal, glass, stone, ping pong balls, ice, hard foam and wood. Of course all of the previously mentioned bodies deform or fracture to some degree when they collide or when enough pressure is exerted on them; a rigid body simulator operating under the assumption that all bodies in the simulation are totally rigid can only approximate what would really happen in the real world.

In order to approximate the bodies and materials such as those listed above, we must be able to specify a number of properties of the bodies in the simulator. Some reasonable properties which were to be implemented are friction, restitution, mass and shape. With these properties we hope to be able to simulate a large spectrum of different rigid bodies in nature, including the above mentioned.

Another goal was to be able to connect the bodies together via joints. This would allow the simulator to capture a much larger part of the systems present in the real world. Many physical systems contain (reasonably rigid) joints like vertebrates, robots and other machines.

Getting the simulator to work in real time for modestly complex scenes was another important goal. Algorithms and optimizations to make this possible had to be found and implemented.

During the development of the simulator the emphasis was on *qualitative* correct behavior of bodies in the simulation, not on the quantitative results or it's ability to *predict* the future. The goal was to create a complex dynamic environment in which the virtual creatures were to be evolved. This is not to say that we assumed 'what looks right is right'. The outcomes of the simulations

had to be plausible in the real world, assuming one could construct the right bodies, structures and mechanisms. No comparisons between real-world and simulated behavior were performed.

An important part of a rigid body simulator is the collision detection module. This module can determine the exact distance between two bodies. This information is required keep the bodies from penetrating each other by applying impulses and forces.

Finally the goal which actually fueled the author's interest in creating a rigid body simulator was to evolve artificial creatures (or robots) to perform certain tasks in the virtual world which we can 'spawn' using the simulator. The target was to equal or even surpass the work of Sims ([Sim94a] and [Sim94b]). Through evolution both morphology and behavior of the creatures were to be evolved.

1.2 Previous work

Using genetic algorithms and artificial evolution (see for e.g. [Hol75], [Gol89] or [Mic96]), the 'digital equivalent' of Darwin's *survival of the fittest*, to evolve creatures or robots 'living' in simulated worlds has been a research topic for some time. A number of researchers have successfully evolved locomotion behavior for simulated 2D stickfigures ([dG90], [NM93] and [PF93]). Sims ([Sim94a], [Sim94b]) evolved the behavior of artificial creatures for a number of tasks, like walking, jumping and swimming. [XTu96] developed artificial fish. [Rey94] evolved vehicles for the 'game of tag' (a predator-prey situation). [Mir99] also simulated a predator-prey situation where the behavior of three-wheeled vehicles had to be evolved. Ongoing research at [LP00] reports about the combination of rigid body simulation and rapid prototyping to evolve *real* machines.

A large body of research on collision detection also exists, with algorithms becoming faster and smarter and reaching their theoretical time complexity limits for certain classes of objects in recent years. While Hahn [Hah88] reports using bounding boxes to cull most possible collisions and then using a quadratic time complexity algorithm to compute the distance between two objects, Baraff [Bar90] already describes a much faster algorithm which caches the results from previous invocations. The algorithm is able to determine (non-) penetration in near constant time. Lin [Lin93] improved on this by creating the *near* $O(1)$ Lin-Canny algorithm which can compute the exact distance between two objects. It works by 'walking' over the surface of objects in search for the two closest points. [Mir97] enhanced the Lin-Canny algorithm, making it more robust and somewhat more efficient.

A total collision detection solution for large scale environments called *I-Collide* is described in [DLMP95]. Such a package is not only able to compute the distance between two objects, but also to efficiently 'cull' pairs of objects which could not possible collide with each other within a certain interval. *V-Collide* [HLCGM97], the successor of I-Collide is another collision detection package aimed at VRML. [Mir96] describes an efficient method to cull pairs of

objects, based on the work of Overmars [Over94].

Methods for simulating articulated bodies (bodies connected through joints) have been a research topic for a long time. Two major approaches are those based on (Lagrangian) multipliers and those based on reduced coordinates descriptions of the systems of articulated bodies. Early methods had $O(n^3)$ time complexity, usually due to inverting some (large) matrix. Featherstone [Fea87] invented an $O(n)$ reduced coordinates method which (despite it's complexity) is very popular. Baraff [Bar96] presents an $O(n)$ multiplier method. A (rather theoretical) presentation of several algorithms involving articulated bodies is given in [Lil93].

Some literature on rigid body simulators also exists. Hahn describes his rigid body simulator in [Hah88]. [Bar92] investigates the problems involved in rigid body simulation as a whole. Mirtich gives an extensive presentation of this impulse-based simulator in [Mir96], from which the author learned a lot.

1.3 Approach

Collision detection is performed using a two-phase algorithm. The first phase (the *broad* phase) culls out all pairs of bodies which could not possibly collide during a specified interval (e.g. 0.05 seconds) or the next integration step. The algorithm selected for this phase is the *hierarchical hashing* algorithm [Mir96]. It divides space into a hierarchy of tiles. Bodies are checked for intersection with tiles and hashed into a table. This results in a very fast filter which lets only possibly colliding bodies pass. The second phase (the *narrow* phase) determines the exact distance between two bodies. The V-Clip algorithm, a descendant of the Lin-Canny algorithm [Lin93] is used for this purpose. In typical dynamic simulation situations this algorithm achieves *near constant* $O(1)$ performance, almost independent of the complexity of the surface description of the bodies.

Rigid body simulation is done using an impulse based approach [Mir96]. In an impulse based simulator all contact is modelled using trains of (micro-)collisions. This has the advantage of one unified method for handling both collisions and contact and thus it is relatively easy to implement. The disadvantage is that it can not efficiently handle certain types of contact (such as high stacks of bodies). Impulses are computed often and thus an efficient algorithm was required to achieve real time performance; impulses are computed using an algebraic method [CR98]. For integration a fourth order RungeKutta (step doubler) integration algorithm is used. The Featherstone algorithm (with extensions) is used to model bodies connected by joints.

An attempt was made to implement the virtual creatures' description, mutation, crossover and evolution as much as possible the way Sims ([Sim94a] and [Sim94b]) did. The creatures' genotypes are described using an object oriented method. Evolvable objects contain information and other objects which describe how a creature must be synthesized from it's genotype. Objects know (through class member functions) how to mutate, grow and develop.

1.4 Structure of the thesis

The rest of this thesis is divided into four chapters.

Chapter 2 deals with collision detection, an important problem in rigid body simulation. Two algorithms are described and discussed. The Voronoi-Clip algorithm (the narrow phase) determines the distance between two objects; hierarchical hashing (the broad phase) is an efficient algorithm to filter out objects which can not intersect during a certain interval. The chapter also gives the results of several experiments done with the *bounce* program. The chapter ends with a discussion and conclusion of the algorithms and the experiments.

The other parts of the rigid body simulator are described in chapter 3. It treats subjects like the physical laws governing the simulation, spatial algebra, simulation of bodies connected by joints, the computation of mass and inertia of arbitrarily shaped bodies, the computation of impulses and how the collision detection module and all the other modules fit together in the simulator. It also presents the results (both qualitative and quantitative) of the experiments that were conducted. The chapter ends with a discussion and conclusion of the results obtained with the simulator.

Chapter 4 explains how an attempt was made to evolve virtual creatures using the simulator described in chapters 2 and 3. It details how the morphology and controllers of the creatures are evolved and gives the results, discussion and conclusion of the evolution experiments.

The thesis ends with chapter 5, an overall discussion and conclusion. As mentioned above, discussions and conclusions about the three individual chapters (collision detection, rigid body simulation and evolution of virtual creatures) are given at the end of each chapter.

A full color appendix and a CDROM containing frames and animations of the results achieved accompany the thesis.

Chapter 2

Collision Detection

2.1 Introduction

Collision detection is an essential part of every rigid body simulator. The most basic law or rule of such a simulator should be that rigid bodies do not penetrate each other. To guarantee that this law is not violated one must know whether or not bodies are penetrating. To compute an approximate lower bound on the time when two bodies are going to penetrate, the distance between those bodies must be know. To compute and apply physically correct impulses or forces, the two closest points on two bodies must be known.

Inside a rigid body simulator, it is sufficient to represented bodies by their envelopes or surfaces. When a rigid body simulation is initialized, the mass and mass distribution of bodies must be computed. *During* simulation, a representation of a body is required only to compute it's distance to other bodies. The surface description of a body can be used to compute it's mass distribution and for collision detection. Because collision detection is more difficult and computationally expensive when smoothly curved surfaces (e.g. build from nurbs or splines) are used, the surfaces are usually approximated by polyhedrals. Polyhedral objects consist of vertices, edges and faces. In this context, vertices, edges and faces are collectively referred to as 'features'.

Besides the description of the surface of a body, two more properties are required to compute the distance between bodies: the position and orientation. To compute the distance or closest points between two bodies, only *relative* positions and orientations are required. This can be exploited to save computation time because only coordinates of the features of one body have to be transformed while coordinates of the features of the other body remain fixed in the body frame.

Collision detection is a computationally expensive operation. Early rigid body simulators sometimes spent more than 95% of their time on collision detection [Hah88]. They often used (object orientated or axis aligned) bounding boxes to determine whether two bodies *could* intersect after which an $O(n^2)$ algorithm (where n is the number of features) was used to check for penetration in case the bounding boxes were intersecting. An example of such an algorithm

is checking every edge of one body against every face of other body (Hahn's method).

Recently a number of smart and fast algorithms have emerged which provide (much) faster than $O(n^2)$ methods to compute the precise distance between two bodies e.g. [GJK88] [Bar92], [Lin93], [Mir97]. They provide $O(n)$ or $O(\sqrt{n})$ time complexity if they do their work from scratch but are able to achieve near constant time when they can exploit the *spatial and temporal coherence* in the position and orientations of bodies. In rigid body simulators and similar applications the (relative) position and orientation of bodies usually change only slightly between time steps. By using the previously closest features of two bodies as a starting point and updating these for the new (slightly different) situation, collision detection can be done very efficiently.

For my simulator we implemented the Voronoi Clip algorithm described in [Mir97]. Voronoi clip (or V-Clip for short) is an enhancement of the Lin-Canny algorithm described in [Lin93], which we initially implemented and used. Both algorithms 'walk' from feature to feature over the surface of the two bodies in search of the closest points. The difference between them is in the algorithms used to decide to which feature to go next and when the two closest points have been found. V-Clip is an enhancement of the Lin-Canny because these algorithms are more robust. Also, V-Clip can detect penetration of bodies much more easily than the Lin-Canny algorithm which gets caught in infinite loops which must be detected so penetration can be reported.

With some extensions the V-Clip algorithm can also report whether two features are (still) in *contact*. This can be used to track the closest points between two *features*, which is useful for contact handling in rigid body simulators.

Even with these fast algorithms at our disposal, collision detection can still be too computationally expensive, because even though we can compute the distance of a pair of objects in near $O(1)$ time, there are still $O(n^2)$ pairs of objects inside a simulation with n bodies. To solve this problem, collision detection is often split into a *narrow* and a *broad* phase. The broad phase culls most of the pairs of bodies (collision pairs) which could not possibly collide. The narrow phase computes the exact distances between the collision pairs that were not culled by the broad phase. For the broad phase collision detection we use a modification of the hierarchical hashing algorithm described in [Mir96].

Measurements are reported in the results section. It describes empirical results on the complexity of both algorithms made with the 'bounce' benchmark program. Bounce is a demonstration, benchmarking and test program. It demonstrates how to use my implementation of both algorithms and show (interactively and graphically) what the algorithms are doing for the purpose of education. It can also be used to do benchmarks by turning off the interactive and graphical elements.

2.2 Representing Bodies inside the Simulator

A short description of how the shape of bodies in the simulator is represented inside the simulator is given in this section. The shape of bodies is required

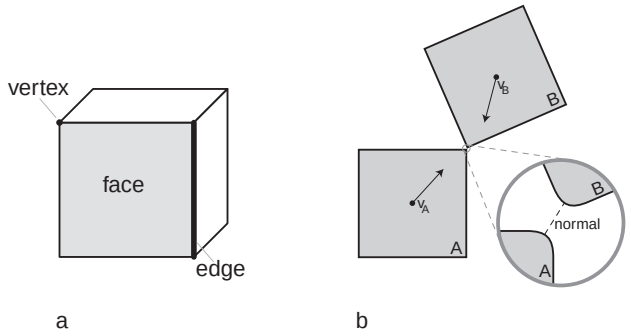

Figure 2.1: a) Polyhedral representation of a cube. b) Computing normals for polyhedral surfaces; the magnification at the bottom right shows that edges an vertices are assumed to be geometrically smooth.

for two purposes by the simulator: to compute the mass properties (mass and inertia) and to compute the distance between two bodies. For both these problems efficient algorithms have been developed which work with a *polyhedral* representation of the bodies. This representation approximates the surface of the 'real' bodies with a number of polygons or faces. Any surface can be approximated to arbitrary degree of accuracy with the polyhedral representation. The level of detail used to represent the bodies is usually a trade off between accuracy and speed or memory requirements; the more accurate the polyhedral model is, the more data is required to describe it.

Polyhedral objects are made from *vertices*, *edges* and *faces* (also called *polygons*). Vertices are points. An edge is a line between two vertices. A face is a plane bounded by three or more edges. Collectively vertices, edges and faces are referred to as *features*. See figure 2.1a. It shows a cube as it would be modelled using the polyhedral representation. 8 vertices are present at each of the corners of the cube, 12 edges connect all vertices and 6 faces are required to cover each side of the cube.

The non-smooth nature of polyhedral objects causes a problem during the computation of impulses or forces. To compute an impulse or force, the normal at the contact point of the bodies involved is required. Suppose two 2D bodies 'A' and 'B' have velocities v_A and v_B such that they collide at one of their vertices (as depicted in figure 2.1b). What would be the normal of the surfaces of both bodies at the contact point? As can be seen in the magnification in 2.1b we assume that both bodies are geometrically smooth at the contact point (the corners of both bodies A and B are round if we look close enough). This causes dashed line between the closest points on A and B to be parallel to the surface normal at the closest points. We can thus easily compute the normal of both objects at the contact point; it is parallel to vector from the closest point on A to the closest point on B. This assumption is also made when (for instance) a vertex collides with a face, or an edge with another edge.

2.3 Broad Phase Collision Detection

2.3.1 Introduction

The task of the broad phase collision detection in rigid body simulation is to filter out most pairs of objects which can not collide within the next time step, at a very low computational cost. The algorithm described here (a modification of the algorithm described in [Mir96]) has $O(n+c)$ time complexity, where n is the number of objects in the simulator and c the number of possibly colliding objects.

Axis aligned bounding boxes are used cull collision pairs. These boxes must be computed every time the algorithm is invoked from the current state of the dynamics simulator (shape, position, orientation, velocity and acceleration of the objects) and from the lower bound on the time to the next collision. It is not always possible to compute an exact bounding box. Details on how to compute the bounding box vary per type of object (e.g. free, constrained, controlled). Bounding boxes of objects which have a fixed position and orientation (surfaces, walls) need to be computed only once. For objects in free ballistic flight an exact bounding box can be computed from the dynamics equations. This is much more difficult for objects connected by joints or objects controlled by separate controller elements such as springs or dampers. For these objects only approximate bounding boxes can be computed. After a time step have been simulated it must be checked whether these approximate bounding boxes have been violated[1]. If a bounding box has been violated, the size of the bounding boxes is adjusted according to the (now known) trajectory of the objects. If this causes new collision pairs to be created, the last simulation step must be undone and evaluated again.

2.3.2 Hierarchical Hashing

Using bounding boxes saves some time because checking for intersection of two axis aligned boxes is so simple. But still $O(n^2)$ checks must be performed, which is too much for simulations involving many objects. To solve this problem we could partition 3D space into cubes or *tiles* and represent these in an (infinitely) large 3D array. See figure 2.2. It shows three bounding boxes a, b and c. The tiles they intersect are marked by stripes in three different directions. If two objects intersect the same tiles, they are reported as possibly colliding, even if their bounding boxes do not intersect. If a bounding box intersects a tile, it stores a pointer in the array. If two or more boxes store a pointer at the same position in the array, their bounding boxes might intersect, and thus more precise (narrow phase) collision checks must be done. However, there are two major problems with this approach.

The first problem is the amount of memory required to represent the partitioning of 3D space. Even of we would choose a (reasonable) limit on the size of the world, we will still require a lot of memory. Since most of the space in the world will not be filled at the same time a lot of memory would be wasted

[1] Actually this is done inside the numerical integration loop for each sub step.

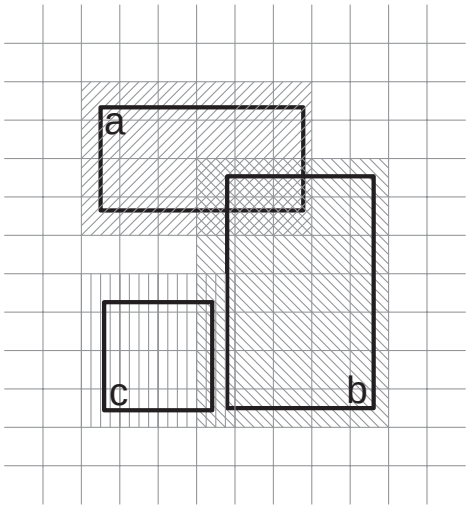

Figure 2.2: The intersection of three objects with the tiles of a partition of a 2D space. Objects a & b and b & c would intersect according to the tile intersection test.

to represent empty tiles. This can be solved by using a *hash table*. Assuming perfect hashing, a hash table requires only an amount of memory proportional to number of tiles intersected by bounding boxes. Every tile which is occupied by a bounding box is hashed into a hash table. The hash key used is composed from the 3 dimensional coordinates of the tile. If one tile is intersected by two or more boxes, they will hash into the same bucket and it is known that these two bounding boxes may be intersecting as well.

The second problem is how to choose the size of the tiles. If the tiles are too small, bounding boxes will intersect many tiles and thus require more processing time and memory. If the tiles are to large, many bounding boxes will intersect the same tiles, which will cause more (relatively expensive) narrow phase collision checks. The problem is even worse when the sizes of the bounding boxes vary (in time and per object), because then there is no optimal tile size. A good solution is to use multiple tile sizes at the same time; a hierarchy of partitions. For each object an optimal tile size (their *native tile size* or *resolution*) is computed (based on the expected size of it's bounding box). By using multiple resolutions, large objects can be mapped to large tiles, small objects to small tiles, etc.

This solution introduces a new problem: intersections of bounding boxes of objects which are not mapped to the same tile size will not be reported. This can be solved by mapping objects not only to their native resolution, but also to all lower resolutions. If two objects share a tile at a certain resolution, and that resolution is native to at least one object, they can be reported as possible intersecting.

Figures 2.3a and 2.3b show an one dimensional example. Five one dimen-

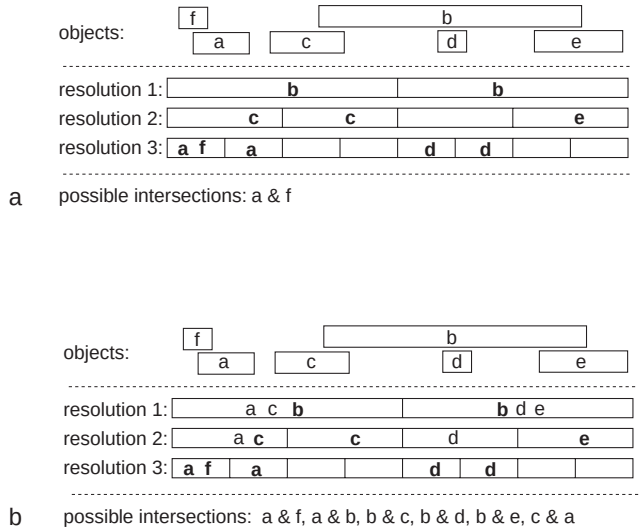

Figure 2.3: Hierarchical hashing. Figure a shows the objects mapped only to their native resolution; not all possible intersections are detected. In figure b the objects are mapped to both their native and all lower resolutions; all possible intersections are detected.

sional 'bounding boxes' (line segments) of varying sizes must be checked for intersection. Three tiles sizes or resolutions are used. Every resolution is twice as large as it's previous one. Because boxes a, d and f are small, they are mapped to the highest resolution (resolution 3). Boxes c and e fit best at resolution 2. Box b is so large it can only be mapped efficiently at resolution 1. Only one intersection is be detected in figure 2.3a because only boxes a and f are mapped to the same tile at the same resolution. Thus it is important to map box both to their native and to all resolutions lower than their native resolution. In figure 2.3b all (possible) intersections are detected. Note that some boxes which do not intersect are reported as possibly intersecting. These could be filtered out by real bounding box intersection tests.

Combining hashing and mapping bounding boxes to hierarchical set of partitionings is called hierarchical hashing. The algorithm can check for all possible intersection between n objects in $O(n + c)$ time, where c is the number of intersections reported.

The reader might notice that hierarchical hashing is very similar to using an octtree. Hierarchical hashing has several advantages over octrees in this application. First of all octrees can cover only the area of space near their origin efficiently. As objects move away from the origin, they are forced to be mapped to higher resolutions. Hierarchical hash tables allow an infinite space to be mapped at the desired resolution. Secondly hierarchical hash tables have better time complexity properties compared to octrees, especially when the hash tables are updated as explained below.

Mirtichs original formulation [Mir96] of the hierarchical hashing algorithm expected the user to choose two constants α and β and allowed for arbitrary tile sizes ρ as long as they satisfied the following condition:

$0 < \alpha < 1$
$\beta >= 1$
$\rho_1 > \rho_2 > \rho_3 > ... > \rho_n > 0$
and such that for each box X there exists an integer$1 <= k < n$ with
$\alpha <= size(X)/\rho_k <= \beta$ (2.1)

In our implementation we fixed α to 0.5 and β to 1.0, because that's the most obvious choice for most applications and to make the implementation somewhat simpler; every tile at resolution i is split up into 2^d (d the dimension of the world) smaller tiles at resolution $i + 1$.

In what follows, bounding boxes will be described using $(x^- y^- z^-)$ and $(x^+ y^+ z^+)$ to specify the minimum and maximum coordinates of a bounding box. ρ will be used to denote the tile size.

The optimal tile sizes are computed by searching for the two objects with the minimal (s_{min}) and maximal (s_{max}) *expected* bounding box size. The tile size ρ_m for the highest resolution is set to the minimum expected bounding box size. The tile size ρ_1 for the lowest resolution is set such that $(2^{m-1}) * \rho_m >= s_{max}$ for some integer $n > 0$. Thus

$$m = ceil(log_2(s_{max}/s_{min})) + 1; \tag{2.2}$$

Mirtich [Mir96] picks the resolution for each bounding box according to (1). Our implementation however picks the resolution based on the number of tiles it will occupy at initialization time . It starts searching at the highest resolution and computes the number of tiles n intersected by a bounding box according to

$$\begin{aligned} n = & \left(ceil\left(\frac{x^+}{\rho}\right) - floor\left(\frac{x^-}{\rho}\right)\right) * \\ & \left(ceil\left(\frac{y^+}{\rho}\right) - floor\left(\frac{y^-}{\rho}\right)\right) * \\ & \left(ceil\left(\frac{z^+}{\rho}\right) - floor\left(\frac{z^-}{\rho}\right)\right) \end{aligned} \tag{2.3}$$

The resolution is decreased until $n <= t$ where t is a (user-specified) threshold. At runtime this process is repeated. Objects are thus allowed to change native resolution. By giving the user (dynamic) control over t higher performance can be achieved.

In rigid body simulators usually a number of objects exist which are fixed (e.g. the floors and walls). These *fixed* objects will be hashed with a higher threshold t than *normal* objects which *do* move. Fixed objects are often larger

than the other objects in the simulation and using a $t_f > t$ prevents them from being hashed at a very low resolution. If large fixed object were hashed at a low resolution, a lot of possible intersections would be reported. There is almost no runtime penalty for hashing fixed objects at a higher resolution because they never have to be updated.

The total number of tiles n required per object at all resolutions depends on the resolution r it is hashed at and the threshold t. If $t \leq 2^3 = 8$, then the upper bound for $n = 8 * r$. If $t > 8$, the exact upper bound is more difficult to compute but it is always $\leq tr$. The total number of tiles intersected by objects at all resolutions is $O(tr)$. If we call R the ratio $\frac{r_{max}}{r_{min}}$ (r_{max} is the maximum expected radius of all objects, r_{min} the minimum expected radius), then the total number of tiles intersected by objects at all resolutions is $O(log(R)t)$. Theorem 1 ([Mir96]) gives the time complexity of the hierarchical hashing algorithm.

Theorem 1. *Let n be the number of objects, c the number of possible bounding box intersections. Treating R and t as constants, assuming perfect hashing, a hierarchical hash table will report all possible intersections in $O(n + c)$ time.*

Proof: The number of tiles that must be set per for each object depend only on R and t, thus it is constant. It follows that the total number of tiles processed for all n objects is $O(n)$. The cost for reporting c possible intersections is $O(c)$.

2.3.3 Exploiting Coherence in the hierarchy hashing algorithm

In rigid body simulators the position of objects usually changes only slightly between time steps. A performance increase could be achieved by exploiting this coherence. Instead of building the hierarchical hash table from scratch, the tiles for every bounding box are updated. Tiles which are intersected by the new bounding box but not by the old bounding box are added. Tiles which are no longer intersected are removed from the hash table. For this method to work, the number of tiles shared between each pair of two objects must be remembered between updates.

As soon as two objects share a tile, a *collision pair* is created. A collision pair contains a reference to both it's objects and a *shared tiles counter*. The shared tiles counter counts how many tiles are shared at the lowest native resolution of the two objects. As long as the number of shared tiles larger than 0, the pair continues to exist. When it drops to 0, no tiles are shared anymore and the collision pair is deleted. Collision pairs are stored in a separate hash table so they can be accessed in constant time (assuming perfect hashing) and require only O(c) memory, where c is the number of active collision pairs. Mirtich ([Mir96]) uses a 2 dimensional array to store the shared tiles counters, requiring $O(n^2)$ memory, where n is the number of objects in the simulation.

To prevent rapid creation and deletion of collision pairs, our implementation allows for some *hysteresis*. Consider a ball bouncing on a surface. As the ball moves up and down, the bounding box of the ball moves in and out of the bounding box of the surface. This might cause rapid creation and deletion of a collision pair consisting of the ball and the surface. It is more efficient to let the collision pair exist for a few more updates after the bounding boxes

no longer intersect. When the shared tiles counter drops to zero, instead of deleting the pair, a reference to the collision pair is set in a separate array (the *hysteresis array*). The collision pair is given a *hysteresis counter* and on every update this counter is decreased. When it drops to 0, the pair is finally deleted. If it's shared tiles counter increases before it's hysteresis counter is 0, the pair is removed from the hysteresis array. Consider the bouncing ball again. If it's bounding box does not separate itself for more than 'hysteresis count' updates from the bounding box of the surface, the collision pair consisting of the ball and the surface is never deleted.

To prevent rapid jumping of an object's native resolution between two neighboring resolutions, some hysteresis is present in the algorithm which changes the resolution of an object as well.

If an object's bounding box is so large that it will require more than four times the recommended number of tiles per object to hash it at the lowest resolution, it will change it's state to be *all over*. The object's tiles are removed from the hash table and pairs are formed with all other objects in the library.

2.3.4 Algorithm summary

The hierarchical hashing algorithm uses two hash tables. The actual hierarchical hash table is used to store the tiles intersected by bounding boxes. The hash key generated for a tile is constructed from it's x, y and z coordinates and the resolution at which the tile is hashed. The other hash table is used to store the close counters. The close counters count the number of tiles which are shared between two objects. The hash keys used for the close counters are made from the id's of the two objects involved.

The hashing algorithm uses two arrays. One array is used to store pointers to all objects. The other array, the hysteresis array, holds pointers to pairs of objects which are no longer colliding, but have to be kept active for a short interval to prevent rapid creation and deletion of collision pairs.

When an object is added to the hierarchical hash table, tiles are hashed for every tile it's bounding box intersects at every resolution less or equal than it's native resolution. The native resolution of an object is the resolution at which it will intersect a maximum of n tiles, where n is a user specified parameter. When an object is removed from the hash table, all tiles for that object are removed from the hash table.

When a tile is hashed to a certain hash bucket in the hierarchical hash table, the algorithm checks every other tile in that hash bucket. If it finds a tile in the bucket with the same position and resolution, and either one of the tiles is hashed at the native resolution of the objects they belongs to, the close counter for that pair of objects is incremented. If the close counter was zero, a new collision pair is created.

When a tile is unhashed from a certain hash bucket in the hierarchical hash table, the algorithm checks every other tile in that hash bucket. If it finds a tile in the bucket which hash the same position and resolution, and either one of the tiles is hashed at the native resolution of the objects they belongs to, the close counter for that pair of objects is decremented. If it reaches zero, a pointer

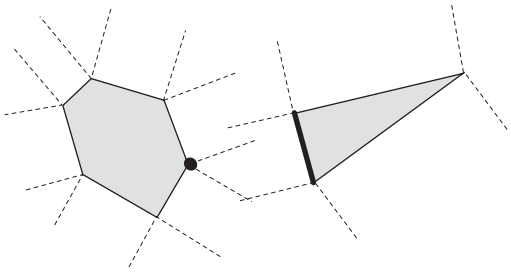

Figure 2.4: Voronoi regions, Voronoi planes (dashed lines) and globally closest features (bold) of two convex 2D polyhedra (grey)

to the pair of objects is put in the hysteresis array.

Before every invocation of the updating function of the algorithm, the bounding boxes for all objects must be set correctly. The algorithm updates all tiles according to the new bounding boxes. New collision pairs may be formed and others moved to the hysteresis array.

After every update, all pairs of object in the hysteresis array are checked. When a pair is in the array too long (longer than a user specified threshold), it is removed and the pair destroyed.

2.4 V-Clip

To compute the precise distance between two bodies in the simulator (or to determine whether they are penetrating or not), we use the V-Clip algorithm which is described in detail in [Mir97]; this section is only a short introduction of the algorithm. V-Clip (short for Voronoi Clip) is a very fast, accurate and robust algorithm which 'walks' over the features (vertices, edges, faces) of a pair of polyhedra in search for the pair of globally closest features. It is an enhancement over the Lin-Canny algorithm [Lin93], both in robustness and in ease of implementation. Both Lin-Canny and it's descendant V-Clip are among the fastest solutions for precise collision detection.

V-Clip operates on *convex polyhedra*. Polyhedra are descriptions of n-dimensional surfaces consisting of vertices, edges and faces. V-Clip does not work for non-convex polyhedra; the proof for theorem 2 will break. Non-convex polyhedra can be build from aggregations of convex polyhedra. But if a polyhedron is utterly non convex V-Clip is not the correct algorithm for the problem. The algorithm must know the neighborship relations between the features. Every edge has two neighboring vertices and two neighboring faces, every vertex has a number (> 2) of neighboring edges and every face has a number (> 2) of neighboring edges. The neighborship relations are symmetrical. Faces which share an edge can not be parallel; these faces should be merged into one face. Faces include their edges, and edges include their vertices: faces and edges are closed sets.

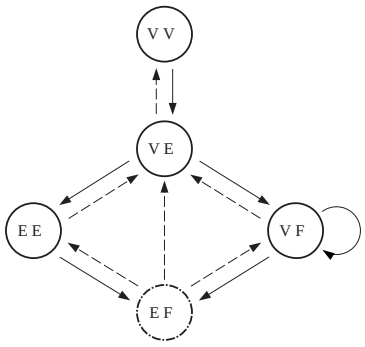

Figure 2.5: State transitions of the V-Clip algorithm. Copied from [Mir97].

Every feature F on a convex polyhedron has an associated Voronoi region $\mathcal{VR}(F)$. This is the set of points outside the polyhedron that are as close to F as any other feature on the polyhedron. Figure 2.4 shows the Voronoi regions for two 2D polyhedra. Voronoi regions are bounded by the features they belong to and by Voronoi planes (actually Voronoi *lines* in 2D). Points *on* Voronoi planes $\mathcal{VP}(F_a, F_b)$ are equally close to two neighboring features F_a and F_b, and thus belong to both Voronoi regions $\mathcal{VR}(F_a)$ and $\mathcal{VR}(F_b)$.

Both Lin-Canny and V-Clip are based on the following theorem:

Theorem 2. *Let X and Y be a pair of features from disjoint convex polyhedra, and let $\mathbf{x} \in X$ and $\mathbf{y} \in Y$ be the closest points between X and Y. If $\mathbf{x} \in \mathcal{VR}(Y)$ and $\mathbf{y} \in \mathcal{VR}(X)$, then $\mathbf{x}$ and $\mathbf{y}$ are a globally closest pair of points between the polyhedra.*

The Proof of theorem 2 is given in [Lin93]. The bold vertex and edge in figure 2.4 are the globally closest features on the pair of polyhedra; since it is not a degenerate situation, the vertex and exactly one point on the edge meet the conditions of theorem 2, but this does not need to be true; multiple pairs of points can satisfy 2 in degenerate situations.

The V-Clip algorithm searches for a globally closest pair of features. It does this by starting with a pair of features (usually from a previous invocation of the algorithm) and checking whether the conditions of theorem 2 are met. If they are, the algorithm reports the pair of features as globally closest. Otherwise it updates one of the features to a (neighboring) feature such that the inter-feature distance decreases *or* the dimensionality of one of the features decreases (a vertex has a lower dimensionality than an edge, an edge has a lower dimensionality than an face).

The transitions V-Clip can make are shown in figure 2.5. The states are identified by their the features the search algorithm is considering (e.g. V E for a vertex on one polyhedron and an edge on the other polyhedron). Solid lines represent transitions which *must* decreases inter-feature distance. Dashed lines decrease the dimensionality of one of the features, but may *not* decreases inter-feature distance; this implies that the inter-feature distance is kept the same because features are closed sets. Since there are no cycles in the graph

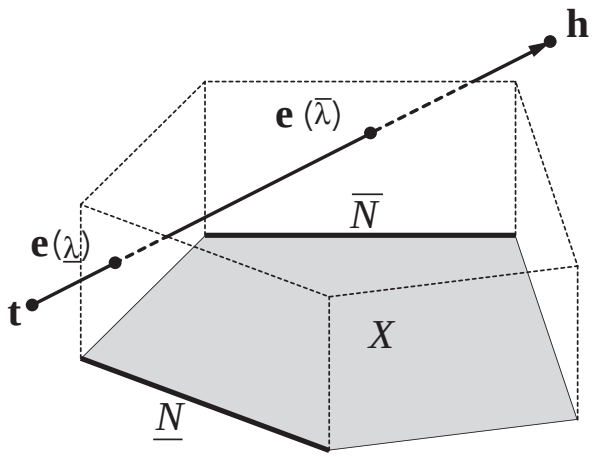

Figure 2.6: Clipping an edge against the planes bounding the Voronoi region of a pentagonal face X. The edge enters the regions as it crosses $\mathcal{VP}(X, \underline{N})$ at the parameterized point $e(\underline{\lambda})$; it exits the region as it crosses $\mathcal{VP}(X, \overline{N})$ at the point $e(\overline{\lambda})$. (Copied from [Mir97]).

consisting of only dashed lines, the algorithm must terminate; an infinite path through the graph implies an infinite number of inter-feature distance reductions, which is not possible if only a finite number of features are present in the polyhedra.

V-Clip basically uses three types of tests to determine which transitions to make: (simple) exclusion, edge clipping followed by derivatives checks, and escaping local minima. Tests are done and transitions are made until two features satisfying theorem 2 have been found.

A feature a is *excluded* from the Voronoi region $\mathcal{VR}(b)$ of another feature b if it lies entirely outside it; this implies that the feature a lies on the wrong side of one or more Voronoi planes bounding $\mathcal{VR}(b)$. If a feature lies on the wrong side of a Voronoi planes, it is said that this plane is *violated* by the feature. The greater the distance of the feature to the plane, the more it violates it. If feature a is excluded from $\mathcal{VR}(b)$ V-Clip will update feature b to the neighboring feature on the other side of the Voronoi plane which a violates maximally.

V-Clip manipulates edges as vectors from a tail point $\mathbf{t}$ to a head point $\mathbf{h}$. Points $\mathbf{e}$ along an edge are parameterized as

$$\mathbf{e}(\lambda) = (1 - \lambda)\mathbf{t} + \lambda\mathbf{h}, 0 \leq \lambda \leq 1. \tag{2.4}$$

It can happen that an edge e is *partially excluded* from the Voronoi region $\mathcal{VR}(f)$ of another feature f. The edge is then *clipped* against $\mathcal{VR}(f)$, as shown in figure 2.6, resulting in two points $\mathbf{e}(\underline{\lambda})$ and $\mathbf{e}(\overline{\lambda})$. These are the points where the edge enters and leaves $\mathcal{VR}(f)$. V-Clip then computes the derivatives of the distance of the e to f at these two points as a function of λ. Based on the signs

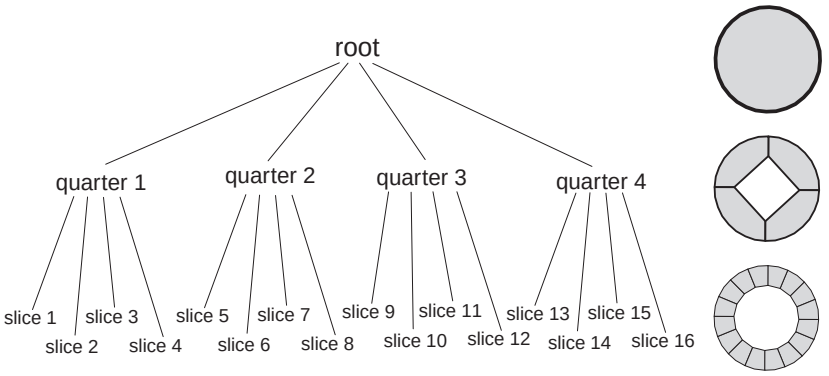

Figure 2.7: Representing a concave polyhedron with a hierarchy of convex polyhedra.

of derivatives it is decided if should be updated, and if so, to which feature f should be updated.

One more test is used in the algorithm: checking for a local minimum. When a vertex v lies *below* a face f, but does not violate any of the Voronoi planes which bound $\mathcal{VR}(f)$, the algorithm can not compute from *local* information whether v is penetrating the polyhedron to which face f belongs, or that is trapped in a local minimum. It must check the vertex v against all faces of the polyhedron until it finds a face which v lies *above*. This is an $O(n)$ search.

The number of features visited by the algorithm during the search (assuming it does not get trapped in a local minimum, which causes an $O(n)$ search) is $(O(\sqrt{(}n))$ at worst assuming the features are reasonably equally distributed across polyhedra. At most states the algorithm will pick the feature which decreases the distance the most (if possible). This causes the algorithm to search along a 'short' (but not necessarily the shortest) path to the pair of closest features. A 1 dimensional path along a 2 dimensional surface visits $(O(\sqrt{(}nW))$ features required to describe the 2D surface.

Trees of convex polyhedra can be used to represent larger (concave) polyhedra, as shown for a 2D torus in figure 2.7. A convex hull is computed (using the Quickhull algorithm, [BDH96]) for each node in the tree which is not a leaf. The algorithm first checks for penetration of the outer hull of the polyhedra. If this is true, the hull is 'opened' and the polyhedra inside it are check for penetration. This process continues until non-penetration has been determined or the leaves of the tree have been reached.

2.4.1 Exploiting Coherence in the V-Clip algorithm

The 'trick' which allows algorithms like V-Clip and Lin-Canny to achieve *near* O(1) performance is *caching* results from previous invocations. In my implementation they are stored in a hash table. Each time the algorithm is called to determine the distance between two polyhedra, it retrieves the stored results

from cache (if they are present). It then uses these (previously closest) features as a starting point for the search. Often just one or two steps have to be made because the relative positions and orientations of the polyhedra have changed little. This makes the performance almost independent from the number of features.

2.4.2 Extensions

The first and most simple extension to the basic V-Clip algorithm as described in [Mir97] is to *also* be able to compute axis aligned bounding boxes for trees of polyhedra, instead of relying on the Quickhull algorithm. This way the algorithm can be used with or without the Quickhull algorithm.

The second extensions is that it can use a threshold on the distance between two non-convex polyhedra before the convex hulls of the polyhedra are opened. In rigid body simulation often a small 'collision epsilon' ϵ is used. When the distance between two polyhedra is smaller than ϵ, it is said that they are colliding or in contact. But the V-Clip algorithm will not always report the true distance between two non-convex polyhedra made from trees of convex polyhedra; only a lower bound on the distance between them is reported. To ensure the true distance is reported below a certain threshold ϵ the *open hull threshold* is provided.

The third extension is it's ability to track contact points. The bare V-Clip algorithm can only report the distance between the *closest* points on two polyhedra. To apply contact forces or micro impulses, it would be nice to be able to *track* points on the polyhedra which were previously closest points. Our implementation does this by returning references to the closest *features* on every invocation. These references can be handed over to an update routine which determines the closest points on the features and checks whether these points still lie inside the Voronoi regions of the features.

2.5 Implementation

Both algorithms (hierarchical hashing and V-Clip) were implemented as C libraries. The libraries do not depend on each other so they can be used either in isolation or together. They do have similar function syntax and structure which makes learning their APIs easier. The bounce program was written in C++.

Both libraries have (optionally compiled) functions to render their 'state' graphically using OpenGL. Almost all rendering in the bounce program is done using these functions. These function can be very helpful while debugging.

The hierarchical hashing algorithm in not dependant on any other library (except OpenGL for rendering, but this is optional). The V-Clip library uses the Quickhull algorithm [BDH96] to compute hulls of compound objects or to compute polyhedral objects if the user only specifies vertices (and no faces). If the Quickhull library is not available, V-Clip will compute axis oriented bounding

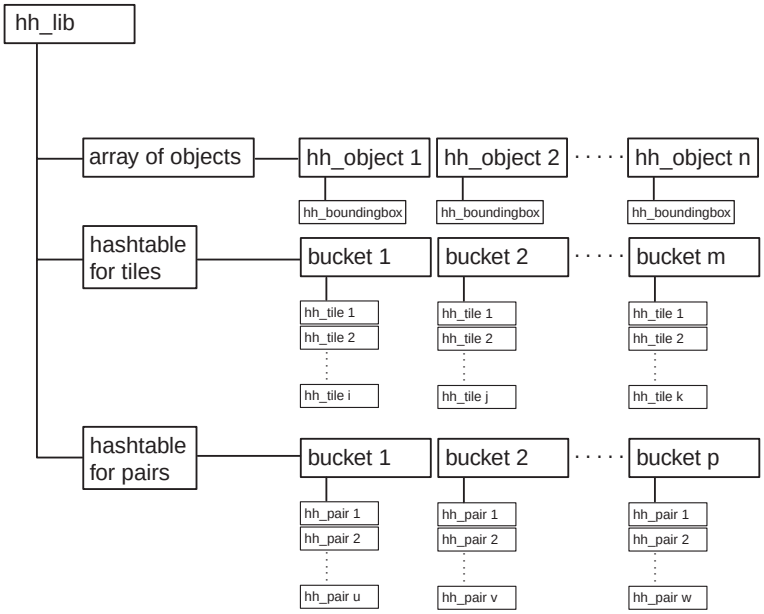

Figure 2.8: Hierarchical hashing structure

boxes instead of real convex hulls for compound objects; users must always fully specify both vertices and faces for polyhedral objects if the Quickhull library is not linked.

The structure of the hierarchical hashing implementation is shown in figure 2.8. The user creates a hh_lib, which contains an array of hh_objects (the objects you want to perform broad phase collision detection on), a hash table which will hold the tiles intersected by hh_objects, and a hash table which contains active collision pairs. After the hh_lib structure has been created, the user can add and remove objects to and from the library and set the bounding boxes (hh_boundingbox) for each object. Objects can be added and removed dynamically, not only when the library is being initialized. The user calls a function to update the hierarchical hash table, which cause tiles to be stored or removed, and new collision pairs to be created or destroyed. The hash tables are automatically rehashed if they get too full. The hysteresis array, not shown in figure 2.8, holds pointer to collision pairs do not collide any more but are kept active for a short interval to prevent rapid creation and deletion of collision pairs.

Figure 2.9 depicts the structure of the V-Clip implementation. The user creates a library which can contain geometric primitives (polyhedra), objects and a hash table to cache results from previous V-Clip computations. Then a number of convex geometric primitives are added to the library. Primitives contain vertices, edges and faces, and the Voronoi planes computed from these features. From primitives (non-convex hierarchies of) objects can be created. Every object has a orientation and position, as specified by the user. Every time

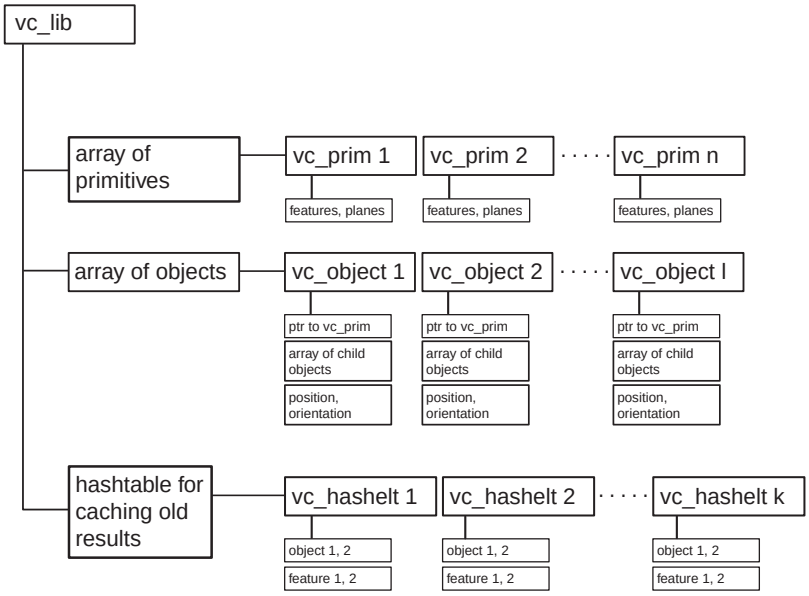

Figure 2.9: V-Clip structure

V-Clip is invoked on a pair of objects, the results are cached in the hash table so they can be used as a starting point for the next call. Every hash element (vc_hashelt) contains pointers to the two object involved, and the indices of the closest features on both objects.

V-Clip can use either rotation matrices or quaternions to represent orientations. Quaternions require on 4 floats storage, rotation matrices 9; vectors can be transformed with fewer floating point operations using rotation matrices however.

Our implementation of V-Clip can automatically return the actual coordinates of the closest points between the two polyhedra (instead of just references to the closest *features*). References to features can be returned as well; they can be used to *track* contact points over a prolonged period of time.

2.6 Bounce Demonstration and Benchmark Program

To test, demonstrate and benchmark the implementations of V-Clip and hierarchical hashing, we wrote the 'bounce' program. Bounce creates a three dimensional 'world' containing a number of objects. The size (width, height, depth) of the world and the number of objects in the world can be specified by the user. The objects are simple geometric primitives (cube, sphere, cone, tube, pyramid and torus) of three different sizes. The bounce program then makes the objects move, rotate and bounce into each other and the boundaries of the world. This process can be viewed in real time by the user, or used without

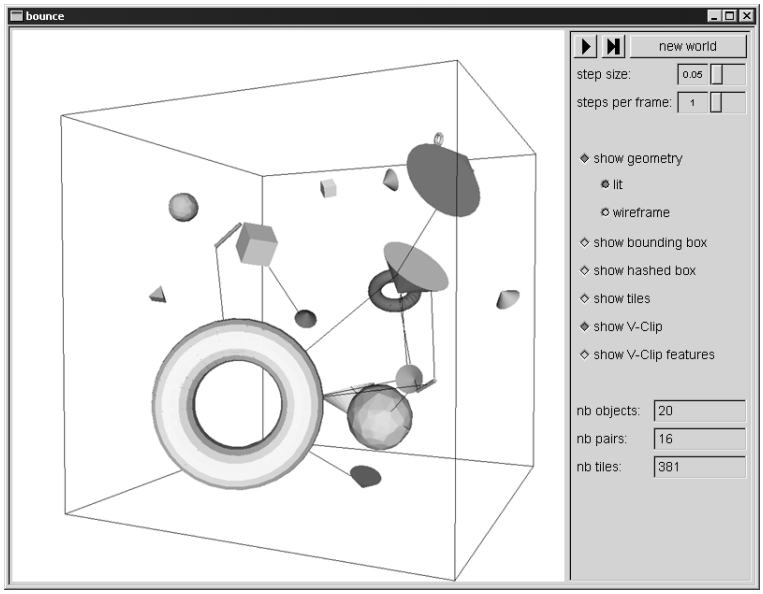

Figure 2.10: Screenshot of the bounce program. The black lines between objects connect the closest points between pairs of objects.

graphical display for benchmarking or testing. The hierarchical hashing algorithm can be enabled or disabled to see or measure the effect of the algorithm on the efficiency of the total collision detection solution used by bounce and our rigid body simulator.

When displayed graphically the user can witness the algorithms 'in action' which can make them easier and more intuitive to understand. The properties of the algorithms which can be displayed by the bounce program are:

- object geometry (V-Clip)
- a line between the closest points of two objects (V-Clip)
- the closest features of two objects (V-Clip)
- bounding box of each object (hierarchical hashing)
- hashed box of each object (hierarchical hashing)
- the tiles hashed for each object (hierarchical hashing)
- number of objects
- number of active collision pairs
- number of tiles currently hashed

Bounce makes the objects move according to a few simple rules. These rules have been kept as simple as possible to keep the source code short and easy to understand. The rules are:

- when an object is initialized it receives a random position, orientation, linear velocity and angular velocity and a random 'gravity factor' (explained below).
- On each step, the state of each object is updated with a step size specified by the user. On each step the following happens:
  1. the bounding boxes for the step are computed.
  2. the hierarchical hash table is updated. Collision pairs are created and removed accordingly.
  3. all objects move according to their velocity and the step size. Their linear velocity is updated according to their gravity factor.
  4. the distances between all objects currently paired are computed using V-Clip
  5. all objects who's center is outside the boundaries of the world receive a new random linear velocity which will direct it back inside the boundaries of the world.
  6. all objects which penetrate receive new random linear velocities such that their centers will move away from each other.

Every object in bounce has a small random 'gravity factor'. On each step, this gravity factor is added to the y-component of the linear velocity. The gravity factor can be positive or negative, so objects 'fall' in two directions. This was done to more closely simulate the dynamic behavior of a real rigid body simulator, since that is what the V-Clip and hierarchical hashing algorithms are used for in our case. Objects fall in two directions to make sure they do not stick to the 'floor' but are sure to visit all parts of the world.

Note that bounce does not enforce non-penetration of objects or non-violation of the world-boundaries. It simply alters velocities of objects which penetrate or are outside the world boundaries such that the penetrations or violations will cease to exist. When multiple objects collide which each other it is possible that some objects will penetrate each other for several steps. bounce was implemented this way for two reasons. First all to make sure the code in V-Clip which handles penetration is also thoroughly tested. If objects would never penetrate, these parts of our V-Clip implementation would never be used by bounce. Secondly to keep it simple. Bounce is also intended to demonstrate how to use our V-Clip and hierarchical hashing implementations. People who read the Bounce source code do not benefit from complicated algorithms which have nothing to do with V-Clip or hierarchical hashing and are only there to enforce non-penetration of objects or non-violation of the world-boundaries.

Bounce has been used to extensively test the V-Clip and hierarchical hashing algorithms. Runs up to 100 millions steps have been made (taking up to 10 hours) to ensure stability of both algorithms. Also, all benchmarks in the collision detection results section were made using bounce.

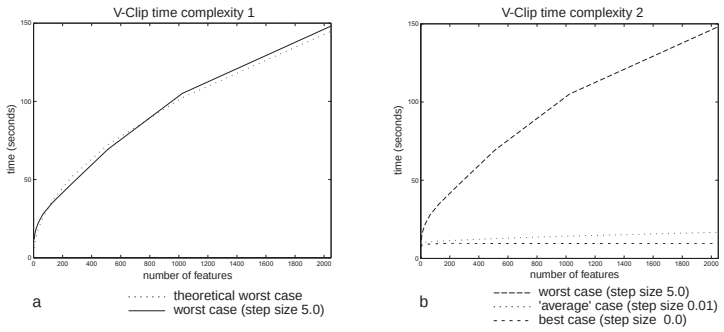

Figure 2.11: Influence of increasing geometric complexity on the V-Clip algorithm

2.7 Collision Detection Results

This section present empirical data, to support the time complexity claims made in the theoretical section. All measurements were made with the 'bounce' program with graphical display disabled. The machine used to make the measurements was based on an Intel Celeron 450 with 100 mhz memory bus. The program and libraries were compiled with Microsoft Visual C++ 6.0, with optimizations turned on and run on Microsoft Windows NT 4.0. Every benchmark value presented is an average of ten runs.

Above it is claimed that V-Clip has a time complexity between $O(1)$ and $O(\sqrt{n})$, where n is the number of geometric features in the objects involved. Experiments were done to verify this claim. The bounce program was used to perform a number of benchmarks in which two objects of increasing geometric complexity are moving around in the world. Hashing was turned off, to isolate the V-Clip algorithm as much as possible. The performance of V-Clip is dependant on the application in a way that the more the objects move relative to each other, the more 'expensive' the V-Clip collision checks get; the algorithm has to visit more features to find the new pair of closest features. Five series of ten benchmarks were done, three of which are shown in figure 2.11a and 2.11b. The difference between the series is the step size made by the bounce program between frames. The larger the step size, the more objects move relative to each other between collision checks, the worse the performance of V-Clip is. The geometry of the objects used in the benchmarks was created by randomly generating n unit vectors (the vertices of the geometry), and computing a convex hull around them using the Quickhull algorithm [BDH96].

The best performance is achieved when the objects do not move at all (step size 0.0). The V-Clip algorithm does nothing but verify that the previously closest features are still valid. It is interesting to see that the performance of the algorithm decreases slightly as the objects get more complex. This is probably related some kind of memory caching by the CPU. The 'average' case, step size 0.01, is representative for what would happen in the typical rigid body simulator. Between frames, objects move little relative to each other. V-Clip

Table 2.1: World sizes used for benchmarking hierarchical hashing

number of objects	world size	number of pairs created
16	5	219
32	21	867
64	40	1807
128	88	2995
256	120	5922
512	180	10300
1024	250	18310
2048	320	36886
4096	430	66839

achieves near $O(1)$ performance, but a small $O(\sqrt{n})$ term is already involved. The worst case (step size 5.0) exhibits $O(\sqrt{n})$ performance, as is clear from figure 2.11a, where a $\sqrt{n}$ function was fitted to the graph.

To support the claim that the combination of hierarchical hashing and V-Clip results in a linear time complexity ($O(n+c)$) collision detection algorithm, measurements had to be made with some care. The idea is to measure the time it takes to do 10000 updates with step size 0.01. The benchmark is run several times for different numbers of objects (16, 32, ..., 4096). If we plot the measurements we would expect to see a linear graph, as predicted by the theoretical linear time complexity. However, if the size of the world is kept constant, the world gets more and more crowded as the number of objects increases. This automatically results in the creation of $O(n^2)$ pairs of objects which could collide, which all have to be detected and reported (the c term in the $O(n + c)$). Thus we would measure a quadratic time complexity, caused by a quadratic c term. To solve this problem, we first wrote an optimization program which computes the world sizes for which the average number of pairs created ($O(c)$) during benchmark is approximately proportional to the number of objects in the world ($O(n)$). The world sizes and the average number of pairs created are listed in table 2.1.

As can be seen in figure 2.12a, the linear time complexity claim holds if measurements are made this way.

To measure the benefits of exploiting coherence while hashing ten runs of 100000 steps (step size 0.01, 25 objects inside world) were made. The hashing algorithm performed 3.6 times better when coherence was taken into account (79.2 seconds vs. 286 seconds per run).

Finally, to demonstrate the huge performance gain which can achieved by using hierarchical hashing, a number of benchmarks were made with hashing either turned off or on, which are plotted in figure 2.12b (note the log scale of the y-axis). For an increasing number of objects (2, 4, ... 32) benchmark runs were done consisting of 10000 steps (step size 0.01). The values for 64 and 128 objects were extrapolated from runs consisting of 1000 steps (step size 0.01).

For very low object counts the performance is better with hashing turned

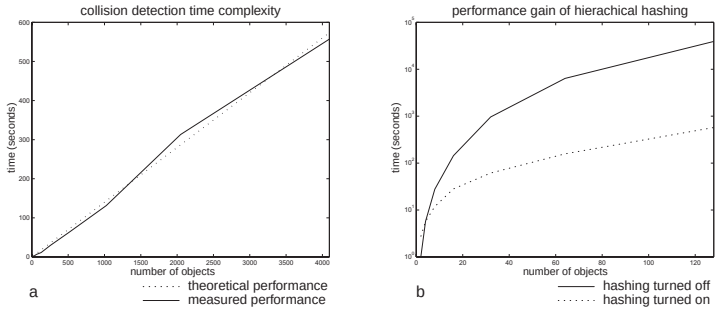

Figure 2.12: a) Increasing the number of objects in the scene. b) Influence of hierarchical hashing on collision detection (note the log scale on the y-axis).

off, but for the higher counts the benefit of using the hierarchical hashing algorithm becomes very clear.

2.8 Discussion and Conclusion

We have implemented the hierarchical hashing algorithm by [Mir96] which allows for $O(n)$ broad phase collision detection. Our implementation allows dynamic removal and addition of objects and dynamic control over the resolution at which at objects should be hashed.

Our example program 'bounce' invoked the hashing algorithm on every frame. In a real application one would not invoke the hashing algorithm on every frame but much less often (depending on the application). The application would determine a suitable time period and predict the swept volumes for objects during that period. It would then invoke the hashing algorithm at the beginning of the period and check whether the objects remained within their predicted bounding boxes. This would probably result in the hashing algorithm speeding up the collision detection, even for low object counts which were actually slowed down by the hashing algorithm as shown in figure 2.12b.

The hierarchical hashing algorithm was implemented such that it exploits coherence. When coherence is exploited the algorithm is approximately 3.6 times faster. Mirtich [Mir96] reports that his implementation becomes 6 times faster when coherence is exploited. This could be because his non-coherence-exploiting implementation is less efficient than mine, or his coherence-exploiting implementation is more efficient than mine. Possibly Mirtich's non-coherence-exploiting implementation reallocates memory resources on every update, which could well double the processing time.

We use a hash table to store the close counters of collision pairs. [Mir96] used a two dimensional array, which results in $O(n^2)$ storage requirements. A hash table requires only $O(n)$ storage and still has $O(1)$ access time (assuming perfect hashing).

Initially we implemented the Lin-Canny algorithm [Lin93] to compute the

exact distance between bodies, but it has the major drawback that it can not properly detect penetrating bodies; either one has to prevent that objects will ever penetrate (by taking very conservative integration steps), or it will get caught in an infinite loop which has to be detected and broken somehow. It can also loop if degeneracies (e.g. parallel edges or faces) are not treated properly. That is why we decided to use the more robust V-Clip [Mir97] algorithm.

One interesting problem was encountered while testing the V-Clip library compiled with the Microsoft Visual C++ compiler. Sometimes the algorithm got stuck in an infinite loop for no apparent reason. When V-Clip gets stuck in a loop this might be an indication that the derivative signs are not being computed correctly. The problem would only occur when the library was compiled with optimizations on.

After some study we found out that the Microsoft Visual C++ compiler sometimes changes the order in which floating point operations are performed. This is done to increase run-time performance. However, the same derivative computation was being done in several places (using an inlined function). The compiler changed the order in which floating point computations are done which resulted in different floating point round off errors (and thus different derivative signs). The result was that the V-Clip algorithm would get stuck in a cycle because at feature a the derivative sign told it to go to feature b, and at feature b the derivative sign told it to go back to feature a. The problem was solved by not inlining the derivative computation function (at the cost of slightly lower performance). Thus we have found that V-Clip is not robust and can also get caught in infinite loops when compiler-optimizations cause different floating point round off errors when the same computation is done in different locations in the code.

Another problem that we overlooked during the implementation of V-Clip was that faces that share an edge must not be parallel. Initially our implementation assumed that all faces are triangles; e.g. the square face of a cube must be modeled with two parallel triangles. This also caused infinite loops. The current implementation allows for faces with an arbitrary number of edges.

We have found that quaternions may be better suited than rotation matrices for use with the V-Clip algorithm because V-Clip is sensitive to non-orthonomal frames. They can cause infinite loops due to incorrect derivative computations. Quaternions and rotations matrices can drift easily if they are updated incrementally due to floating point round off errors, but quaternions are normalized ('orthonomalized') much easier than rotation matrices because the four quaternions elements do not depends on each other as much as the nine rotation matrix elements.

We have shown through experiments that the combination of the hierarchical hashing algorithm and V-Clip truly results in $O(n + c)$ time complexity. These results were not present in literature (as far as we know).

Also we have made measurements on the performance of V-Clip for a wide range of object geometry complexities. We have shown through experiments that the *near* $O(1)$ time complexity claim holds in practice. Also we have shown an upper bound on the time complexity ($O(\sqrt{n})$, where n is the number of features) exists by measuring the worst case performance.

Chapter 3

Rigid Body Simulation

3.1 Introduction

This chapter discusses the internals of the rigid body simulator. The problem of rigid body simulation is, given the initial state of a system of rigid bodies, predict the state of the system at some time t in the future.

The equations describing how to predict that state are too complicated to solve in closed form (for a *general* enough system), thus we must use explicit simulation. Although the laws governing the behavior of rigid bodies are well known (and may be regarded as high school physics), it's no simple task to write a simulator which handles every physical system or situation correctly. If bodies are connected by (rotational or translational) joints the equations derived from the above mentioned laws become very non trivial.

The simulator consists of many parts, of which the collision detection system discussed in the previous chapter is only one (albeit a large one). All these parts have to work together seamlessly.

The rest of this chapter is divided as follows. First, an introduction to the mathematics used in this chapter is given. This is followed by description of the physical laws that govern the simulation and the assumptions made to make the simulation viable in real time. The computation of mass properties of bodies (such as the mass, the center of mass and the inertia tensor) is explained in the next section. The topic of computing a pair of equal but opposite impulses is dealt with in section 3.4. A compact but rather lengthy summary of the Featherstone algorithm for articulated bodies [Fea87] and the extensions my simulator uses is given in section 3.6, which is followed by the section describing how all these parts work together in the simulator.

Table 3.1: Symbols used for common rigid body dynamics quantities

quantity	symbol used
position	$\mathbf{x}$
linear velocity	$\mathbf{v}$, $\mathbf{u}$, $\dot{\mathbf{x}}$
linear acceleration	$\mathbf{a}$, $\ddot{\mathbf{x}}$
orientation	$\mathbf{q}$
angular velocity	$\boldsymbol{\omega}$, $\dot{\mathbf{q}}$
angular acceleration	$\boldsymbol{\alpha}, \dot{\boldsymbol{\omega}}, \ddot{\boldsymbol{q}}$
force	$\mathbf{f}$
torque	$\boldsymbol{\tau}$
(linear) impulse	$\mathbf{p}$
angular impulse	$\boldsymbol{\phi}$
mass	m
matricized mass	$\mathbf{M}$
inertia tensor	$\mathbf{I}$
center of mass	$\mathbf{r}$
collision matrix	$\mathbf{K}$
friction coefficient	μ
restitution coefficient	e
collision epsilon	ϵ
gravity	$\mathbf{g}$

3.2 Mathematical notation and preliminaries

3.2.1 Common rigid body dynamics quantities

The names of common quantities used in rigid body dynamics simulation are listed in table 3.1. Sometimes different symbols are used for the same quantity, depending on the context.

3.2.2 Matrices, vectors

All matrices and vectors are denoted with bold characters, such as $\mathbf{M}$ and $\mathbf{v}$. The $n \times n$ identity matrix will be denoted with $\mathbf{1}$, where n depends on the context. The $n \times m$ null matrix or vector will be denoted with $\mathbf{0}$; again n and m depend on the context.

3.2.3 Frames, points and vectors

Any point or vector (direction) in 3D space can be represented using a weighted sum of three independent basis vectors. If we take for instance $\mathbf{i} = \begin{bmatrix}1 & 0 & 0\end{bmatrix}$, $\mathbf{j} = \begin{bmatrix}0 & 1 & 0\end{bmatrix}$, $\mathbf{k} = \begin{bmatrix}0 & 0 & 1\end{bmatrix}$, then the point $\mathbf{p} = \begin{bmatrix}4 & -5 & 6\end{bmatrix}$ can be repre-

sented as $4 * \mathbf{i} - 5 * \mathbf{j} + 6 * \mathbf{k}$. In matrix notation this yields:

$$\mathbf{p} = \mathbf{I}\mathbf{p} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p} \tag{3.1}$$

The matrix $\mathbf{I}$, which represents a *frame*, is called the identity matrix because it leaves $\mathbf{p}$ unchanged.

In rigid body dynamics it is (very) convenient to represent the position and orientation of a body by a vector specifying the offset $\mathbf{o}$ of the body's center of mass relative to the global origin of the world and a frame (either a rotation matrix $\mathbf{M}$ or a rotation quaternion $\mathbf{q}$) specifying the rotation of the body relative to the identity frame. To problem of relating a local point $\mathbf{p}_l$ connected to the body to it's position $\mathbf{p}_g$ in global coordinates then arises. To transform $\mathbf{p}_l$ to $\mathbf{p}_g$ we have to compensate for the offset and rotation:

$$\mathbf{p}_g = \mathbf{M}\mathbf{p}_l + \mathbf{o} \tag{3.2}$$

The other way around (from global to local coordinates) is performed by the following equation:

$$\mathbf{p}_l = \mathbf{M}^{-1}\left(\mathbf{p}_g - \mathbf{o}\right) \tag{3.3}$$

Rotation matrices have the nice property that their inverse is equal to their transpose:

$$\mathbf{M}^{-1} = \mathbf{M}^T \tag{3.4}$$

so that the above is efficient to compute.

3.2.4 Transpose, dot product

The transpose of a vector $\mathbf{v}$ will be denoted with $\mathbf{v}^T$. So, suppose $\mathbf{v}$ is a vector of length 3, then

$$\mathbf{v}^T = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}^T = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix}. \tag{3.5}$$

The dot (also known as *inner* or *scalar*) product of two vectors $\mathbf{v}$ and $\mathbf{w}$ can then easily be written as:

$$\mathbf{v} \cdot \mathbf{w} = \mathbf{v}^T\mathbf{w} = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix} \begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix} = v_x w_x + v_y w_y + v_z w_z. \tag{3.6}$$

The magnitude of the dot product of two vectors $\mathbf{v}$ and $\mathbf{w}$ is $\mid \mathbf{v} \mid\mid \mathbf{w} \mid \cos\theta$, where θ is the angle between the vectors.

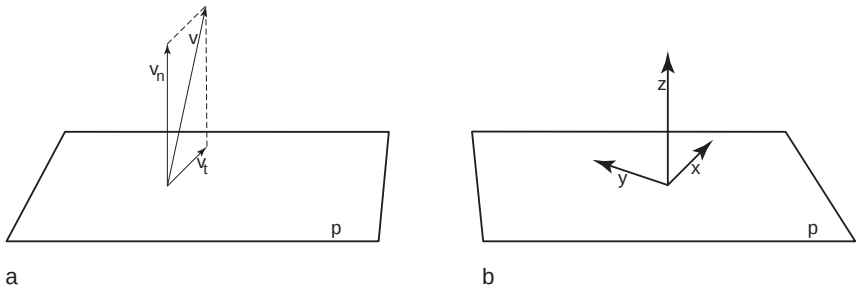

Figure 3.1: a) normal component $\mathbf{v}_n$ and tangential component $\mathbf{v}_t$ of a vector $\mathbf{v}$ relative to a plane p. b) constructing a frame with it's z-axis parallel to the normal parallel to a plane p.

3.2.5 Cross product

The cross product $\mathbf{v} \times \mathbf{w}$ of two vectors is a vector of magnitude $\mid \mathbf{v} \mid\mid \mathbf{w} \mid \sin\theta$, normal to the plane containing both $\mathbf{v}$ and $\mathbf{w}$, such that $\mathbf{v}$, $\mathbf{w}$ and $\mathbf{v} \times \mathbf{w}$ form a right-handed set. The cross product of is defined as

$$\mathbf{v} \times \mathbf{w} = \begin{bmatrix} v_y w_z - v_z w_y \\ v_z w_x - v_x w_z \\ v_x w_y - v_y w_x \end{bmatrix}. \tag{3.7}$$

This can be abbreviated to

$$\mathbf{v} \times \mathbf{w} = \tilde{\mathbf{v}}\mathbf{w} \tag{3.8}$$

by defining $\tilde{\mathbf{v}}$ as

$$\tilde{\mathbf{v}} = \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix}. \tag{3.9}$$

3.2.6 Tangential and normal components of a vector

During this discussion the terms *normal* component and *tangential* component of a vector will be used. These terms are usually used in the context of a plane, since that plane defines the normal. The normal of a plane a line perpendicular to that plane with length 1. If the equation of a plane p is

$$p : Ax + By + Cz + D = 0 \tag{3.10}$$

then the normal of that plane is

$$\mathbf{n} = \| \, [ABC]^T \, \| \, . \tag{3.11}$$

A vector $\mathbf{v}$ may then be factored into a normal component $\mathbf{v_n}$ and tangential ('not normal') component $\mathbf{v_t}$ as follows:

$$\mathbf{v}_n = (\mathbf{v}^T \mathbf{n})\mathbf{n} \tag{3.12}$$

$$\mathbf{v}_t = \mathbf{v} - \mathbf{v_n}. \tag{3.13}$$

$\mathbf{v}_n$ has no component in the plane p, while $\mathbf{v}_t$ lies in it entirely . The normal and tangential components of a vector $\mathbf{v}$ relative to a plane p are shown in figure 3.1a.

Sometimes we will construct a frame with it's z-axis parallel to the normal of a plane to simplify certain computations. This is shown in 3.1b. The directions of the x- and y-axis are usually not important, only that they form an righthanded frame with the z-axis.

3.2.7 Using Quaternions for Representing Orientation

For use in dynamics, the quaternion rotation operator (see for instance [Kui99] for a comprehensive presentation of rotation matrices and quaternions, or [GLD93] for an introduction to *geometric algebra*, which makes quaternions much easier to understand intuitively) is often much more suited to represent orientation than a the matrix rotation operator.

The quaternion representation or rotations requires less values (four as opposed to nine for the rotation matrix) and is easily normalized. When updating a vector (such as position or orientation) incrementally (such as when integrating numerically), the values of the vector drift due to integration round off errors. But it is important to keep the axes of a rotation matrix orthonormal, which is a rather complicated operation. Quaternions also suffer from drift but can be normalized by simply dividing it by it's Euclidean length. The quaternion rotation operator requires slightly more floating point operations to rotate a vector than a the rotation matrix operator, but multiplying two quaternions requires *less* floating point operations.

It is simple to construct a quaternion $\mathbf{q}$ to represent a rotation about a certain axis $\mathbf{a}$ with an angle ϕ:

$$\mathbf{q} = \begin{bmatrix} \cos(0.5\phi) \\ \sin(0.5\phi)a_x \\ \sin(0.5\phi)a_y \\ \sin(0.5\phi)a_z \end{bmatrix} = \begin{bmatrix} q_s \\ q_x \\ q_y \\ q_z \end{bmatrix} \tag{3.14}$$

Equation 3.14 shows the naming convention for the individual quaternion components (q_s, q_x, q_y and q_z). It is also easy to deduce from equation 3.14 that the inverse $\mathbf{q}^{-1}$ of a quaternion $\mathbf{q}$ is

$$\mathbf{q}^{-1} = \begin{bmatrix} q_s \\ -q_x \\ -q_y \\ -q_z \end{bmatrix} \tag{3.15}$$

Two rotation quaternions $\mathbf{p}$ and $\mathbf{q}$ can be 'summed together' (their effects accumulated) by multiplying them as follows:

$$\mathbf{pq} = \begin{bmatrix} q_s & q_x & q_y & q_z \\ q_x & q_s & q_z & q_y \\ q_y & q_z & q_s & q_x \\ q_z & q_y & q_x & q_s \end{bmatrix} \mathbf{p} \tag{3.16}$$

To rotate a vector $\mathbf{v}$ we use

$$\mathbf{v}_{rot} = \mathbf{q} \begin{bmatrix} 0 \\ v_x \\ v_y \\ v_z \end{bmatrix} \mathbf{q}^{-1} \tag{3.17}$$

The derivative of a quaternion representing a rotation about an axis $\| \boldsymbol{\omega} \|$ at $|\boldsymbol{\omega}|$ rads/sec with respect to time is

$$\dot{\mathbf{q}} = \begin{bmatrix} \dot{q_s} \\ \dot{q_x} \\ \dot{q_y} \\ \dot{q_z} \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -q_x & -q_z & -q_z \\ q_s & -q_z & q_y \\ q_z & q_s & -q_x \\ -q_y & q_x & q_s \end{bmatrix} \boldsymbol{\omega} \tag{3.18}$$

3.2.8 Featherstone's spatial algebra

Spatial quantities

When describing the dynamics of three dimensional systems, spatial algebra, developed by Featherstone [Fea87], is a useful notation. A spatial vector is a six-dimensional vector which replaces two three-dimensional vectors. Spatial vectors usually consist of a part describing *linear* (translational) quantities, and a part describing *angular* (rotational) quantities. For instance, spatial velocity is the combination of linear and angular velocity; a spatial force describes both force and moment. Spatial quantities (1×6 vectors and 6×6 matrices) will be denoted with a caret, as follows: $\hat{\mathbf{v}}$, $\hat{\mathbf{M}}$. Spatial algebra will be used in the section describing the Featherstone algorithm.

The most common spatial quantities used in rigid body simulation are defined in table 3.2.

Spatial transformations

Just as with ordinary 3D vectors, we will need change-of-basis or transformation matrices as described above. The construction of these 6×6 *spatial transformation matrices* is most conveniently done in two steps, one for translation, and one for rotation.

Suppose we have two frames $\mathcal{F}$ and $\mathcal{G}$ which are translated but not rotated relative to each other. Suppose we want to express the spatial velocity $\hat{\mathbf{v}}_{\mathcal{F}}$ in frame $\mathcal{F}$ in frame $\mathcal{G}$, we will need to adjust the linear velocity for the translation $\mathbf{r}$ from $\mathcal{F}$ to $\mathcal{G}$ as follows:

Table 3.2: Common spatial quantities

spatial quantity	definition	ordinary quantities
spatial velocity	$\hat{\mathbf{v}} = \begin{bmatrix} \boldsymbol{\omega} \\ \mathbf{v} \end{bmatrix}$	angular velocity ($\boldsymbol{\omega}$), linear velocity ($\mathbf{v}$)
spatial acceleration	$\hat{\mathbf{a}} = \begin{bmatrix} \boldsymbol{\alpha} \\ \mathbf{a} \end{bmatrix}$	angular acceleration ($\boldsymbol{\alpha}$), linear acceleration ($\mathbf{a}$)
spatial impulse	$\hat{\mathbf{p}} = \begin{bmatrix} \mathbf{p} \\ \boldsymbol{\phi} \end{bmatrix}$	impulse ($\mathbf{p}$), angular impulse ($\boldsymbol{\phi}$)
spatial force	$\hat{\mathbf{f}} = \begin{bmatrix} \mathbf{f} \\ \boldsymbol{\tau} \end{bmatrix}$	force ($\mathbf{f}$), moment ($\boldsymbol{\tau}$)
spatial axis	$\hat{\mathbf{s}} = \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix}$	prismatic part ($\mathbf{u}$), revolute part ($\mathbf{v}$)

$$\hat{\mathbf{v}}_{\mathcal{G}} = \begin{bmatrix} \boldsymbol{\omega} \\ \mathbf{v}_{\mathcal{G}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\omega} \\ \mathbf{v}_{\mathcal{F}} + \boldsymbol{\omega} \times \mathbf{r} \end{bmatrix} = \begin{bmatrix} \mathbf{1} & \mathbf{0} \\ -\tilde{\mathbf{r}} & \mathbf{1} \end{bmatrix} \begin{bmatrix} \boldsymbol{\omega} \\ \mathbf{v}_{\mathcal{F}} \end{bmatrix} = \hat{\mathbf{X}}_{\mathcal{FG}} \begin{bmatrix} \boldsymbol{\omega} \\ \mathbf{v}_{\mathcal{F}} \end{bmatrix}.$$

$\hat{\mathbf{X}}_{\mathcal{FG}}$ is a spatial *translation* matrix translating spatial vectors from frame $\mathcal{F}$ to frame $\mathcal{G}$. If $\mathcal{F}$ is rotated but not translated with respect to $\mathcal{G}$, another spatial transformation matrix is required:

$$\hat{\mathbf{v}}_{\mathcal{G}} = \begin{bmatrix} \boldsymbol{\omega}_{\mathcal{G}} \\ \mathbf{v}_{\mathcal{G}} \end{bmatrix} = \begin{bmatrix} \mathbf{R}\boldsymbol{\omega}_{\mathcal{F}} \\ \mathbf{R}\mathbf{v}_{\mathcal{F}} \end{bmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{0} \\ \mathbf{0} & \mathbf{R} \end{bmatrix} \begin{bmatrix} \boldsymbol{\omega}_{\mathcal{F}} \\ \mathbf{v}_{\mathcal{F}} \end{bmatrix} = \hat{\mathbf{X}}_{\mathcal{FG}} \begin{bmatrix} \boldsymbol{\omega}_{\mathcal{F}} \\ \mathbf{v}_{\mathcal{F}} \end{bmatrix}$$

where $\mathbf{R}$ is a 3×3 rotation matrix transforming vectors from $\mathcal{F}$ to $\mathcal{G}$. Here, $\hat{\mathbf{X}}_{\mathcal{FG}}$ is a spatial *rotation* matrix rotating spatial vectors from frame $\mathcal{F}$ to frame $\mathcal{G}$.

The general case in which both translation and rotation is involved, is now easily constructed from both parts:

$$\hat{\mathbf{X}}_{\mathcal{FG}} \begin{bmatrix} \mathbf{1} & \mathbf{0} \\ -\tilde{\mathbf{r}} & \mathbf{1} \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{0} \\ \mathbf{0} & \mathbf{R} \end{bmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{0} \\ -\tilde{\mathbf{r}}\mathbf{R} & \mathbf{R} \end{bmatrix}. \tag{3.19}$$

Using the general *spatial transformation matrix* $\hat{\mathbf{X}}_{\mathcal{FG}}$ the transformation of a vector from $\mathcal{F}$ to $\mathcal{G}$ can be written:

$$\hat{\mathbf{v}}_{\mathcal{G}} = \hat{\mathbf{X}}_{\mathcal{FG}} \hat{\mathbf{v}}_{\mathcal{F}}.$$

This works for all spatial quantities defined in the table above.

Spatial transpose, spatial dot product

The *spatial transpose* $\hat{\mathbf{v}}'$ of a spatial vector $\hat{\mathbf{v}}$ is defined as

$$\hat{\mathbf{x}}' = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}' = \begin{bmatrix} \mathbf{x}_2^T & \mathbf{x}_1^T \end{bmatrix} \tag{3.20}$$

The spatial transpose is denoted with a prime instead of a superscript 'T' to differ it from the normal transpose operator. Although the definition of the spatial transpose operator may seem unusual at first, it is very useful to define the *spatial dot product*. For instance, the spatial dot product of a spatial velocity $\hat{\mathbf{v}}$ and a spatial force $\hat{\mathbf{f}}$ (power) is

$$\hat{\mathbf{f}}'\hat{\mathbf{v}} = \begin{bmatrix} \mathbf{f} \\ \boldsymbol{\tau} \end{bmatrix}' \begin{bmatrix} \boldsymbol{\omega} \\ \mathbf{v} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\tau}^T & \mathbf{f}^T \end{bmatrix} \begin{bmatrix} \boldsymbol{\omega} \\ \mathbf{v} \end{bmatrix} = \boldsymbol{\tau} \cdot \boldsymbol{\omega} + \mathbf{f} \cdot \mathbf{v} \tag{3.21}$$

3.3 Laws and assumptions

3.3.1 Laws

The simulator works on the basis of a number of laws which describe the real world up to certain limits. We use Newtonian physics; we want to simulate 'ordinary' (day to day) systems with an ordinary size and behavior so we have no use for relativity or quantum mechanics.

The first two laws describe how the sum of all forces acting on the bodies are related to the accelerations of the bodies:

Law 1 (Newton's Second Law).

$$\sum \mathbf{f}(t) = m\ddot{\mathbf{x}}(t) \tag{3.22}$$

Law 2 (Euler equations).

$$\sum \boldsymbol{\tau}(t) = \mathbf{I}\boldsymbol{\alpha} + \boldsymbol{\omega}(t) \times \mathbf{I}\boldsymbol{\omega}(t) \tag{3.23}$$

These two laws (together called the Newton-Euler equations) describe the behavior of bodies under the influence of external forces such as gravity or contact forces, and the inertial forces.

The first order counterparts of these two equations (the result of integrating 3.22 and 3.23 over time) relate impulse to change in velocity:

$$\mathbf{p} = m\Delta\mathbf{v} \tag{3.24}$$

$$\boldsymbol{\phi} = \mathbf{I}\Delta\boldsymbol{\omega} \tag{3.25}$$

where $\boldsymbol{\phi} = \mathbf{r} \times \mathbf{p}$ is angular impulse. Equations (3.24) and (3.25) are used to apply an impulse to bodies when they collide.

Law 3 (Coulomb friction law). *If two bodies are in contact at one or more points: if the tangential velocity at the contact point is not zero the tangential contact force maximally opposes acceleration at the contact point. Otherwise the contact force lies within a friction cone. Mathematically this may be stated as:*

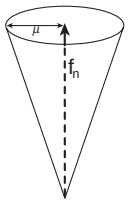

Figure 3.2: The friction cone resulting from the Coulomb friction law for a unit normal force $\mathbf{f}_n$; the total force $\mathbf{f}$ must lie within the cone.

$$\begin{array}{lll} \mathbf{u}_t \neq \mathbf{0} & \Rightarrow & \mathbf{f}_t = -\mu \parallel \mathbf{f}_n \parallel \mathbf{u}_t \\ \mathbf{u}_t = \mathbf{0} & \Rightarrow & \parallel \mathbf{f}_t \parallel \leq -\mu \parallel \mathbf{f}_n \parallel \end{array}$$

Coulomb's friction law states exactly what the direction of the tangential component of contact forces should be in case of dynamic (sliding) friction. In the case of static (non-moving) contact, it only states that the magnitude of the tangential component of force is smaller than the normal component of the contact force multiplied by the friction coefficient μ. The force must thus lie within a cone as depicted in figure 3.2. The Coulomb friction law is used both for contact *forces* and collisional *impulses*. This means that the magnitude of the tangential component $\mathbf{p}_t$ of an impulse $\mathbf{p}$ can never be larger than $\mu \parallel \mathbf{p}_n \parallel$.

3.3.2 Assumptions

A number of assumptions are made to make rigid body simulation viable in real time.

Assumption 1. *All bodies in the simulation are perfectly rigid.*

This means that they can not deform or penetrate each other and that impulses and forces are propagated through them at infinite speed. This implies that collisions happen over an infinitesimal time period. A collision changes the velocity of the involved bodies instantaneously. It also implies that the change in velocity is immediately propagated across the entire body. Thus multiple collisions can happens during the same infinitesimal time period because the colliding bodies can collide with other bodies immediately. The assumption also simplifies impulse computation because forces like gravity are negligible over infinitesimal time periods, and the position of the colliding bodies remain constant during the collision.

Another implication of 1 is that bodies can not vibrate internally. Many mechanical systems such as machines 'suffer' from (or exploit) vibrations yet the exact details of such vibrations are not perfectly understood.

Assumption 2. *Impulses are and forces are applied only at points and this is sufficient to simulate the behavior of bodies in the real world*

In the real world, impulses and forces are applied to approximate points, lines, and surfaces. To simplify computation and application of impulses in a

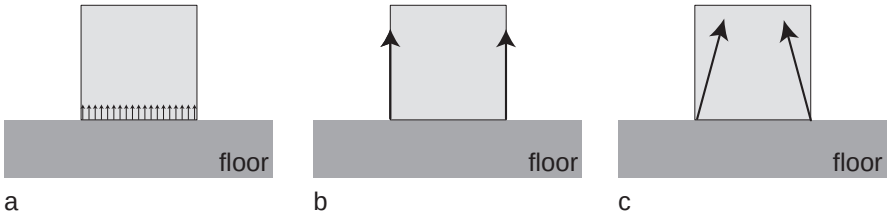

Figure 3.3: a) Force distribution in the real world. b) Force distribution in the simulator. c) Absurd force distribution

rigid body simulator impulses are applied only at points. Imagine a box resting on a surface. In the real word, force would be applied all over the lower face of the box. Though the force would not be equally distributed over the entire face (depending on the face/surface geometry), it would be applied across a large area. In our rigid body simulator, the box would receive impulses or forces only at it's lower vertices. Compare figure 3.3a and 3.3b. In 3.3a force is equally applied all along the lower edge of the 2D box, while in 3.3b, it is only applied to it's two lower vertices. Though the resulting net force at the center of mass of the box might be equal in both cases, 3.3b is not physically realistic. If that were the case, one could just as well argue that 3.3c is physically realistic just because the resulting net force is equal to 3.3a. But since we are not interest in the force distribution but only in the resulting behavior of the bodies, we assume that model 3.3b is just as well suited for this type of rigid body simulation as model 3.3a, because the net resulting force acting on the bodies is equal.

Assumption 3. *Polyhedral objects are geometrically smooth at contact points*

The how and why of this assumption was explained above in the section 2.2 on the representation of bodies in the simulator.

3.4 Impulse Computation and application

3.4.1 Introduction

In an impulse based simulator important aspects of the behavior of the bodies result from the impulse computation module. An impulse changes the velocity of two bodies such that they move away from each other or at least come the a relative halt at the contact or collision point. Impulses are responsible for introducing both friction and restitution into the simulation. Thus we require a physically accurate algorithm. In an impulse based simulator many (micro-) impulses are used to simulate contact, so it must also be efficient. Unfortunately these two goals are hard to combine and we'll have to settle with a good approximation of what happens in nature. We make assumption 1 that the bodies are totally rigid so that we do not have to take full body deformation and vibration into account. We use the simple Coulomb friction law 3,

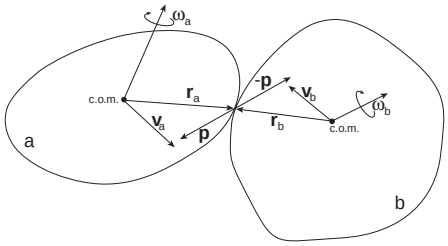

Figure 3.4: A collision between two bodies. $\mathbf{r}_i$ is a vector from the c.o.m. of a body to the contact point

which does not take into account phenomena like contact mechanics, surface chemistry and vibrations.

The choice between two classes of impulse computation algorithms had to be made, *incremental* and *algebraic*. A third class, based on *full body deformation*, was not considered because of efficiency requirements and the complicated models they use. Algebraic impulse computation algorithms compute the impulse directly from the input parameters (which describe the situation at the contact point and the mass properties of the bodies involved) using algebraic equations. Incremental impulse computation algorithms use some kind of model that involves deformation and slip in the contact region. They usually derive differential equations from the rigid body equations, which then have to be integrated (*incrementally*) to compute an impulse. The initial conditions for the integration are computed from the same input parameters as the algebraic laws.

Initially we used an incremental law described in [Mir96]. Although it gave satisfactory results, the algorithm was slow (because of the numerical integration) and complicated to code (because of singularities in the differential equation which had to be avoided). We then implemented an algebraic impulse law by Chatterjee and Ruina [CR98], described in section 3.4.3. This algorithm was easy to implement and is very efficient. Having implementations of both algorithms at our disposal, the choice was simple. We saw no great qualitative advantage of the incremental algorithm over the algebraic one (recall that we are interested in the qualitative results of the simulator, not in it's quantitative results or predictive power, and we are also concerned with the time performance of the simulator), so preferred the algebraic algorithm over the incremental, even thought the incremental law might give slightly more accurate results.

3.4.2 Impulse application

Suppose we have computed an impulse $\mathbf{p}$ which should be applied to body a and an impulse $-\mathbf{p}$ which should be applied to body b, as shown in figure 3.4. How do the velocities $\mathbf{v}_a, \boldsymbol{\omega}_a, \mathbf{v}_b$ and $\boldsymbol{\omega}_b$ change in response to these impulses? Inverting equations 3.24 and 3.25 (derived from the Newton-Euler equations)

tells us what will happen:

$$\Delta \mathbf{v}_i = \frac{\mathbf{p}_i}{m_i} \tag{3.26}$$

$$\Delta \boldsymbol{\omega}_i = \mathbf{I}_i^{-1} \mathbf{r}_i \times \mathbf{p}_i \tag{3.27}$$

3.4.3 Computation of impulse

The collision matrix

An important quantity in computing impulses is the collision matrix $\mathbf{K}$. It tells us how the *contact point velocity* $\mathbf{u}$ will change in response to an impulse $\mathbf{p}$:

$$\Delta \mathbf{u} = \mathbf{K}\mathbf{p} \tag{3.28}$$

where $\mathbf{u}$ is defined as $\mathbf{u}_a - \mathbf{u}_b$; $\mathbf{u}_i$ is the velocity of body i at the contact point. $\mathbf{K}$ can be seen as an matrix filled with three impulse responses $\Delta \mathbf{u}_x$, $\Delta \mathbf{u}_y$ and $\Delta \mathbf{u}_z$, each resulting from an unit impulse in the direction of one of the three axes:

$$\mathbf{K}_i = \begin{bmatrix} \Delta \mathbf{u}_{x_x} & \Delta \mathbf{u}_{y_x} & \Delta \mathbf{u}_{z_x} \\ \Delta \mathbf{u}_{x_y} & \Delta \mathbf{u}_{y_y} & \Delta \mathbf{u}_{z_y} \\ \Delta \mathbf{u}_{x_z} & \Delta \mathbf{u}_{y_z} & \Delta \mathbf{u}_{z_z} \end{bmatrix} \tag{3.29}$$

We could compute $\mathbf{K}$ using equation 3.29, and we *will* in section 3.6.7, but here we will derive how to compute $\mathbf{K}$ directly. Since $\Delta \mathbf{u} = \Delta \left(\mathbf{u}_a - \mathbf{u_b} \right) = \Delta \mathbf{u}_a - \Delta \mathbf{u}_b$, we can separate equation 3.28 into two equations of the form

$$\Delta \mathbf{u}_i = \mathbf{K}_i \mathbf{p}_i. \tag{3.30}$$

Since the change in contact point velocity $\Delta \mathbf{u}_i$ is

$$\Delta \mathbf{u}_i = \Delta \mathbf{v}_i + \Delta \boldsymbol{\omega}_i \times \mathbf{r}_i \tag{3.31}$$

we can write (substituting equations 3.26 and 3.27)

$$\Delta \mathbf{u}_i = \frac{\mathbf{p}_i}{m_i} + \left(\mathbf{I}_i^{-1} \mathbf{r}_i \times \mathbf{p}_i \right) \times \mathbf{r}_i = \left(\frac{1}{m_i} \mathbf{1} - \tilde{\mathbf{r}}_i \mathbf{I}_i^{-1} \tilde{\mathbf{r}}_i \right) \mathbf{p}_i. \tag{3.32}$$

Substituting equation 3.32 twice (once for $\Delta \mathbf{u}_a$ and once for $\Delta \mathbf{u}_b$) into 3.28 and noting $\mathbf{p} = \mathbf{p}_a = -\mathbf{p}_b$ gives:

$$\begin{aligned} \Delta \mathbf{u} = \mathbf{K}\mathbf{p} = \Delta \mathbf{u}_a - \Delta \mathbf{u}_b = \\ \left(\frac{1}{m_a} \mathbf{1} - \tilde{\mathbf{r}}_a \mathbf{I}_a^{-1} \tilde{\mathbf{r}}_a \right) \mathbf{p}_a - \left(\frac{1}{m_b} \mathbf{1} - \tilde{\mathbf{r}}_b \mathbf{I}_b^{-1} \tilde{\mathbf{r}}_b \right) \mathbf{p}_b = \\ \left[\left(\frac{1}{m_a} + \frac{1}{m_b} \right) \mathbf{1} - \left(\tilde{\mathbf{r}}_a \mathbf{I}_a^{-1} \tilde{\mathbf{r}}_a + \tilde{\mathbf{r}}_b \mathbf{I}_b^{-1} \tilde{\mathbf{r}}_b \right) \right] \mathbf{p} \end{aligned} \tag{3.33}$$

from which it is clear that

$$\mathbf{K} = \left(\frac{1}{m_a} + \frac{1}{m_b} \right) \mathbf{1} - \left(\tilde{\mathbf{r}}_a \mathbf{I}_a^{-1} \tilde{\mathbf{r}}_a + \tilde{\mathbf{r}}_b \mathbf{I}_b^{-1} \tilde{\mathbf{r}}_b \right). \tag{3.34}$$

The algorithm

Impulses are computed using an algebraic algorithm described in detail in [CR98]. We decided to use a fast algebraic algorithm instead of the (*possibly* more accurate) incremental algorithm because *any generally applicable collision law, whether coming from detailed continuum modeling, approximating ordinary differential equations, or summarizing functions will be highly approximate unless applied to a narrow range of collisional situations* [CR98].

The impulse computed by the collision law is the weighted sum of two impulses

$$\mathbf{p}_I = -\left(\frac{\mathbf{n}^T\mathbf{u}}{\mathbf{n}^T\mathbf{K}\mathbf{n}}\right)\mathbf{n} \tag{3.35}$$

$$\mathbf{p}_{II} = -\mathbf{K}^{-1}\mathbf{u}, \tag{3.36}$$

where $\mathbf{n}$ is the contact point normal and $\mathbf{u}$ the contact point velocity as above. $\mathbf{p}_I$ brings the normal contact point velocity to 0 (perfectly plastic and frictionless), and $\mathbf{p}_{II}$ causes the entire contact point velocity to become $\mathbf{0}$ (perfectly plastic, sticking). The law uses two restitution parameters e and e_t to specify the weighting between the two impulses. e, ($0 \leq e \leq 1$), specifies the amount of *normal restitution*, and e_t, the *tangential* restitution coefficient, ($-1 \leq e_t \leq 1$), specifies the amount in which the tangential contact point velocity is reversed. A third coefficient μ (from law 3) specifies the size of the friction cone and is used to *clip* the initial candidate impulse

$$e_{can} = (1+e)\,\mathbf{p}_I + (1+e_t)\,(\mathbf{p}_{II} - \mathbf{p}_I)\,. \tag{3.37}$$

The clipping or *friction check* projects the candidate impulse e_{can} computed by 3.37 onto the friction cone specified by Coulomb's friction law.

3.4.4 Micro impulses

What actually makes impulse based rigid body simulation work is the use of *micro impulses*. Imagine what would happen to a body resting on a surface if a simulator only applied 'ordinary' impulses (i.e. as computed by the algorithm in section 3.4.3) to it. Every time the body collides, a little kinetic energy disappears[1] from the system (that is, if the coefficient of restitution e is smaller than 1). The fact that kinetic energy of the body decreases on each collision and the potential energy is transformed into kinetic energy as the body is falling, implies that the body must come closer and closer to the surface, since it contains only a finite amount of kinetic and potential energy. The simulator would have to take smaller and smaller integration steps as the distance between the body and the surface decreases. Eventually the limit would be reached and the simulator would halt.

To solve this problem, 'new' energy is introduced artificially into the system. By increasing the coefficient of restitution e to values above 1 (under certain conditions, explained below), the simulator compensates for the kinetic

[1] In the real world, this energy would be transformed into some other form such as sound, heat or vibrations, but the simulator does not take these quantities into account.

energy which is lost during collisions so that the body can come to rest at a small distance (in the order of the collision epsilon ϵ) of the surface. These collisions with artificially 'boosted' restitution are called *micro collisions*. The impulses which are applied in the event of a micro collision are called *micro impulses*.

The conditions under which micro impulses are applied are that the distance between the bodies must be less than the collision epsilon ϵ, the contact point velocity must be such that the closest points between them are closing (these are the conditions for a standard collision), and the magnitude of the contact point velocity must be lower than some small threshold. This threshold is usually related to the size of the vector of gravity, i.e. $2|g|\epsilon$, since gravity dictates in many situations at what velocity bodies will collide.

The boosted coefficient of restitution e_b can lie anywhere between it's ordinary value of e and the maximum boosted coefficient of restitution e_{max}, depending on the distance between the bodies. e_{max} is a small value > 1, usually chosen somewhere between 2 and 5. Since the bodies are said to collide when the distance between them is smaller than the collision epsilon ϵ, they have a small *collision envelope* around them. e_b is a function of amount of penetration p_ϵ (the range of p_ϵ is $[0, 1]$) of this envelope:

$$e_b = e + (e_{max} - e)\, p_\epsilon \tag{3.38}$$

The idea is that as the envelope gets penetrated more by the bodies, the simulator should force the bodies apart harder.

3.5 Computation of mass properties

The mass properties of a body are important values required for rigid body simulation. We need the mass m of a body for equations (3.22) and (3.24) and to compute the collision matrix $\mathbf{K}$. The inertia tensor is also required to compute the collision matrix $\mathbf{K}$ and for equations (3.23) and (3.25). The center of mass of a body must be known to compute the amount of torque ($\mathbf{r} \times \mathbf{f}$) a force $\mathbf{f}$ generates ($\mathbf{r}$ is the vector from the center of mass to the position where the force is applied). If one can compute the *body frame* $\mathcal{F}_b$ with it's axes aligned with the principle axes of inertia, many dynamics equations are greatly simplified (especially the Featherstone algorithm treated in section 3.6).

The principle axes of inertia are the axes which cause the inertia tensor to have only non-zero values on the diagonal. For every physical body a disk can be found which has the same mass properties. The principle axes of the disk will be aligned with the principle axes of inertia. The center of the disk will be the center of mass of the original object. So the problem of computing the mass properties can be viewed as finding the right disk position, orientation and size for the body. This process is depicted in figure 3.5. The user specifies the geometry of the object at any position and in any orientation. The (initial) mass properties are computed, and finally the body frame is found.

To compute m, $\mathbf{I}$ and the center of mass, my simulator uses an algorithm by Miritch [Mir96]; the implementation we use is also by Mirtich. It efficiently

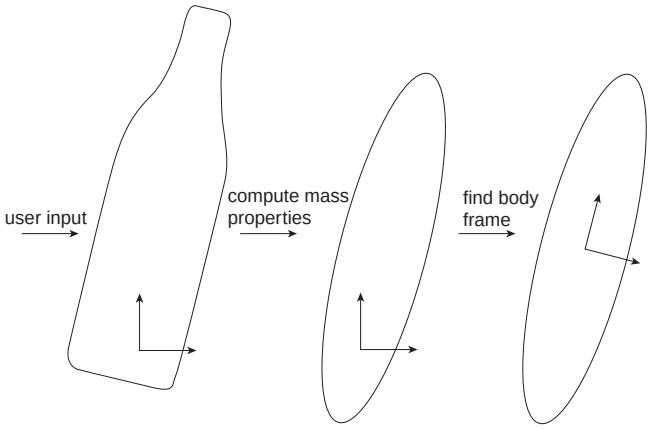

Figure 3.5: The process of computing the mass properties of a body

computes the mass properties for any type of polyhedron as long as it's surface is closed. The algorithm takes as input a description of the faces of the body and the unit mass (it assume mass is distributed equally across the body), and gives as output 10 parameters describing the mass properties of the body:

$$m,$$
$$\mathbf{I}'' = \begin{bmatrix} \mathbf{I}''_{xx} & -\mathbf{I}''_{xy} & -\mathbf{I}''_{xz} \\ -\mathbf{I}''_{yx} & \mathbf{I}''_{yy} & -\mathbf{I}''_{yz} \\ -\mathbf{I}''_{zx} & -\mathbf{I}''_{zy} & \mathbf{I}''_{zz} \end{bmatrix} \text{ and}$$
$$\mathbf{r}$$

The inertia tensor $\mathbf{I}$ is marked with a double prime because it if not the final matrix. It still has to be 'translated' to the c.o.m. $\mathbf{r}$, and diagonalized. Also not that $\mathbf{I}$ is symmetric since $\mathbf{I}''_{xy} = \mathbf{I}''_{yz}$, $\mathbf{I}''_{zy} = \mathbf{I}''_{yx}$ and $\mathbf{I}''_{zx} = \mathbf{I}''_{xz}$. The tensor is translated to the c.o.m. using (see for instance [MK86])

$$\begin{aligned} \mathbf{I}'_{xx} &= \mathbf{I}''_{xx} - m\left(r_y^2 + r_z^2\right) & \mathbf{I}'_{yy} &= \mathbf{I}''_{yy} - m\left(r_z^2 + r_x^2\right) \\ \mathbf{I}'_{zz} &= \mathbf{I}''_{zz} - m\left(r_x^2 + r_y^2\right) & \mathbf{I}'_{xy} &= \mathbf{I}''_{xy} - m\left(r_x + r_y\right) \\ \mathbf{I}'_{yz} &= \mathbf{I}''_{xy} - m\left(r_y + r_z\right) & \mathbf{I}'_{zx} &= \mathbf{I}''_{xy} - m\left(r_z + r_x\right) \end{aligned}$$

Finally, the Jacobi method (see [GvL96] or [PTVF92]) is used to find a frame aligned with the principle axes of inertia. This results in a diagonal inertia tensor $\mathbf{I}$ and a rotation matrix $\mathbf{R}$ which specifies how the body frame is rotated relative to the frame in which the geometry was specified. The mass properties of the body can then be described with just four values m, $\mathbf{I}_{xx}$, $\mathbf{I}_{yy}$ and $\mathbf{I}_{zz}$, although the other values (c.o.m. vector $\mathbf{r}$ and the rotation matrix $\mathbf{R}$) are

retained inside the simulator. When the user of the simulator specifies coordinates regarding the body, they are considered to be relative to the *original* frame in which the geometry was specified.

3.6 The Featherstone algorithm

This chapter discusses how bodies connected by *joints* (called constrained bodies, articulated bodies or multibodies) are handled in my simulator. The simulator must be able to simulate such mechanisms, since this is what the creatures we eventually want to evolve are built from. Also, a much larger class of mechanical systems can be modelled if we support constrained bodies. The problem of simulating constrained bodies is that of computing the accelerations of the joints from the current positions and velocities of bodies and the external forces acting on the bodies.

The Featherstone algorithm is a reduced coordinate method; there are as many coordinates required to describe the system as there are degrees of freedom in it. This means that there are no invalid states and constraints are automatically enforced. The only thing we have to compute and integrate over time are the coordinates describing the system. The advantage of the Featherstone algorithm is it's $O(n)$ time complexity (where n is the number of coordinates required to specify the state of the system). The disadvantage is that it is difficult to understand intuitively and requires complex notation.

Another group of algorithms called *multiplier methods* or *maximal coordinate methods* describe the state of a body with it's maximum number of coordinates. More coordinates are used to describe the state of the system than the degrees of freedom in the system, thus *constraint forces* are used to enforce the constraints imposed by joints. These constraint forces are computed by solving a set of linear equations. The advantage of multiplier methods is that they allow an arbitrary set of constraints to be combined, which is difficult (if not impossible) to do with reduced coordinates methods, and that they are easier to understand than reduced coordinates methods. A disadvantage of multiplier methods are their time complexities; it typically requires $O(n^2)$ time to solve the $n \times n$ matrix system resulting from the constraints forces, although [Bar96] shows that this can also be done in $O(n)$ time. Another disadvantage of multiplier methods is drifting; because the constraint forces are not computed exactly and due to inevitable integration errors, bodies which should stay together at a joint may drift apart somewhat.

This presentation of the Featherstone algorithm is not a tutorial and does not provide a derivation of the equations from the basic equations of rigid body dynamics; this would take some 30 pages and be quite redundant, since others have already discussed the Featherstone algorithm in detail (see [Fea87] or [Lil93] for a theoretical presentation, or [Mir96] for a more hands-on presentation). This section only provides a summary of the Featherstone algorithm and some extensions of it, which are then used to derive an equation which used to apply impulses to joints (the basic algorithm only allows for *forces*). To describe the algorithm we make extensive use of spatial algebra described

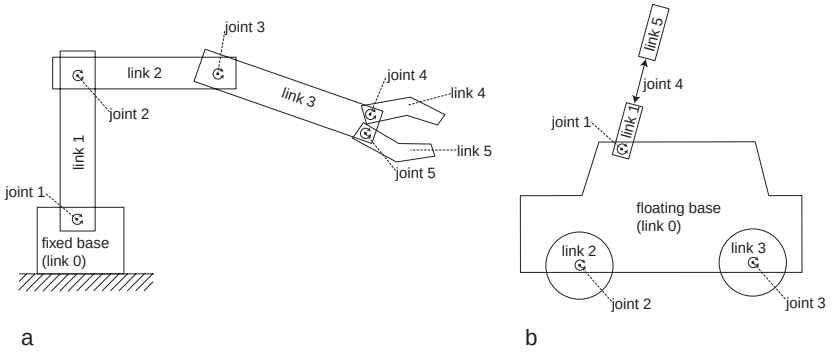

Figure 3.6: a) An articulated body fixed to the ground ('industrial robot') b) A free floating articulated body ('car')

in section 3.2.8; it allows us to write down equations very compact and unify separate linear and angular equations into one equation.

3.6.1 Overview of the Featherstone algorithm

The Featherstone algorithm simulates articulated bodies such as those shown in figure 3.6a and 3.6b. Figure 3.6 also demonstrates the naming convention used in describing such articulated bodies.

Figure 3.6a shows a model of an industrial robot. It's first link (called 'link 0' by convention) is fixed to the ground and thus has 0 d.o.f. The second link (link 1) is connected to link 0 via it's *inboard* joint (joint 1). The *outboard* joint (joint 2) of link 1 connects it to link 2. Every link (except the base link) has an inboard joint, which connects it to it's *parent* link. All joints of the robot are rotational. Some links may have outboard joints, which connect them to their *child* link(s). Note that some links (e.g. link 3) have multiple children; the articulated bodies handled by the Featherstone algorithm must have a tree-like shape. No kinematic loops are allowed. Figure 3.6b shows a model of an car, with a floating base link. It also has a *prismatic* (or translational) joint (the antenna).

The joints and links in a kinematic tree are numbered in a breadth first manner by starting at the base link and traversing all other links breadth first. This is done so all links and joints are visited in the right order by the Featherstone algorithm.

The Featherstone algorithm handles both rotational and prismatic joints. Prismatic joint allow for translations along an axis, such as the antenna of a car. Rotational and prismatic joints are handled exactly the same, except for the computation of the *coriolis vector* (used to compute Coriolis forces) and during the computation of the velocities of links.

The Featherstone algorithm consist of three passes. First it computes the velocities, zero acceleration force and spatial isolated inertia of all links. The zero acceleration force is the force which should be exerted on a link to prevent

it from accelerating if it was isolated from the other links. Thus zero acceleration force includes all external forces (e.g. gravity) and the inertial forces. The spatial isolated inertia (a spatial matrix) is the spatial counterpart of mass and inertia (which are constant). To compute the velocities and zero acceleration forces, the tree of links must be traversed from link 0 to link n, because the velocity of a link is determined by the velocity of it's parent and the joint velocity of that link.

The second pass works from the leaves of the tree to it's root (so from link n to link 0) and computes the *articulated* (as opposed to isolated) inertia and articulated zero acceleration forces for all links. To do this, the information from the first pass is required. The articulated inertia of a link is it's isolated inertia plus the inertia of all it's child links, compensated for their current velocities, translation and rotation. The articulated zero acceleration force is the force which should be exerted on a joint to prevent it from accelerating if it's *inboard* joint was removed.

The last pass computes the accelerations of all joints and links and works from link 0 to link n. The results of the second pass are used to compute these values. Table 3.3 summarizes the notation used in the next three sections to describe these three passes.

3.6.2 Initializing the links and computing velocities

During the first pass, the velocities of all links i are computed. This recursive computation start at link 0, the base link. If it is fixed to an inertial frame, it is initialized as follows:

$$\boldsymbol{\omega}_0, \mathbf{v}_0, \boldsymbol{\alpha}_0, \mathbf{a}_0 = \mathbf{0} \tag{3.39}$$

Otherwise, only the *acceleration* of link 0 is set to $\mathbf{0}$, since velocity is part of the coordinates describing the state of the link. The following equations specify the rest of the computation, which is done for all links i, in order, from link 1 to link n.

$$h = \text{index of link inboard to joint } i$$

$$\hat{\mathbf{X}}_{\mathcal{F}_h \mathcal{F}_i} = \text{spatial transformation matrix from } \mathcal{F}_h \text{ to } \mathcal{F}_i \tag{3.40}$$

$$\hat{\mathbf{s}}_i = \begin{bmatrix} \mathbf{u}_i \\ \mathbf{u}_i \times \mathbf{d}_i \end{bmatrix} \text{ if joint } i \text{ is rotational} \tag{3.41}$$

$$\hat{\mathbf{s}}_i = \begin{bmatrix} \mathbf{0} \\ \mathbf{u}_i \end{bmatrix} \text{ if joint } i \text{ is prismatic} \tag{3.42}$$

$$\hat{\mathbf{v}}_i = \hat{\mathbf{X}}_{\mathcal{F}_h \mathcal{F}_i} \hat{\mathbf{v}}_h + \dot{q}_i \hat{\mathbf{s}}_i \tag{3.43}$$

The spatial transformation matrix computed in (3.40) is constructed from the vector $\mathbf{r}$ from the c.o.m. (center of mass) of link h to the c.o.m. of link i and the rotation matrix $\mathbf{R}$ from body frame of link h to the body frame of link, as shown in section 3.2.8. For each link, after the velocities have been computed, the zero acceleration force, inertia and coriolis vector are initialized to the values they

Table 3.3: Featherstone algorithm notation, copied from [Mir96]

n	number of links of serial linkage
$\mathbf{v}_i$	linear velocity of link i
$\boldsymbol{\omega}_i$	angular velocity of link i
$\mathbf{a}_i$	linear acceleration of link i
$\boldsymbol{\alpha}_i$	angular acceleration of link i
$\mathbf{v}_{rel}$	relative linear velocity of link i
$\boldsymbol{\omega}_{rel}$	relative angular velocity of link i
m_i	mass of link i
$\mathbf{M}_i$	matricized mass of link i
$\mathbf{I}_i$	diagonalized (body frame) inertia of link i
$\mathbf{d}_i$	vector from link i inboard joint to link i center of mass
$\mathbf{r}_i$	vector from the parent of link i c.o.m. to link i c.o.m.
$\mathbf{u}_i$	unit vector in the direction of joint i axis
$\hat{\mathbf{I}}_i$	spatial isolated inertia of link i
$\hat{\mathbf{I}}_i^A$	spatial articulated inertia of link i
$\hat{\mathbf{Z}}_i$	spatial isolated zero acceleration force of link i
$\hat{\mathbf{Z}}_i^A$	spatial articulated zero acceleration force of link i
$\hat{\mathbf{f}}_i^I$	spatial force applied by inboard joint to link i
$\hat{\mathbf{f}}_i^O$	spatial force applied by outboard joint to link i
$\hat{\mathbf{v}}_i$	spatial velocity of link i
$\hat{\mathbf{a}}_i$	spatial acceleration of link i
$\hat{\mathbf{s}}_i$	spatial joint axis of joint i
$\hat{\mathbf{c}}_i$	spatial Coriolis force for link i
q_i	scalar position of joint i
$\dot{q}_i$	scalar velocity of joint i
$\ddot{q}_i$	scalar acceleration of joint i
Q_i	joint actuator force of joint i
$\boldsymbol{\nu}_i$	vector velocity of joint i ($\dot{q}\mathbf{u}$)
$\boldsymbol{\xi}_i$	vector acceleration of joint i ($\ddot{q}\mathbf{u}$)

would have in isolation:

$$h = \text{index of link inboard to joint } i$$

$$\hat{\mathbf{Z}}_i^A = \begin{bmatrix} -m_i \mathbf{g} \\ \boldsymbol{\omega}_i \times \mathbf{I}_i \boldsymbol{\omega}_i \end{bmatrix} \tag{3.44}$$

$$\hat{\mathbf{I}}_i^A = \begin{bmatrix} 0 & \mathbf{M}_i \\ \mathbf{I}_i & 0 \end{bmatrix} \tag{3.45}$$

$$\hat{\mathbf{c}}_i = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{\omega}_h \times (\boldsymbol{\omega}_h \times \mathbf{r}_i) + 2\boldsymbol{\omega}_h \times \boldsymbol{\nu}_i \end{bmatrix} \tag{3.46}$$

if joint i is prismatic, or

$$\hat{\mathbf{c}}_i = \begin{bmatrix} \boldsymbol{\omega}_h \times \boldsymbol{\nu}_i \\ \boldsymbol{\omega}_h \times (\boldsymbol{\omega}_h \times \mathbf{r}_i) + 2\boldsymbol{\omega}_h \times (\boldsymbol{\nu}_i \times \mathbf{d}_i) + \boldsymbol{\nu}_i \times (\boldsymbol{\nu}_i \times \mathbf{d}_i) \end{bmatrix} \tag{3.47}$$

if joint i is rotational.

External forces other than gravity (such as controller of contact forces) should also be included in equation (3.44). The coriolis vector computed in (3.46) and (3.47) is used to unify computation of the Coriolis forces of both rotational and prismatic joints. Note that the isolated spatial inertia (3.45) is constant and has to be computed only once.

3.6.3 Computing the articulated zero acceleration forces and inertias

The articulated zero acceleration and articulated inertias are computed from the results of the first pass. The recursive computation proceeds from the leaves of the tree to the base link; both the articulated zero acceleration force and the articulated inertia of a node must be known before the articulated zero acceleration force and the articulated inertia of it's parent can be computed. The following equations describe this:

$$h = \text{index of link inboard to joint } i$$

$$\hat{\mathbf{I}}_h^A = \hat{\mathbf{I}}_h^A + \hat{\mathbf{X}}_{\mathcal{F}_i \mathcal{F}_h} \left[\hat{\mathbf{I}}_i^A - \frac{\hat{\mathbf{I}}_i^A \hat{\mathbf{s}}_i \hat{\mathbf{s}}_i' \hat{\mathbf{I}}_i^A}{\hat{\mathbf{s}}_i' \hat{\mathbf{I}}_i^A \hat{\mathbf{s}}_i} \right] \hat{\mathbf{X}}_{\mathcal{F}_h \mathcal{F}_i} \tag{3.48}$$

$$\hat{\mathbf{Z}}_h^A = \hat{\mathbf{Z}}_h^A + \hat{\mathbf{X}}_{\mathcal{F}_i \mathcal{F}_h} \left[\hat{\mathbf{Z}}_i^A + \hat{\mathbf{I}}_i^A \hat{\mathbf{c}}_i + \frac{\hat{\mathbf{I}}_i^A \hat{\mathbf{s}}_i \left[Q_i - \hat{\mathbf{s}}_i' \left(\hat{\mathbf{Z}}_i^A + \hat{\mathbf{I}}_i^A \hat{\mathbf{c}}_i \right) \right]}{\hat{\mathbf{s}}_i' \hat{\mathbf{I}}_i^A \hat{\mathbf{s}}_i} \right] \tag{3.49}$$

3.6.4 Computing the accelerations of the joints

When the forces $\hat{\mathbf{Z}}_i^A$ acting on the links and the inertias $\hat{\mathbf{I}}_i^A$ are known, the accelerations of the joints and links can be computed (starting at link 1, proceeding to link n):

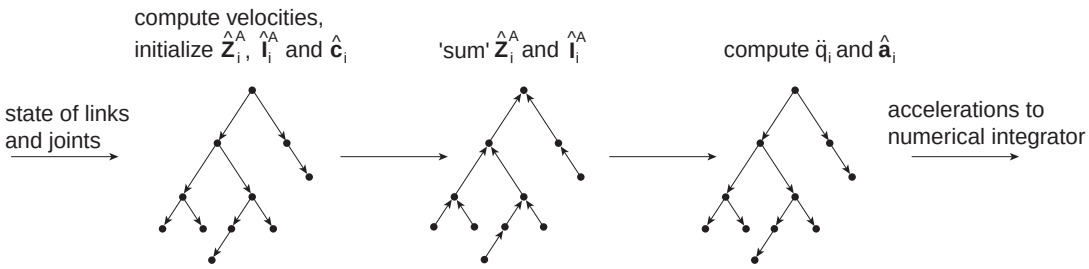

Figure 3.7: Featherstone algorithm algorithm summary

$$h = \text{index of link inboard to joint } i$$

$$\ddot{q}_i = \frac{Q_i - \hat{\mathbf{s}}_i' \hat{\mathbf{I}}_i^A \hat{\mathbf{X}}_{\mathcal{F}_h \mathcal{F}_i} \mathbf{a}_h - \hat{\mathbf{s}}_i' \left(\hat{\mathbf{Z}}_i^A + \hat{\mathbf{I}}_i^A \hat{\mathbf{c}}_i \right)}{\hat{\mathbf{s}}_i' \hat{\mathbf{I}}_i^A \hat{\mathbf{s}}_i} \tag{3.50}$$

$$\hat{\mathbf{a}}_i = \hat{\mathbf{X}}_{\mathcal{F}_h \mathcal{F}_i} \hat{\mathbf{a}}_h + \hat{\mathbf{c}}_i + \ddot{q}_i \hat{\mathbf{s}}_i \tag{3.51}$$

3.6.5 Featherstone algorithm algorithm summary

Figure 3.7 gives a summary of the Featherstone algorithm. The algorithm takes as input the morphology, mass properties, position and velocities describing the current state of all links. It then iterates over all link three times. During the first pass, which proceeds from the base link to the leaves of the tree, the spatial articulated zero acceleration force is initialized to it's isolated counterpart, the spatial inertia is initialized using the mass properties of the individual links and the Coriolis vector is computed. The second pass 'sums' the isolated spatial zero acceleration forces and inertias to compute the *articulated* spatial zero acceleration forces and inertias. This pass works from the leaves to the root. The last pass computes the accelerations of the joints and links from the *articulated* spatial zero acceleration forces and inertias. The accelerations are then fed to a numerical integrator which integrates the state of the links and joints over time.

3.6.6 Computing the acceleration of a floating base

One important detail has not been treated yet: how to compute the spatial acceleration of a floating base link (such as the body of the car in figure 3.6b. During the second pass, this acceleration is set to $\mathbf{0}$ by (3.39). Before starting the third pass, the acceleration of link 0 is computed using the following equation:

$$\hat{\mathbf{a}}_0 = -\left(\hat{\mathbf{I}}_0^A \right)^{-1} \hat{\mathbf{Z}}_0^A \tag{3.52}$$

The 6×6 spatial articulated inertia matrix of link 0 has to be inverted for this computation.

3.6.7 Handling impulses

The basic Featherstone algorithm treated so far is meant for forward dynamics. It computes the accelerations of the joints and link given the current state (joint positions, velocities) and forces acting on the system. These can then be fed to a numerical integrator to forward the time. However, the articulated bodies must also be able to handle *impulses*. To fit the Featherstone algorithm into the impulse based simulation scheme, we require some algorithm to handle impulses. The requirements of the algorithm, which is described in detail in [Mir96], are:

- it must be able to compute the collision matrix $\mathbf{K}$ for the colliding bodies and
- it must be able to change the velocities of a articulated body in response to an impulse $\mathbf{p}$ applied at a position $\mathbf{r}$

If we have an algorithm which can do the latter, we can compute the first by applying *unit test impulses* in the directions of the x, y and z axis, as mentioned in section 3.4.3. The matrix $\mathbf{K}$ relates impulses and changes in velocities according to the linear equation

$$\Delta \mathbf{v} = \mathbf{K}\mathbf{p}.$$

Recall that the 3×3 matrix $\mathbf{K}$ is the sum of two matrices $\mathbf{K}_1$ and $\mathbf{K}_2$. Each of these two matrices $\mathbf{K}_1$ and $\mathbf{K}_2$ describe the situation of one the bodies involved in the collision, and they can be computed independently from each other. If we know $\Delta \mathbf{v}_x$ for $\mathbf{p}_x = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T$, $\Delta \mathbf{v}_y$ for $\mathbf{p}_y = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T$ and $\Delta \mathbf{v}_z$ for $\mathbf{p}_z = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$, then we can construct the appropriate matrix $\mathbf{K}_i$ from $\Delta \mathbf{v}_x$, $\Delta \mathbf{v}_y$ and $\Delta \mathbf{v}_z$

$$\mathbf{K}_i = \begin{bmatrix} \Delta \mathbf{v}_{x_x} & \Delta \mathbf{v}_{y_x} & \Delta \mathbf{v}_{z_x} \\ \Delta \mathbf{v}_{x_y} & \Delta \mathbf{v}_{y_y} & \Delta \mathbf{v}_{z_y} \\ \Delta \mathbf{v}_{x_z} & \Delta \mathbf{v}_{y_z} & \Delta \mathbf{v}_{z_z} \end{bmatrix}. \tag{3.53}$$

Thus, to integrate the Featherstone algorithm into the simulator, we require an algorithm which can compute how articulated bodies respond to impulses. The key to deriving the right equations for the algorithm we require is computing the limit of the integral over time of certain second order equations from the previous section; an impulse could be considered to be an infinitely large force applied over an infinitesimal interval. We need the limit as $t \to 0$ of the integral of the three equations which allow us compute the spatial articulated zero acceleration force, the joint accelerations and the link accelerations. The impulse handling algorithm will have the same multi-pass structure as the

Featherstone algorithm, and could be considered to be a 'first order' version of the 'second order' Featherstone algorithm.

The first step is to compute the velocities of links, to initialize the isolated spatial inertias, and to set the isolated spatial zero acceleration *impulses* to their correct values. Since finite forces (such as gravity and inertial forces) have no effect over an infinitesimal interval, the isolated spatial zero acceleration *impulses* of all links except the colliding link will be $\hat{\mathbf{0}}$. The colliding link receives an impulse $\mathbf{p}$ (in the inertial world frame $\mathcal{W}$), which is transformed to the body frame $\mathcal{F}$ by a spatial transformation matrix.

$$\hat{\mathbf{p}} = \hat{\mathbf{X}}_{\mathcal{W}\mathcal{F}} \begin{bmatrix} \mathbf{0} \\ \mathbf{p} \end{bmatrix}$$

This isolated spatial zero acceleration *impulse* of the colliding link is set to $-\hat{\mathbf{p}}$. The next step is to propagate all spatial isolated zero acceleration impulses and spatial isolated inertias to the base link. The computation of the inertias stays exactly the same as in the Featherstone algorithm. The propagation of zero acceleration impulses becomes simpler because the equation contains some terms which remain finite during a collision. To derive the equation to propagate impulses up the tree, we evaluate the following limit (derived from equation (3.49) used to propagate zero acceleration forces):

$$\lim_{t\to 0} \int_0^t \hat{\mathbf{Z}}_h^A =$$

$$\lim_{t\to 0} \int_0^t \left\{ \hat{\mathbf{Z}}_h + \sum_{j=1}^{m} \hat{\mathbf{X}}_{\mathcal{F}_{i_j}\mathcal{F}_h} \left[\hat{\mathbf{Z}}_{i_j}^A + \hat{\mathbf{I}}_{i_j}^A \hat{\mathbf{c}}_{i_j} + \frac{\hat{\mathbf{I}}_{i_j}^A \hat{\mathbf{s}}_{i_j} \left[Q_{i_j} - \hat{\mathbf{s}}'_{i_j} \left(\hat{\mathbf{Z}}_{i_j}^A + \hat{\mathbf{I}}_{i_j}^A \hat{\mathbf{c}}_{i_j} \right) \right]}{\hat{\mathbf{s}}'_{i_j} \hat{\mathbf{I}}_{i_j}^A \hat{\mathbf{s}}_{i_j}} \right] \right\} dt$$

Joint forces Q_{i_j} and Coriolis forces $\hat{\mathbf{I}}_{i_j}^A \hat{\mathbf{c}}_{i_j}$ stay finite during the collision, so these can be discarded. By defining zero acceleration impulses as

$$\hat{\mathbf{Y}}_i = \lim_{t\to 0} \int_0^t \hat{\mathbf{Z}}_i dt, \tag{3.54}$$

we can simplify the above to:

$$\hat{\mathbf{Y}}_h^A = \hat{\mathbf{Y}}_h + \sum_{j=1}^{m} \hat{\mathbf{X}}_{\mathcal{F}_{i_j}\mathcal{F}_h} \left[\mathbf{1} - \frac{\hat{\mathbf{I}}_{i_j}^A \hat{\mathbf{s}}_{i_j} \hat{\mathbf{s}}'_{i_j}}{\hat{\mathbf{s}}'_{i_j} \hat{\mathbf{I}}_{i_j}^A \hat{\mathbf{s}}_{i_j}} \right] \hat{\mathbf{Y}}_{i_j}^A \tag{3.55}$$

The third and final pass propagates the changes in velocity of the joints and links from the base link towards the leaves of the tree. We compute the limits of equations (3.50) and (3.51) to do just that:

$$\lim_{t\to 0} \int_0^t \ddot{q}_i dt = \lim_{t\to 0} \int_0^t \frac{Q_i - \hat{\mathbf{s}}'_{i_j} \hat{\mathbf{I}}_{i_j}^A \hat{\mathbf{X}}_{\mathcal{F}_h\mathcal{F}_{i_j}} \hat{\mathbf{a}}_h - \hat{\mathbf{s}}'_{i_j} \left(\hat{\mathbf{Z}}_{i_j}^A + \hat{\mathbf{I}}_{i_j}^A \hat{\mathbf{c}}_{i_j} \right)}{\hat{\mathbf{s}}'_{i_j} \hat{\mathbf{I}}_{i_j}^A \hat{\mathbf{s}}_{i_j}} dt \tag{3.56}$$

which becomes

$$\Delta\dot{q}_i = -\frac{\hat{\mathbf{s}}'_{i_j}}{\hat{\mathbf{s}}'_{i_j}\hat{\mathbf{I}}^A_{i_j}\hat{\mathbf{s}}_{i_j}}\left[\hat{\mathbf{I}}^A_{i_j}\hat{\mathbf{X}}_{\mathcal{F}_h\mathcal{F}_{i_j}}\Delta\hat{\mathbf{v}}_h - \hat{\mathbf{Z}}^A_{i_j}\right]. \tag{3.57}$$

And simplifying

$$\lim_{t\to 0}\int_0^t \hat{\mathbf{a}}_i dt = \lim_{t\to 0}\int_0^t \left(\hat{\mathbf{X}}_{\mathcal{F}_h\mathcal{F}_{i_j}}\hat{\mathbf{a}}_h + \ddot{q}_i\hat{\mathbf{s}}_i + \hat{\mathbf{c}}_i\right) dt \tag{3.58}$$

gives us

$$\Delta\hat{\mathbf{v}}_i = \hat{\mathbf{X}}_{\mathcal{F}_h\mathcal{F}_{i_j}}\Delta\hat{\mathbf{v}}_h + \Delta\dot{q}_i\hat{\mathbf{s}}_i \tag{3.59}$$

3.6.8 Extension to the articulated body impulse handling algorithm

Our simulator uses an extended version of the impulse handling algorithm described above which can handle impulses applied at *joints*.

Joint impulses

To enforce *joint limits* (limits on the amount of rotation or translation allowed at a joint), one could use a penalty method (like stiff springs). But the entire simulator is impulse based already, so these limits are more efficiently enforced using (micro-impulses). Mirtichs algorithm assumes however that joint forces are finite during the collision and discards the Q_{i_j} terms. We need to make similar derivations as above, this time including the Q_{i_j} terms.

The limit of (3.49) is

$$\lim_{t\to 0}\int_0^t \hat{\mathbf{Z}}^A_h =$$

$$\lim_{t\to 0}\int_0^t \left\{\hat{\mathbf{Z}}_h + \sum_{j=1}^{m}\hat{\mathbf{X}}_{\mathcal{F}_{i_j}\mathcal{F}_h}\left[\hat{\mathbf{Z}}^A_{i_j} + \hat{\mathbf{I}}^A_{i_j}\hat{\mathbf{c}}_{i_j} + \frac{\hat{\mathbf{I}}^A_{i_j}\hat{\mathbf{s}}_{i_j}\left[Q_{i_j} - \hat{\mathbf{s}}'_{i_j}\left(\hat{\mathbf{Z}}^A_{i_j} + \hat{\mathbf{I}}^A_{i_j}\hat{\mathbf{c}}_{i_j}\right)\right]}{\hat{\mathbf{s}}'_{i_j}\hat{\mathbf{I}}^A_{i_j}\hat{\mathbf{s}}_{i_j}}\right]\right\} dt.$$

To extend the algorithm to handle *joint*, we must evaluate this limit again, this time *not* assuming joint forces Q_{i_j} are finite during the collision. The result is (using the same definition $\hat{\mathbf{Y}}_i = \lim_{t\to 0}\int_0^t \hat{\mathbf{Z}}_i dt$ as before):

$$\hat{\mathbf{Y}}^A_h = \hat{\mathbf{Y}}_h + \sum_{j=1}^{m}\hat{\mathbf{X}}_{\mathcal{F}_{i_j}\mathcal{F}_h}\left[\hat{\mathbf{Y}}^A_{i_j} + \frac{\left[Q_{i_j} - \hat{\mathbf{s}}'_{i_j}\hat{\mathbf{Y}}^A_{i_j}\right]\hat{\mathbf{I}}^A_{i_j}\hat{\mathbf{s}}_{i_j}}{\hat{\mathbf{s}}'_{i_j}\hat{\mathbf{I}}^A_{i_j}\hat{\mathbf{s}}_{i_j}}\right] \tag{3.60}$$

The limit of equation (3.50) also needs to be reevaluated. Equation (3.51) stays the same since it contains no Q_{i_j} terms. Simplifying

$$\lim_{t\to 0}\int_0^t \ddot{q}_i dt = \lim_{t\to 0}\int_0^t \frac{Q_i - \hat{\mathbf{s}}'_{i_j}\hat{\mathbf{I}}^A_{i_j}\hat{\mathbf{X}}_{\mathcal{F}_h\mathcal{F}_{i_j}}\hat{\mathbf{a}}_h - \hat{\mathbf{s}}'_{i_j}\left(\hat{\mathbf{Z}}^A_{i_j} + \hat{\mathbf{I}}^A_{i_j}\hat{\mathbf{c}}_{i_j}\right)}{\hat{\mathbf{s}}'_{i_j}\hat{\mathbf{I}}^A_{i_j}\hat{\mathbf{s}}_{i_j}} dt \tag{3.61}$$

results in

$$\Delta\dot{q}_i = \frac{Q_i - \hat{\mathbf{s}}'_{i_j}\left(\hat{\mathbf{I}}^A_{i_j}\hat{\mathbf{X}}_{\mathcal{F}_h\mathcal{F}_{i_j}}\Delta\hat{\mathbf{v}}_h - \hat{\mathbf{Z}}^A_{i_j}\right)}{\hat{\mathbf{s}}'_i\hat{\mathbf{I}}^A_i\hat{\mathbf{s}}_i} \tag{3.62}$$

By replacing equation (3.55) and (3.57) with (3.60) and (3.62) respectively, the algorithm can handle impulses applied at the joints as well as at the surface of the bodies.

3.7 Controllers

Inside the simulator two types of controllers are provided: high level controllers and low level controllers. Controllers can exert some form of control over the simulation by applying forces or impulses to bodies.

Low level controllers do their work inside the inner integration loop. Every time the forces acting on the bodies are computed, the low level controller functions are called. The low level controller functions compute and apply forces or torques. This can be thousands of times per second of real time, depending on the step size of the numerical integrator. Low level controllers are used to implement basic properties of a physical system, like friction, dampers and springs or to apply constant forces to some body or joint. As a side node, gravity could be seen as a low level controller.

High level controllers are usually more 'intelligent' active components inside the system. High level controllers work by either applying impulses to bodies or by changing the properties (e.g. the target value of a spring) of low level controllers. Usually high level controllers invoke some kind of algorithm which dictates the behaviors of the controller, but in interactive situations humans can be high level controllers as well. High level controllers are implemented as timers. They are called periodically (e.g. at 10hz), do their work and reschedule a controller event. Timers also have other uses, such as invoking a function which stores the current state of the simulation to disk so the result can be rendered or reviewed at a later time.

3.8 Simulator Summary

Having read about all separate algorithms and techniques involved in rigid body simulation, the reader might have lost a clear view of the big picture; in

this section we will give a summary of how all the parts of the simulator fit together.

A typical run of the simulator consists of two parts. The first part is specification and initialization of the physical system which is to be simulated, the second is simulating the system, which is done is the main loop.

3.8.1 Initializing the simulator

During initialization of the simulation specifications of all bodies and their relations are entered into the simulator. For all bodies an initial position and orientation must be specified. Their geometry, (hierarchies of) convex polyhedra, is stored. The geometries are used for collision detection (chapter 2), to compute the convex hulls of the whole bodies in case the geometry is a hierarchy (section 2.4), and to compute the mass and moments of inertia (section 3.5). The mass and inertia tensor of an object are computed immediately when a body's geometry is specified. In case a linkage of bodies connected by joints must be simulated, the joints must be specified during the initialization phase as well.

After all bodies have been added to the simulator, the hierarchical hash table is initialized (section 2.3) using the initial bounding boxes (which are computed from the geometry of the bodies) of all bodies. The hash table reports all bodies which could possibly penetrate. These bodies are checked for penetration using the V-Clip algorithm (see section 2.4). If bodies are penetrating initially the simulator can not proceed and reports an error.

3.8.2 Main loop of the simulator

After the simulator has been initialized it enters the main loop. The main loop consists five parts; impulse computation and application, computing the maximum size of the next integration step, updating the hierarchical hash table, and doing the actual dynamic integration.

During the first part of the main loop, which is called the *collision check*, all bodies which are colliding (the distance between their closest points is smaller than the collision epsilon ϵ and the relative velocity of their closest points is such that the distance between them is decreasing) receive impulses. The impulses must be such that the distances between the closest points are no longer decreasing. The other properties of the impulse are dictated by the physical properties (such as mass, restitution and friction) of the bodies involved. The impulses are computed by the impulse computation algorithm (section 3.4), which requires the collision matrix $\mathbf{K}$ describing situation at the collision point. If one of the colliding bodies is part of an articulated body, the computation of $\mathbf{K}$ is quite involved (section 3.6.7). After the impulse computation and application phase is done, the closest points of all colliding bodies have come to a relative stop or are moving apart. Also, all joint limits have been enforced; they are dealt with similarly as ordinary collisions.

The second part of the main loop of the simulator is to estimate the maximum size of the next integration step. This is the minimum of three values.

The first value is the maximum step the user allows the simulator to take, the second value is the estimated lower bound on the time when the next collision will occur and the third value is the time until the next scheduled timer (e.g. a high level controller) expires.

The lower bound on the time when the next collision will occur is based on the current dynamic state of the bodies which are currently being tracked by the hierarchical hash table as 'possibly colliding'. The dynamic state of these bodies can be used to predict a lower bound on the time when they will collide. We do not wish to integrate beyond that time, otherwise we could fail to detect certain collisions. Imagine for instance a bullet speeding towards a sheet of paper. If we would take a too large integration step, the bullet could pass through the paper without a collision being detected.

Periodically (i.e. 20 times per second) the hierarchical hash table is updated. This is done by taking the current dynamic state of the bodies (position, velocity and acceleration), and predicting a bounding box inside which the body will stay during the next period. Of course this bounding box could be violated by the body. That problem is solved by checking for bounding box violations during dynamic integration. If a body violates the predicted bounding box, the dynamic integration step is undone and the hierarchical hash table updated using the information now available (i.e. where and when the bounding box will be violated).

The last step in the main loop is dynamic integration. This is performed by a dynamic integrator (e.g. a fourth order Runge Kutta integrator with step doubling, see [PTVF92]). The integration algorithm is provided with the current state of the bodies (the positions $\mathbf{x}$ and velocities $\dot{\mathbf{x}}$ and an algorithm to compute the accelerations $\ddot{\mathbf{x}}$. The integrator uses these values to solve the differential equations

$$\frac{d\mathbf{x}}{dt} = \dot{\mathbf{x}} \tag{3.63}$$

$$\frac{d\dot{\mathbf{x}}}{dt} = \ddot{\mathbf{x}} = f\left(\mathbf{x}, \dot{\mathbf{x}}, t\right). \tag{3.64}$$

During each 'sub step' (a typical fourth order Runge Kutta integrator takes four sub steps per step), the bounding boxes are checked for violation and the bodies are checked for penetration. If bounding boxes are violated, the integrator is interrupted are the bounding boxes are adjusted such that the violations will not occur again. If bodies are penetrating, the integrator stops as well, and a smaller step is attempted. Otherwise the integrator continues until the total step size specified by the user has been simulated.

After the integration part of the main loop is done, the simulator checks for collisions again (step 1). This process is repeated until the entire interval requested by the user has been simulated.

3.9 Implementation

The simulator described in this chapter was implemented as a collection of C++ classes and a number of independent algorithms (such as impulse com-

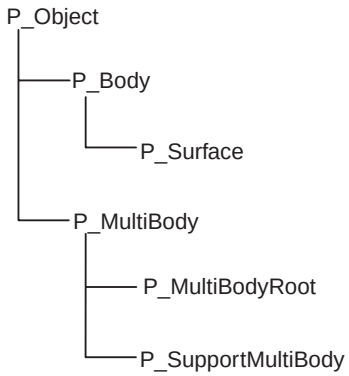

Figure 3.8: The C++ class hierarchy of P_Object

putation) also implemented in C++. The collision detection module is entirely separated from the simulator and is contained in a C library.

Four algorithms were not implemented by us but taken from other sources, namely the jacobi transformation algorithm used to compute the principle axes of inertia of a body (taken from [PTVF92]), a function to compute the SVD-'inverse' of a rank deficient matrix (also taken from [PTVF92]), the Quickhull algorithm for computing convex hulls (from [BDH96]) and a function to compute the mass properties of a body (as described in section 3.5; taken from source code provided by B. Mirtich). All other algorithms and source code were implemented by the author.

The central classes are those representing the bodies inside the simulator. As can be seen in the class hierarchy in figure 3.8, the lowest common denominator for all types of bodies inside the simulator is P_Object ('P' stands for Physical). It contains the position, orientation, linear and angular velocity, controllers, physical properties, geometry, V-Clip and mass properties information which is common to every type of body. Several functions, such as drawing and setting the physical properties of a body, are provided by P_Object. It also contains a number of pure virtual functions which should be implemented by the subclasses. The most important onces are those which can handle an impulse applied to the body, compute the collision properties for a collision and compute the velocity of the body at a certain point.

P_Object itself is never used in a simulator since it contains *pure virtual member functions*. These functions should be implemented by classes derived from P_Object. For instance, P_Body represents 'ordinary' unconstrained bodies. This type of body was relatively easy to implement. *Surfaces* (the P_Surface class) are derived from P_Body, because they are very similar to unconstrained bodies. all bodies inside the simulator which can not move (such as floors and walls) are called surfaces. They are implemented as P_Bodies with very high mass which will not move in response to forces or impulses.

Constrained bodies, whose time derivatives are computed by the Featherstone algorithm , are represented by the P_MultiBody class, which is de-

rived directly from P_Object. The P_MultiBody class contains the entire implementation of the Featherstone algorithm, except for the algorithms concerned with free floating multibodies. Those algorithms are implemented in the P_MultiBodyRoot class, which represents the root of a tree of constrained bodies. A special case is the P_SupportMultiBody 'class', which is actually just a P_MultiBody with neither mass nor geometry. The P_SupportMultiBodies are inserted in the tree between normal P_MultiBodies to implement joints with more than 1 d.o.f. Note that a single P_MultiBodyRoot with no P_MultiBodies connected to it will display the same behavior as a P_Body.

All bodies are contained in a class named P_World; it orchestrates the entire simulation. P_World contains the main loop of the simulator, does the bookkeeping of bodies, contains the hierarchical hash table and V-Clip library instance which are both used for collision detection. It is able to draw the entire 'world' inside the simulator using OpenGL. It can also write the state of the simulation to a file or network socket so that the world can be rendered elsewhere (either in real-time or after simulation has finished).

Two types of controllers are provided. Low level controllers, represented by the class P_LLController, are invoked from inside the inner integration loop. High level controllers are implemented as functions invoked by a timer (class P_Timer). The timer class is also used to implement features such as recording the state of the scene periodically and doing certain experiments with the simulator which require periodic measurements of the state of the simulation.

A number of other algorithms were required to successfully build a simulator, such as a numerical integrator, an impulse computation algorithm and matrix and vector mathematics classes. These are implemented in separate C++ files and could be reused in other projects.

3.10 Improving contact handling

The most 'disturbing' property of impulse based simulation is the fact the bodies never settle completely while they should be at rest. They always bounce a little because the contact forces that keep bodies from penetrating in the real world are replaced by micro impulses. Although the micro impulses are also one of the merits of impulse based simulation, allowing it to efficiently simulate situations where other types of simulator fail, this bouncing behavior can lead to instability and simulation problems (experiments demonstrating the problems will be given in next section).

First of all the bouncing can cause instability in certain physical systems, e.g. stacks or walls of bodies. The bouncing can cause bodies to slowly slide of each other; this will cause stacks of bodies to collapse after a certain time.

Secondly, small systematic errors due to the way micro impulses are delivered to the bodies can accumulate and cause bodies to spin about an axis parallel to the vector of gravity and through the center of mass of the body. A situation where this can occur is shown in figure 3.9a. A cube rests on a surface and receives micro impulses at each of it's four corners. An impulse based simulator as described in [Mir96] will only apply micro impulses to the

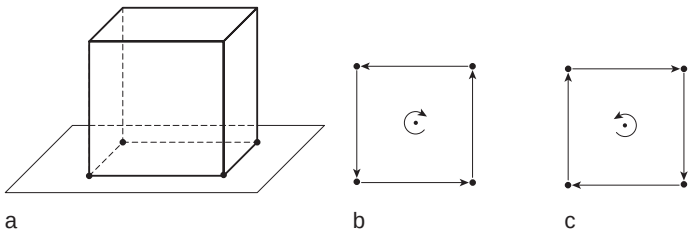

Figure 3.9: a) A cube at rest on a surface. The corners at which the cube receives micro impulses are marked with large points. b) The same cube seen from above. The arrows connecting the points (corners) represent the order in which the micro impulses are delivered over time. The cube will spin clockwise about an axis perpendicular to the paper. c) The same cube, but the delivery of micro impulses has settled into the opposite order, causing it to spin in the opposite direction.

corner which is *closest* to the surface. Over time, each of the four corners will receive impulses because the body will rotate slightly in response to the micro impulses. We found that often the order in which the corners receive impulses will settle into a fixed order, like in figure 3.9b and figure 3.9c. This will cause small but systematic errors and the body will start to spin (very slowly) about the axis parallel to the vector of gravity, largely independent of the amount of friction between the surface and the body. The order shown in figure 3.9b will cause the cube to rotate clockwise (seen from above), the order in 3.9c counter clockwise. To which order the system settles is decided by the initial conditions of the simulation, since it seems to depend the orientation of the body when it starts receiving micro impulses. The spinning effect can be as much a 360 degrees for every 1 minute of simulated time.

Under the hypothesis that these instabilities and errors are (at least partially) caused by the fact that pairs of bodies receive impulses only at their *closest* points, we made an attempt to alleviate the problems by applying impulses at all *contact points*. This is done by tracking the closest points between features (vertices, edges, faces) which have received micro impulses recently. The exact location of these *contact points* has to be recomputed each time micro impulses should be applied, because bodies can move relative to each other.

The method is to apply micro impulses not only at the closest points between two bodies, but to all *valid* contact points which are moving towards each other. Contact points are valid if both closest points lie in the Voronoi regions of the other feature (see section 2.4 and figure 2.4). By applying impulses to all contact points we hope to improve the stability of the simulations. By alternating the order in which the contact points are visited, we hope to remove the possibility of spinning from the system.

The price we pay for this (possible) improvement is higher computational cost. The simulation can become anywhere between 2 and 8 times slower, depending on the complexity of the contact. More collision have to be checked

for and more impulses have to be computed and applied.

3.11 Experiments and Results

In this section we present the results of experiments performed using the rigid body simulator described in this chapter. We performed two types of experiments.

First of all we wanted to test whether the improved contact handling method as treated in section 3.10 really does improve the stability of the simulations. We performed experiments in which we measured the stability of a single body and of a stack of bodies (a wall).

Secondly we performed experiments which demonstrate the capabilities of the simulator, e.g. what kind of physical properties can be assigned to bodies, the limitations of the simulator, what kind of systems can be simulated, and simulations which simply 'show off' the capabilities of the simulator. Frames from a number of these simulations have been added to this thesis in the form of an appendix. Full animations are supplied on the CDROM which accompanies this thesis.

The results obtained by evolving virtual creatures could be seen as a third type of experiment performed using the simulator. During the evolution experiments many (more than a million) short (10 to 20 seconds) simulations were run. Animations of some of the creatures which were evolved are also present on the CDROM.

Since this thesis is not so much about physics itself, no attempts were made to validate the model we use (i.e. compare the results of the simulations to the real world). The goal of the work described was to create a sufficiently realistic (compared to our world) and complex artificial world in which virtual creatures could be evolved. To validate the model through comparative experiments (real world vs simulation) would have increased amount of work significantly.

3.11.1 Improving contact handling

To verify empirically that the improved contact handling method really does improve contact handling, we performed a number of experiments in which we compared the performance of the standard way of contact handling (only apply impulses to closest points) to that of the method described in section 3.10. Two situations were examined. In one simulation a cube comes to rest after a short fall on a surface. We measured the stability of the cube as it rests on the surface. In the other situation we simulated a brick wall made out of 34 individual bodies and measured the stability of all bodies. Both situations are shown in figure 3.10.

With the term 'stability' we mean to describe how stable the bodies are during the simulation. During the two simulation described in this section the bodies should not move at all. In an impulse based simulator they *do* move

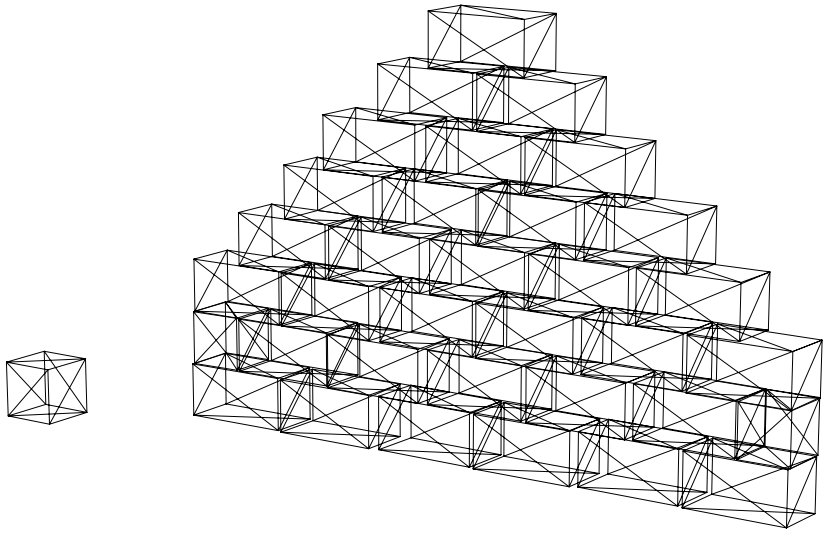

Figure 3.10: The cube and the brick wall used for the experiments in section 3.11.1

however, and the more they move the less stable we call the simulation. We measure the stability of the simulations in two ways.

In the case of the single cube, we measure the *offset* (both linear and angular) of the body to some fixed reference point in the past (i.e. when the body has settled) during the simulation. These linear and angular offsets can be plotted over time and give us some insight into what is going in the simulation.

The other way is by periodically (i.e. 10 times per second) measuring the offset (once again both linear and angular) of the bodies relative to the *previous* measurement. Adding the absolute values of these offsets results in two values (linear and angular 'stability') which give us a way to measure the stability of the simulation. This method was used to measure the stability of the brick wall.

Single Body

In this experiment a single rigid body (the cube shown in figure 3.10) is dropped on a surface floor from a small distance. After the body has settled (i.e. after 5 seconds) we measure the rotation of the body about the y-axis (which is parallel to the vector of gravity) and the magnitude of translation of the body for 60 seconds of real time (it takes the simulator less than 1 second to simulate those 60 seconds in all cases). The magnitude of the translation is measured relative to the position of the body when the measurements start.

In total we performed six measurements. The collision epsilon ϵ was varied from 10% of the size of the body to 0.1% of the size of the body in three steps (10%, 1%, 0.1%). The improved contact handling was either turned on or off.

As can be seen in figure 3.11a, when standard contact handling is used, the cube clearly spins for the two largest collision epsilons. At the smallest collision

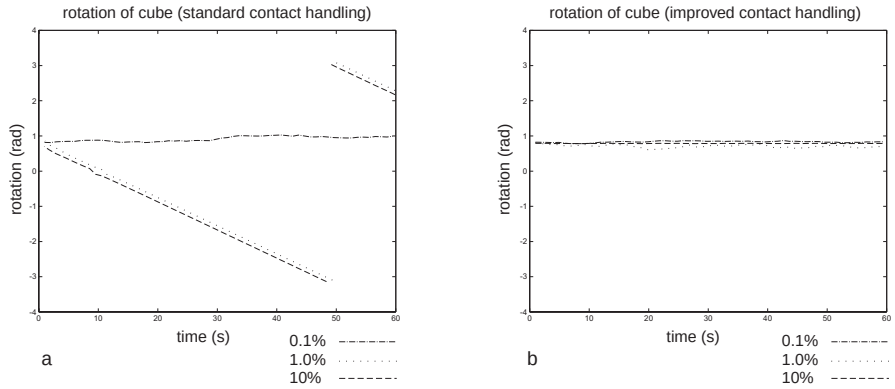

Figure 3.11: a) rotation about the y-axis of the cube for three different collision epsilons (standard contact handling) b) rotation about the y-axis of the cube for three different collision epsilons (improved contact handling)

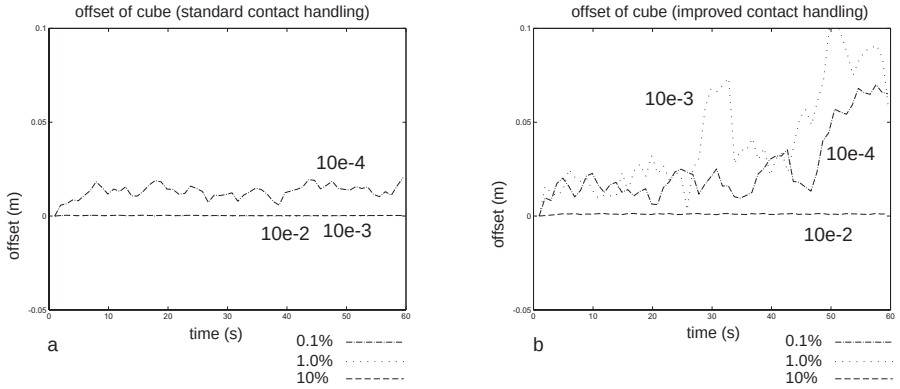

Figure 3.12: a) offset of the cube relative to a reference point for three different collision epsilons (standard contact handling) b) offset of the cube relative to a reference point for three different collision epsilons (improved contact handling)

Table 3.4: Results of the brick wall stability experiment

ϵ	improved c.h.	time required	rotation sum	offset sum	nb collisions
10.0%	off	4.94s	19.6	23.0	54330
1.0%	off	17.0s	3.70	5.61	153506
0.1%	off	failed	failed	failed	failed
10.0%	on	27.0s	0.337	0.543	917634
1.0%	on	27.1s	1.60	2.10	788654
0.1%	on	70.6	0.226	0.464	1688037

epsilon the cube does not spin. Figure 3.11b tells us that the body does never spins when the improved contact handling method is used. Note that there is some kind of 'brownian motion' in the amount of rotation about the y-axis in all cases where the body does not spin.

In figure 3.12a the offset of the cube relative to it's initial position is shown. We see that for the standard contact handling method, the collision epsilons of 10% and 1% cause the body to move very little. The *smallest* collision epsilon causes a lot of instability. In the case where the improved contact handling method is used, all collision epsilons cause a relatively large amount of translation.

Stack of Bodies

A stack of bodies is hard to simulate using an impulse based approach. Impulses have to be propagated up and down the stack constantly, which can cause the simulator to slow down to a crawl. We used this kind of simulation as an experiment anyway to test the improved contact handling method.

The stability of the wall was computed by measuring the (rotational and translational) offset of the bodies in the wall periodically and summing these values. The simulation was run for 5 seconds after the bricks had settled.

As can be seen in table 3.4 the improved contact handling method does really improve the stability of the simulations, but at a considerable computational price. We were unable to simulate the brick wall with the collision epsilon ϵ set to 0.1% and standard contact handling. The simulation would fail because of a problem outlined in the discussion. Note the dramatically increased number of collisions required when the improved contact handling method is enabled.

The effect of the improved contact handling method is best demonstrated by looking at the animation called *bwstab.fmt* on the CDROM (see appendix A) which shows the same scene simulated with and without improved contact handling.

3.11.2 Demonstration simulations

All demonstration simulations were made with the collision epsilon set to 10% of the size of the body and improved contact handling, unless explicitly stated otherwise.

Sliding cubes (friction)

The sliding cubes example demonstrates the effect of friction. 7 cubes with different friction parameters are put on sloped surfaces (with the same friction parameters as the cubes which rest on them). Because 6 of the 7 cubes can not generate enough friction they slide down the surface and will crash into the horizontal surface. From the animation one can easily see what the effect of friction in the simulation is. The friction parameter of the cube which does not move was computed specifically for the slope of the surface. The cube can generate just enough friction to stay put. A slight increment of the slope of the surface would cause it to start sliding down as well.

Bouncing balls (restitution)

The bouncing balls simulation shows the effect of the restitution parameter. The restitution parameter (range $[0, 1]$) specifies how much of the kinetic energy is lost during a collision. A value of 1 results in full restitution; no energy is lost during a collision. A value of 0 results in total dissipation of all kinetic energy during a collision; the colliding points will come to relative halt. The bouncing balls experiment shows 7 balls made of different 'materials', with the restitution parameter varying from 0 to 1 in 6 steps. It clearly demonstrates the effect of the restitution parameter. No friction is present in the simulation which leads to an interesting artifact. Because the models used are not perfect spheres, but polygonal approximations of them, collision will not occur *directly* underneath the center of mass of the sphere. This causes not only the *linear* velocity of the sphere to change, but also the angular velocity. Because the amount of kinetic energy of the sphere certainly does not *increase* during an ordinary (non-micro) collision, this implies that the linear velocity will be less than it would have if the collision would have been *directly* underneath the center of mass of the body (linear velocity is 'exchanged' for angular velocity). This is effect is clearly visible for the sphere with restitution parameter 1.0, which should always bounce up just as high, but doesn't.

Coin toss

The coin toss and simulation demonstrates a type of simulation at which impulse based simulators excel: quickly changing contacts and almost no stacked bodies. This simulation of 18 bodies ran at $\pm$210% real time.

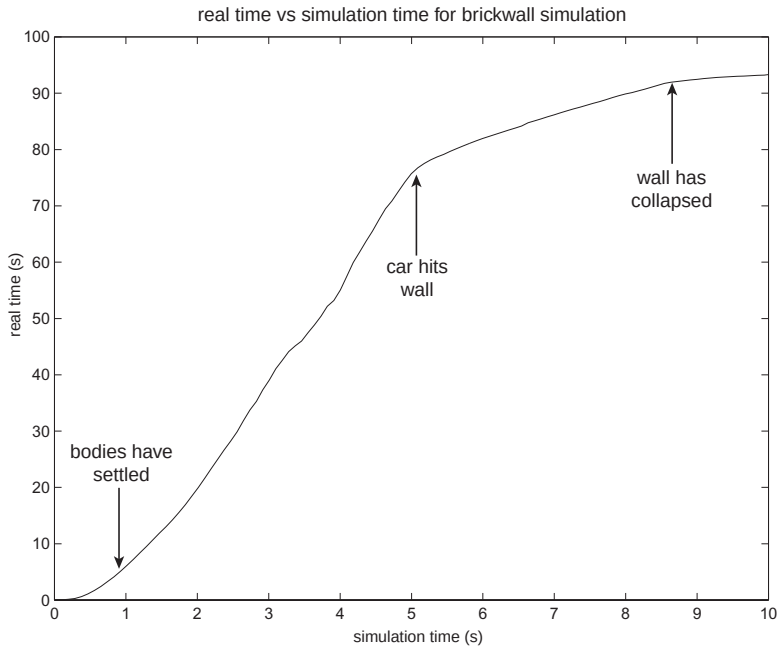

Figure 3.13: Real time vs simulation time for the 'car drives into brick wall' simulation.

Car track

The car simulation demonstrates the use of low and high level controller elements and constrained bodies. The translation joints between the 'chassis' and the wheels contains damped springs. Torques are applied to all four wheels which cause the car the accelerate. The magnitude of the torque is generated by a high level controller which limits the velocity of the wheel to prevent slip. The car is helped a little bit while driving through the looping by lowering the gravity. The car simulation is relatively simple to simulate; it runs at $\pm 110\%$ real time.

Brick wall

Even though brick walls are slow to simulate using an impulse based simulator, they deliver spectacular results when they collapse. As soon as the brick wall has been (partially) smashed the simulation speeds up because most of the stacks of bodies in the simulation cease to exist. Simulations of a car (as described above) driving into a brick wall and a 'demolition ball' smashing either a single or double brick wall have been done.

Figure 3.13 shows a graph of how fast simulation time advances compared to real time for the simulation where the car drives into the brick wall. The simulator requires approximately 94 seconds to simulate 10 seconds. The steeper

the graph, the slower the simulation is runs. It is interesting to see how the events in the simulation can be recognized in the graph. When the simulation is initialized, there is a small gap between all bricks of the wall; during the 0.75 seconds, the bricks settle. This can be recognized in the graph by the fact that the speed of the simulation decreases (the graph gets steeper). During the next 4.25 seconds, the car drives down the sloped surface. This part of the simulations runs very slowly because of the (micro-) impulses which have to be propagated up and down the brick wall all the time to simulate the contact; the simulation runs at $\pm$0.05% of real time. After 5 seconds simulation time, the car hits the brick wall. Immediately, the simulation speeds up to $\pm$25% real time. When the wall has collapsed (after 9 seconds simulation time), the simulation runs at $\pm$110% real time.

Simulations of a 'demolition chain' hitting a (single or double) brick wall have also been performed.

Dominos

A domino track consisting of 150 dominos was simulated. The falling dominos climb up and down stairs and fall from a tower down onto other dominos. The simulation ran at $\pm$25% real time.

Marble

In this simulation a marble is lead through a track consisting of 17 gearwheels, 6 sloped ridges and 6 trapdoors. The trapdoors contain damped springs which try to keep the door at the initial position. When the marble rolls onto a trap door, it's weight causes the trap door to open, allowing it to continue. The gearwheels are driven by a low level controllers which attempts to keep the their angular velocity constant. This simulation ran rather slowly at $\pm$14% real time.

3.12 Discussion and Conclusion

The main property that distinguishes 'traditional' rigid body simulators from impulse based simulators is the fact that *all* contact is modelled using (micro-) impulses. We will discuss here what the weaknesses and strengths of this approach are and how an attempt was made to alleviated several of the weaknesses.

The main strength of a rigid body simulator is that all contact is modelled by a unified approach: impulses. Whether bodies are colliding or resting on each other, there is no change in how the simulator treats the (extended) contacts between them. Also, computing and applying the contact impulses is relatively easy compared to LCP methods [Bar94], and does not lead to stiff integration problems as with penalty methods.

The impulse based approach is better suited for the class of simulations where contacts change quickly and frequently. LCP methods are not efficient when contact 'geometry' (the points where bodies contact each other) changes

rapidly because they have to set up a linear system again for each change [Mir96]. LCP methods use results from the previous invocation, which become useless if the contact geometry changes. Impulse based simulators handle changing contact geometry without any extra effort. Also, LCP methods are not guaranteed to terminate in the case of dynamic friction [Bar94]. Impulse based methods handle friction without any extra effort.

Impulse based simulators are bad at modelling prolonged contact between bodies, and even worse at modelling 'contact chains' (e.g. stacks of bodies). Too simulate high stacks of bodies which are at rest trains of impulses have to be propagated from the floor to the top of the stack and back. This can slow the simulator down to a crawl, as can be seen in the execution times of the brick wall simulation in section 3.11. This is where LCP methods excel. They compute exactly how much force should be applied at each contact point so that all bodies are at rest.

The fact that the simulation of stacks of bodies at rest is inefficient leads to the phenomenon that simulations will run faster as soon as they get interesting. For example, the simulation of the crash of a car into a brick wall runs very slowly until the car actually crashes into the wall and the interesting part of the simulation starts as shown in figure 3.13.

A major efficiency improvement could be shifting bodies which are at rest into a 'sleep state' where the simulator does not simulate them anymore but simply assumes they will stay where they are until some external influence starts to act on them could speed up simulations considerably. However, such a hack could easily lead to consistency problems if bodies are not switched from sleep state to full simulation state at the right moments.

An artifact of the impulse based approach is that the bodies never settle. They always keep bouncing because of the (micro-) impulses they are receiving constantly. Although Mirtich claims that the amount of bouncing can be reduced almost arbitrarily (at higher computational cost) by reducing the collision epsilon ϵ, we found this method difficult to use and unpredictable because of the interdependencies between certain parameters (such as the step size which is made). Lowering the collision epsilon could cause the simulation to become *less* stable. This is caused by the fact that as the distances between the bodies get smaller, the simulator will make a more accurate estimation of the lower bound on the step size which can be made, which results in deeper penetration of the collision envelope, which in turn results in micro impulses (see section 3.4) with a higher boosted coefficient of restitution. A higher boosted coefficient of restitution will lead to a less stable simulation. Also, reducing the collision epsilon leads to relatively large floating point rounds error because the ratio between the scale of the bodies and the collision epsilon gets larger. This would force us to use double precision floating point calculations.

Another significant artifact that can appear in an impulse based simulator is 'spinning'. A body which rests on a flat surface experiences a continuous train of micro-collisions. For instance, a cube would receive micro-impulses at each of it's for lower corners. The impulse based simulation method as described in [Mir96] can cause the order in which the corners receive impulses to settle, which results a systematic error (as explained in section 3.10). We devised an

slightly altered version of the impulse based simulation method which does not cause spinning, and in most situations increases (or at least does not decrease significantly) the stability of bodies in the simulator.

The improved contact handling method (section 3.10) tracks the *contact points* of bodies and applies impulses not only at the closest points between a pair of bodies, but also at all (valid) contact points. Because of the absence of a specific order in which contact points receive impulses (they all receive them at the same time, and the order in which contact points are visited is altered periodically), lowering the possibility of systematic errors (e.g. spinning) occurring. There still is some systematic error in the brick wall simulation, as can be seen by looking closely at the brick wall on the right in the *bwstab* animation on the CDROM.

It seems however that *any* method which sufficiently randomizes the order in which micro-impulses are delivered will prevent systematic errors from occurring. E.g. in the experiment described in section 3.11, the cube did not spin when the collision epsilon was set to 0.1% of the size of the body because the simulation was relatively instable because the collision envelope got penetrated so much.

Also, because bodies receive impulses more evenly distributed over their surface, the stability of certain systems (such as stacks of bodies) is increased significantly, as demonstrated in section 3.11. Although this can also be achieved with the simulator described in [Mir96] by lowering the collision epsilon, this may not be desirable. Lowering the collision epsilon will increase computational cost for all bodies in the simulator, while the altered method only increases computational cost for the bodies involved in complicated contact. We had problems when using small collision epsilons and the standard contact handling method, i.e. we could not run the brick wall simulation with ϵ set to 0.1% of the size of the bodies. Also, as a rule of thumb, parameter tuning should be eschewed, since it quickly turns into a black art. The collision epsilon should serve as a tool for the user to determine the precision of this simulation, not as a method for getting simulations to run or not.

A third problem our simulator exhibits is halting in certain situations; this is what happened in the failed brick wall experiment where the collision epsilon was set to 0.1% of the size of the bodies.This problem is due to floating point round off errors. Due to round off errors the collision detection algorithm fails to detect a collision between two bodies which are extremely close (the distance between them is at the limit of floating point precision). The next integration step will result in penetration of the extremely close bodies. The simulator will try smaller and smaller integration steps until floating point precision is reached for the step size and it is unable to continue. Such a situation can occur if the previous integration step caused two bodies to come extremely close and their relative velocities to become very low. A solution could be to undo the previous integration step, and redo it with a smaller step size, preventing the bodies from ever coming extremely close. During our experiments we got the impression that the improved contact handling algorithm causes the problem to occur much less often, but no exact measurements were made to (dis-)prove this. Also, lowering the collision epsilon caused the problem to occur more

often.

For simulation of articulated bodies, we selected the Featherstone algorithm for our simulator because it is faster than multiplier method. Even though multiplier methods can also have $O(n)$ performance, [Bar96] states that reduced coordinate method are fast than multiplier methods if the degrees of freedom of the system is small compared to the maximum number of coordinates. In retrospect using a multiplier method might have allowed us to experiment with an implicit integration method. Implicit integration methods increase stability in case of stiff integration problems, but require the computation of a Jacobian matrix, which is very complicated with the Featherstone algorithm.

The rigid body simulator is able to simulate a wide range of physical systems, as shown in the demonstration simulations (section 3.11.2). Many systems of moderate size (5 to 25 bodies) can be simulated in (super-) real time, as long as no high stacks of bodies are formed. Such stacks lead to significantly lower performance.

As a final remark, we would like to note that even though the simulator is quite efficient and reliable in it's current state, it will likely benefit from a total rewrite in terms accuracy, performance, flexibility, usability and simulation speed. The collision detection module inside the simulator was rewritten three times, becoming more stable, flexible and faster each incarnation. We think the same would be true for the simulator.

Chapter 4

Evolving Virtual Creatures

4.1 Introduction

In this chapter we describe how we attempted to evolve 'virtual creatures' using the simulator, a genetic description of the creatures, a method to convert a creature's genotype to it's phenotype, genetic operators (mutation, recombination), a parallel creature evaluation method and a genetic algorithm.

A genetic algorithm is a computer program which attempts to optimize some function by using the principle of survival of the fittest and combining and altering genetic material describing *individuals*. A genetic algorithm operates on a *population* of individuals; each individual is a possible optimal solution to the function we want to optimize. The genetic algorithm tests or evaluates how well each individual performs and assigns it a *fitness* value (the higher the fitness, the better the individual performed). After it has evaluated the fitness of every individual the genetic algorithm selects (according to some *sampling mechanism*) the best individuals from the population and uses the genetic description of these individuals to create the genetic description of the next *generation*. This should cause the average fitness of the population to increase. For an introduction to genetic algorithms, see for instance [Hol75] or [Mic96].

In our case, the individuals are virtual creatures and the function we want to optimize is some task we want the creatures to perform. Virtual creatures are, in this context, robots made from rigid bodies connected by joints. They are simulated using our rigid body simulator described in chapter 3. The creatures can sense certain properties of the world, such as the location of certain objects and the amount of rotation at their joints. The creatures have controllers, simple neural networks, which control the joint forces exerted at their joints. The neural networks do not learn (the weights are not adjusted, no new connections are formed) while the creatures are being simulated, only through evolution. That is, creatures with relatively good morphology and controllers are more likely to have offspring in the next generation. Morphology and controllers will evolve alongside each other.

The creatures had to learn certain tasks like hitting a ball, walking and

jumping through evolution. For every generation, every creature is built inside the simulator, simulated for a number of seconds and allowed to move itself by controlling the joint forces via it's neural network. While the creature is being simulated, a fitness evaluation module inspects the creature's performance. At the end of the simulation the evaluation module reports a fitness value ($\geqq 0$) for the creature. The higher this value, the more likely it is that the genetic algorithm will select the creature to have offspring.

We tried to stick as closely possible to the description of the creature morphology and controllers as described in [Sim94a], in order to reproduce Sims results. We have implemented the description of the genotype of the creatures quite differently from Sims, but this should (theoretically) not have any effect on the results.

4.2 Morphology

The virtual creatures are built from body parts connected by joints, thus forming articulated bodies. These bodies are be simulated using the Featherstone algorithm described in section 3.6. The genotype of the creatures gives instructions on how to build or grow[1] the creatures. We want the genotype to have as much control over the shape of the creatures as possible within the constraint that the Featherstone algorithm is used to simulate them. This means that the genotype will describe, among others, the shape and size of bodies, topology (how the bodies are connected), the degrees of freedom of joints, the position and orientation of joints and the joint force controllers.

To save space in the genotype (which has a variable size) and to promote the use of the same genetic information to build to multiple (nearly) identical body parts (such as a left- and a right arm), we split the information describing the creatures into two parts: the *nodes*, which describe the body parts, and the *connections*, which describe how the nodes are connected. Together the nodes and connections form a graph (which may contain cycles and / or unconnected nodes). To build a creature from the graph, the graph is traversed and body parts are added to the creature according to the information in the nodes and connections.

Nodes contain the following information:

1. The base shape of the body part. A number of shapes (e.g. cube, sphere, cone) can be available as body parts during an evolution. A value determines which shape is used. During most evolutions this option was disabled however, and only one shape was used.

2. The size of the body part. This is a 3D vector specifying how much to scale the base shape (which has a 'unit' size) in the $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ direction. This option too was disabled for some evolutions.

[1] We will use the term 'grow' in this context to refer to the construction of a creature inside the simulator; after a creature has been constructed, it will not grow or develop (e.g. become larger, stronger or smarter) in any way .

3. The degrees of freedom of the joint connecting this body part to it's 'parent body part'. A joint has either 1, 2 or 3 d.o.f. Bodies are arranged in a parent-child relation. Every body has a parent, except for the root body, and every body has one or more child bodies, except for the leaf bodies.

4. The joint axis for each d.o.f. (a maximum of three 3D vectors). Creatures can choose arbitrary joint axes for each d.o.f., as long as they are sufficiently independent. The angle between each of the joint axes must be larger than 30 degrees (a rather arbitrary limit, which works well in practice); the Featherstone algorithm can not handle two consecutive joints with (nearly) parallel axes.

5. The limits on the joints. Joint limits restrict the amount of rotation of a joint to a certain range (like all human joints). The joint limits are described by a maximum of three pairs of two scalars. Each pair of scalars specifies the minimum and maximum rotation angle for each d.o.f. Joint limits are enforced by applying micro impulses at the joints, as described in section 3.6.8.

6. A number of neurons. The neurons described in nodes are part of the local controller mechanism. They control the forces exerted at the joints.

7. Neural inputs, one for each degree of freedom. A neural input connects a neural output to the input of a joint force effector. The output of the neuron dictates to the amount of force exerted at a joint.

8. A replication limit. Because of the way the connections between body parts are described, a body part can be 'replicated' or built multiple times while the creature is being constructed. This value limits the number of times a node can be used to replicate a body part. This limit allows chains of a limited length to be constructed from identical genetic material, and also prevents infinite recursion while traversing the graph.

Inside the genotype nodes are stored in an array, so every body part has a unique index.

Which body parts are connected to which body parts by joints is described by connections. A connection states which nodes (the parent and the child) the connection connects by index, what the offset and orientation of the child relative to the parent is, whether the child is reflected, and how much the child is scaled relative to the parent. By allowing for reflection identical but mirrored linkages (like human arms or legs) can be constructed easily. Scaling simplifies the construction of linkages of bodies which get smaller each link (like human fingers).

Figure 4.1 shows an example of two graphs (designed by hand) and the creatures which could be grown from them, assuming the nodes and connections contain the right information. Not all information to build the creatures (such as size and shape of the body parts) is shown in the graphs. The nodes in the graph are labeled with names according to function they have in the final anatomy ('body segment', 'leg segment') of the creatures, but the genotype has no such notion.

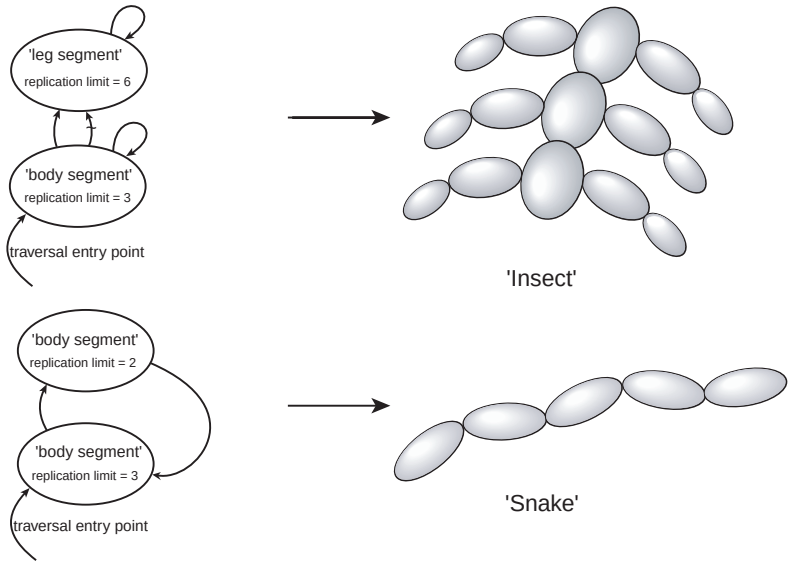

Figure 4.1: Two graphs (designed by hand) and the creatures which could be grown from them.

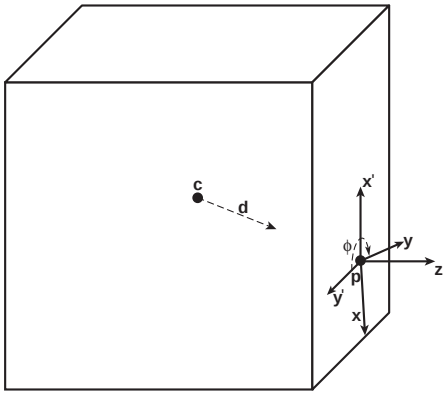

Figure 4.2: Position and orientation of a child body relative to a parent body. The joint of the child body is fixed inside the **xyz** frame.

The traversal of the graph starts at the entry point. For every node encountered during the traversal a body part and local controller neurons are created, using the information in the connection and the node. Every time a node is used to create a body part, the replication limit of that node is decreased. The traversal continues (breadth first) until a node with no outgoing connections is found, or a node with a replication limit of zero is encountered.

The information in the connection about the relative position and orientation of the child to the parent is used as demonstrated in figure 4.2. First the offset direction vector $\mathbf{d}$ is used to compute *where* the child body should be connected relative to the center of mass $\mathbf{c}$ of the parent body. Then the normal of the parent body surface at the link point $\mathbf{p}$ is computed and a frame is constructed (see section 3.2.6). The $\mathbf{z}$-axis of the frame will be parallel to the normal of the surface, and the $\mathbf{x}'$- and $\mathbf{y}'$-axes will have rather arbitrary (but *consistent*) directions. That is why we will allow the genotype to specify a rotation ϕ of the frame about the z-axis. This rotation takes the $\mathbf{x}'$- and $\mathbf{y}'$-axes to final the $\mathbf{x}$- and $\mathbf{y}$-axes. The joint axes (not shown in the figure) for each d.o.f. are fixed inside this rotated frame. The child body is attached to the final joint, with an appropriate offset such that the surface of the parent and child body just touch each other. Reflection and scaling influence the process described above in obvious ways. Scaling and reflection are passed on during traversal of the graph so their effects accumulate.

Note that some information in the genotype may not be used to construct the actual creature. E.g. the root node of the hierarchy contains information on how to connect it to it's parent body, even though it does not have a parent. Similarly every node contains information about joints (e.g. the joint limits) for the *maximum* of three joints, even though they may use only one. Thus the genotype contains some 'junk information[2]' which mutates along with the rest of the genotype, and may or may not be activated at some later time by a mutation.

4.3 Controllers

The controllers of the creatures control the joint forces which are exerted at the joints. Every d.o.f. of a joint corresponds to a link in a Featherstone tree of linkages. The magnitude of the force exerted at such a joint is represented by the term Q_i (see e.g. equation 3.56). By controlling the joint forces Q_1 to Q_n (where n is the total number of d.o.f. inside the creature's joint mechanism) creatures can move their body parts in all directions their morphology allows. The task of the controllers is to supply the right forces for the task at hand.

The controllers are evolved alongside the morphology of the creatures. This means that the genotype contains information about how to build the controller networks. This information is subject to mutation and crossover when the genotypes for the a new generation are created, just like the information

[2]Though not entirely similar, biological genotypes (DNA) also contain a lot of junk information, leftovers from previous stages in the evolution of a species. Sometimes these junk genes are activated through a random mutation.

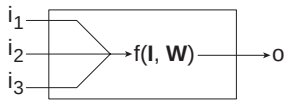

Figure 4.3: A neuron as used in the neural network controllers of the creatures.

describing the creatures' morphology. The controllers will thus be evolved by the same genetic algorithm.

The controllers are simple neural networks. Every neuron (shown in figure 4.3) in the network has three inputs (the number of *used* inputs depends on the function the neuron performs) and one output. The sources which can serve as inputs are

- neural outputs
- world sensors (e.g. the position of an object in the world)
- joint angle sensors
- touch sensors

The neurons take their inputs $\mathbf{I}$, weights $\mathbf{W}$, internal state $\mathbf{S}$ and perform some function $o = f(\mathbf{I}, \mathbf{W}, \mathbf{S})$ on them and output the result o The function f scales each of the inputs i_j with the respective element w_j from the scaling vector $\mathbf{W}$. Then it applies one of the neuron functions on the scaled input vector. The internal state of the neuron is a vector which can be used to store results from previous computations, allowing for neurons which perform some kind of filter operation on the input (e.g. a differentiator). The scaling vector $\mathbf{W}$ is subject to the genetic operators, as are the sources used for the input vector $\mathbf{I}$. A wide variety of neuron functions are available: add, subtract, multiple, divide, less than, greater than, minimum, maximum, absolute value, several types of oscillators, sine, cosine, sigmoid, delay, integrate, differentiate, logarithm, exponential, invert and constant value. These functions use anywhere from 0 to 3 of the inputs. The neuron function is selected by an index which is subject to genetic operators.

The neural network is divided into two parts: the global network and several local networks. The global network could be considered to be the central brain of the creature. A local network is replicated at every body in the creature from the descriptions in the nodes. Every node contains a description of a local network. The description of the global neural network is stored separately.

When the creature is built from it's genotype, this network is created without connecting the neural inputs and outputs. Once the entire creature has been built the neurons are connected. This is not possible in advance because the neurons refer to each other using an indirect addressing scheme. Neurons can specify for instance that 'input 2 should be connected to the output of the fourth neuron of the second child of this node'. This indirect addressing scheme is used because relationships between bodies can change (because connections can change). Neurons can not be connected in advance because when

a node disappears from the graph (e.g. through mutation), some neurons might suddenly be left unconnected.

There are seven types of input a neuron can use:

- The output of 'local' neurons. Local neurons are neurons belonging to the same body part of the creature.
- The output of 'parent' neurons. Parent neurons are the neurons belonging to the parent of a body.
- The output of 'child' neurons. Child neurons are the neurons belonging to the child of a body.
- The output of 'global' neurons. Global neurons are part of the central brain of the creature.
- The state of 'local' sensors. Local sensors sense properties of the body the neuron belongs to (joint angles).
- The state of 'world' sensors. World sensors measure properties of the world around the creature, for instance the position of some interesting object.

Note that not all neural outputs and sensors are available to all the neurons. They can only connect directly to local or neighboring neurons, to local or world sensors, or to the global neural network.

A fixed number of times per second, 25 times per second in the evolution experiments presented here, the output values of the neurons are updated. They compute the new output from the current input values and possibly their internal state (e.g. the memory function delays input signals).

4.4 Implementation

This section presents how we implemented the specification of the creatures detailed above. This implementation is different from the one used in [Sim94a] and [Sim94b], who used lisp expressions to describe the creatures.

4.4.1 Representing the genotype

To store and manipulate a genotype as complicated as the one described above in sections 4.2 and 4.3 the 'standard bit string representation' will not be a good choice. The standard bit string representation used in genetic algorithms (see for instance [Mic96]) stores the genotypes of the individuals in the population as a fixed length strings of bits. When an individual is to be evaluated, this bit string has to be translated into some more useful form. The genotypes are mutated by randomly flipping the state of bits, and crossovers are performed by swapping parts of the bit strings.

This approach has a number of disadvantages. First of all, the fixed length of the bit strings does not allow for the encoding of variable size genotypes, unless we reserve *a lot* of space (which will be unused in most genotypes). In any

case we will impose an upper limit on the size of the genotype. Secondly when the bit string is interpreted to convert it to some more useful representation (e.g. when evaluating the individuals fitness), the information in the bit string has to be checked for errors and repaired somehow. The standard genetic operators (mutation, crossover) used for bit strings have no notion of what they are operating on and this leads to invalid genotypes, which either have to be discarded or repaired. Another problem is that hierarchical structures (such as graphs) can not be represented easily by a bit string. And even if a method were devised to store the hierarchy, it would get damaged by the genetic operators.

An improvement over the bit string representation for certain applications is the use of strings of floating point numbers. The mutation operator is changed somewhat (adding for instance a random value from a gaussian distribution to the value, instead of randomly flipping bits), and the crossover operator only exchanges whole floating point numbers between genotypes, without cutting up the bit strings that make up the individual floats. This approach leads to significant improvements over the standard bit string representation [Mic96].

We take this generalization one step further by using evolvable objects. Evolvable objects contain members (e.g. floating point values, integers, or other evolvable objects). An evolvable object 'knows' (through functions associated with it) how to mutate itself, how to initialize itself randomly, how recombine itself with another evolvable object, or even how to 'built' a phenotype from a genotype. They also contain functions to store their content in an array of bits (which can be stored on disc, or transmitted through a network connection) and to retrieve their content from such an array. Evolvable objects of several types can be defined, each representing another part of the genotype (i.e. the whole creature, neurons, nodes or connections). The evolvable object approach to representing the genotype has a number of advantages over the bit string approach:

- Hierarchical structures can be constructed easily within the genotype by embedding evolvable objects inside evolvable objects. In our case, we use arrays of nodes, connections and neurons which are embedded inside other evolvable objects. By pointing to each other the nodes and connections form a graph.

- The distinct parts of the genotype (i.e. neurons, nodes, connections) are isolated from each other. The evolvable objects are implemented as separate C++ classes. This allows us to add or change a feature of a specific evolvable object easily.

- The functions used to mutate or recombine the evolvable objects are written specifically for that evolvable object. The functions immediately repair invalid configurations (which is usually much easier than repairing them afterwards). Also mutations can be much more effective (as with using floating point values instead of bit strings) because we can mutate each member of an evolvable object in a specific way.

- We have control over the mutation frequency of every member of every evolvable object. For instance, we could stop morphology mutations and allow only the controllers to mutate.

It also has some disadvantages:

- More work is required to write the specific functions for every evolvable object. Although the functions give us more power, they can be tedious to write. E.g. if the bit string representation were used, we could use one algorithm for performing all mutations, instead of one algorithm for every type of evolvable object.
- Although we have more control over the mutation frequencies of the distinct part of the genotype, this also gives on us the responsibility to set those mutation frequencies to their (near-) optimal values. It is not clear how this should be done.
- It is questionable whether the schema theorem and the building block hypothesis (see e.g. [Mic96]) are still valid for evolvable objects.

As stated above each type evolvable object has a number of *functions* (implemented as virtual class member functions in C++) associated with it that can perform a certain task like mutating the object or storing it as a bit string. The functions take as argument the evolvable object and several other parameters which may vary per function. These functions are:

- Random initialization. To form an initial population of random individuals some way must exist to create random genotypes. This is done by the random initialization function.
- Mutation. The mutation function of an evolvable object mutates every member with a probability specified by the function argument and calls the mutation function of every evolvable object embedded within it. The probability with which a member will be mutated is specified by a function argument which lists the probability of mutation of each member, or a whole class of values (e.g. all floats).
- Recombination. The recombination function recombines two evolvable objects to form one new evolvable objects. The recombination operation can be crossover, but other functions which combine two genotypes to form one new genotype (e.g. crafting) are also allowed.
- Grow. With a complex genotype it is not possible to evaluate the fitness of an individual directly, just as it is not possible to tell the exact properties of a biological individual from just it's DNA. The grow function interprets the genotype and creates a 'runtime' individual which can be evaluated. In our case, this comes down to building the creature (inside the simulator) and creating the neural network[3].

[3]this process is actually done in two stages called 'grow' and 'develop' as detailed in section 4.4.2

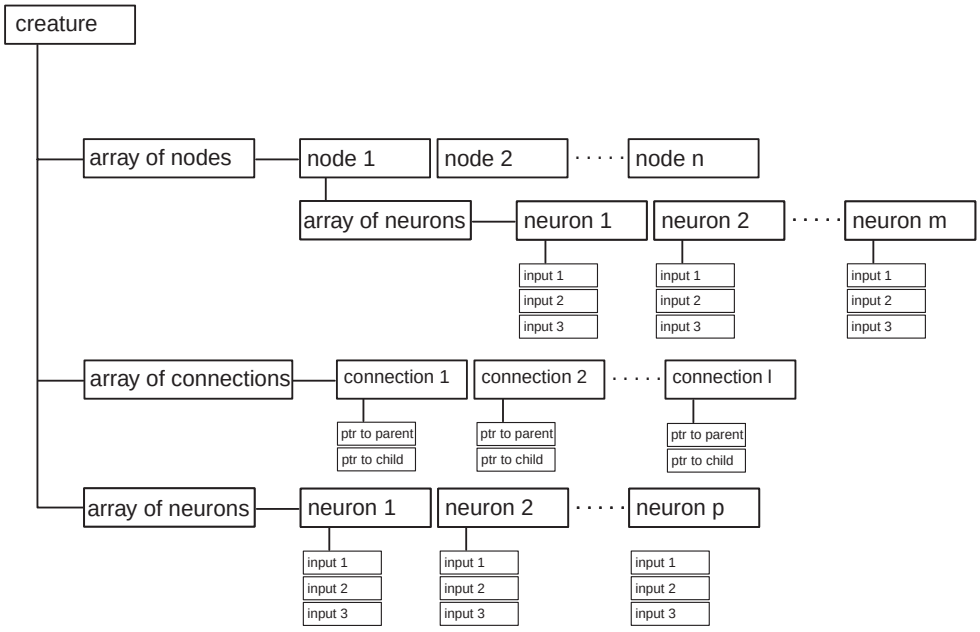

Figure 4.4: The evolvable object hierarchy used to represent our creatures' genotype.

- Store. Allows the evolvable object to be stored in an array of bytes. At runtime, evolvable object can be complicated structures in memory which can not be written to disk or transmitted via a network easily. Writing two *(de-)serialization* functions (store and retrieve) for every evolvable object makes it possible to store and retrieve (nested) evolvable objects to and from a flat array of bytes.
- Retrieve. Performs the opposite action of store.

Figure 4.4 shows the evolvable object hierarchy we use to represent the genotype of our creatures. At the top of the hierarchy stands an object called *creature*, which contains the entire genotype. The creatures contains three arrays of evolvable objects: the nodes (body parts), connections and the global neurons. The nodes contain local neurons. Neurons contain evolvable objects which specifies the source of their inputs.

4.4.2 Construction of the phenotype from the genotype

In our case, the phenotype is created from the genotype in a two-stage process. These two stages are required to wire all neural connections correctly. During the first stage (called 'growing') the runtime representation (the phenotype) of the creature is constructed by the evolvable objects through the 'grow' function. This involves traversing the nodes graph following all connections,

replicating nodes and neurons as required. During the second stage ('developing') the creatures' description is entered into the simulator. For every body part a body is added to the simulator and connected to it's parent body. Another important operation during this stage is connecting the neural inputs and outputs. This operation can not be completed until all body parts have been constructed and thus can only happen during the second stage. As detailed in section 4.3, neurons can connect their inputs to a number of sources, including (other) neurons' outputs and sensors. After the second stage has been completed, the simulator can initialize itself and simulate the creature for a number of seconds.

4.5 Creature Evaluation and Tasks

While the creature is being simulated an evaluation module inspects the performance of the creature. This module is implemented as a timer inside the simulator (see section 3.7). Periodically it gets called and is allowed to do some measurements on the performance of the creature.

The evaluation modules used for the tasks used in our work usually inspect the center of mass or velocity of bodies or distance of some body relative to another body. This information is requested from the simulator or the V-Clip algorithm.

The simulator is entirely deterministic. If the initial conditions and the simulation parameters are identical, the outcome of the simulation will be identical. To prevent the creatures from learning very specific behaviors which are only possible with very specific initial conditions, the simulation parameters such as the collision epsilon, the restitution and friction parameters and gravity are varied slightly on each evaluation. If they were identical all the time the creatures could learn to exploit some event which is very unlikely to occur in a slightly different simulation. By varying the properties of the simulation we make the simulation less predictable, thus preventing the creatures from exploiting highly unstable situations.

In some evolution we experimented with evaluating the same creature multiple (i.e. 3) times, each time with slightly different parameters, and letting only the lowest fitness count. Although this method maybe less efficient in terms of number of evaluations, it filters out the creatures which are specialized for a certain set of simulation parameters quickly.

For tasks where gravity is enabled, the creatures are first simulated for a number of seconds *without* allowing them to exert forces at their joints. This causes them to drop onto the floor and prevents the creatures from cheating by using tactics such as falling over to achieve higher fitness.

We attempted to evolve creatures for performing the following tasks:

- Hitting a ball. The fitness of the creatures was defined as the z coordinate of the center of mass of a ball, which was placed at z = 0 at the start of the simulation, divided by the number of body parts of the creature. We divide by the number of body parts to encourage the creatures to use as little body parts as possible. Gravity is disabled during the simulation.

The creature's root node can not move. Each generation the ball is placed slightly further away from the creature (in the direction of the positive x-axis), so they have to adapt to still be able to hit the ball.

- Hitting a ball which is not always at the same location. The same fitness criterion for the tasks described above is used, except the position of the ball is varied slightly by adding a random vector to it's initial position. The creatures can sense the position of the ball through three world sensors.
- Jumping straight up. For this task, the fitness of the creatures was defined as the maximum distance between the floor and *lowest* body of the creature during the simulation divided by the distance of the creature's center of mass from the origin at the end of the simulation. This should cause the creatures to learn to jump straight up and down.
- Jumping away from the origin. We defined the fitness of the creatures as the maximum distance between the floor and *lowest* body of the creature during the simulation *multiplied* by the distance of the creature's center of mass from the origin at the end of the simulation. This definition should cause the creature to both jump up and to a certain direction at the same time.
- Locomotion. To learn the creatures to move in a certain direction (i.e. along the x-axis), the fitness of the creatures was defined as the distance of center mass along that x-axis divided by the distance of center mass from the z-axis. Creatures whose center of mass come above a certain threshold were disqualified. This measure was used to prevent creatures from jumping.

4.6 Parallel Evaluation

4.6.1 Introduction

The evaluation of the fitness of the creatures is very expensive in terms of CPU time because of the rigid body simulation. Though most creatures can be simulated faster than real time, the evolution of a typical population of 300 creatures would still take too much time. Some 100 generations are required to get interesting results, so, assuming an optimistic 5 seconds simulation time per creature (for e.g. 15 seconds real time), this requires $100 \times 300 \times 5$ seconds $= 150000$ seconds ≈ 42 hours to evolve a population.

By simulating the creatures in parallel on a number of processors, a linear speedup can be achieved can easily. The genotypes of the creatures are distributed across the available processors. Every process (running on a separate processor) evaluates the creatures assigned to it and returns the fitness of the creature to the 'master' process. The master process records the fitness of the creatures and performs a simple form of dynamic load balancing by sending genotypes of creatures to processors only when they finished their previous evaluation.

4.6.2 Implementation

This simple scheme to run the evaluations in parallel was implemented as follows. The master process is started by the user. It reads a configuration file which contains (among others) the number of creatures in the population, the number of generations to be evolved and the task the creatures must learn. It creates a TCP (Internet) listen port. This port will be used to accept connections from slave processes. The master process then initializes a random initial population (or loads one from disk) and enters the main loop.

The main loop of the master process consists of polling for new connections on the TCP port, polling the active network connections to slave processes, sending genotypes to slave processes which have nothing to do, and, when all creatures of a generation have been evaluated, recombining and mutating genetic material to create the next generation. The main loop of the master process runs until the required number of generations has been evolved, or until it is interrupted by the user.

As long as no slave processes connect to the master process, no creatures will be evaluated, so no progress will be made. The user has to start a number of slave processes. This is done by starting a proxy (for reasons explained shortly) on the slave processors. The proxy polls whether the master process is running (by trying to connect to the master TCP port). If the master process is running, it spawns a slave process which will connect to the proxy. The proxy connects to the master process and acts as an intermediary between the master and slave process. The proxy will create as much slave processes on a computer as there are processors. All these slave processes connect to the master process (via the proxy).

The proxy is used for two reasons. First of all, the simulator is not entirely stable; sometimes it may crash or get caught in a infinite (or *very* long loop). For a typical evolution 30000 simulations have to be done, so if there is a problem in the simulator it will probably occur. The proxy will detect this problem (because of the absence of communication between the slave process and the proxy), terminate the slave process and spawn a new slave process. The proxy will report a negative fitness for the creature the old slave was evaluating to the master process (indicating a 'faulty' creature). The master process will not use the genetic material of faulty creatures for the next generation. The second reason for using a proxy is that the proxy automatically spawns the right number of slave processes. It does this by retrieving the number of processors present in the machine from the OS and starting a slave process for each processor.

When a slave process receives a genotype of a creature, it initializes the simulator and builds the creature, simulates it and sends the fitness of the creature to the master process. The slave process knows what task the creature should perform (because the master process sends that information with the genotype), so it can use the right fitness evaluation module to measure the competence of the creature. While simulating, the client periodically sends 'pings' to the proxy to let it know it is still alive and not caught in an infinite loop.

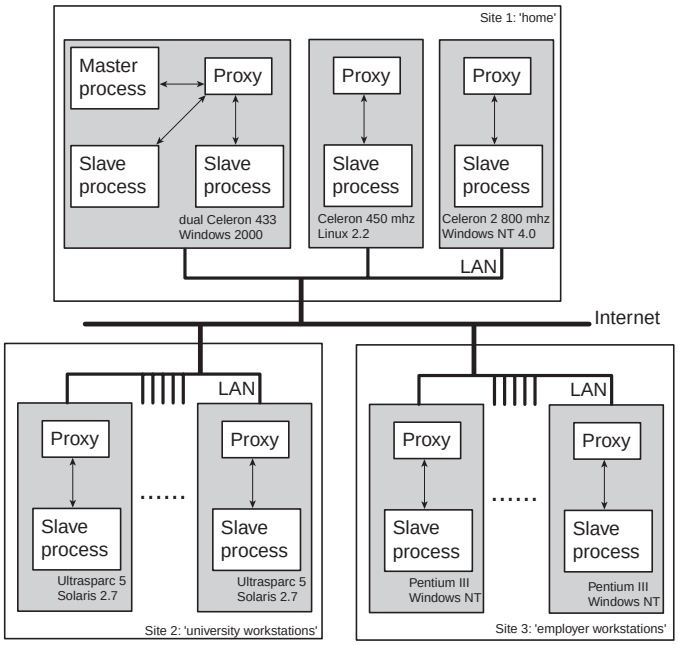

Figure 4.5: Example of setup for parallel evaluation of the fitness of the virtual creatures. Machine types and operating systems are shown to give an idea of the diverse platforms the creature evaluation software can run on.

Example

A typical setup is shown in figure 4.5. The computers which will be used to evaluate the creatures' fitness are distributed across three physical locations. The sites are connected with each other via the Internet. The computers at one site are connected to each other via a local area network (LAN). The master process runs on a computer at site '1'. Because the master process does not require much CPU time, and the computer it runs on has two processors, two slave processes are run also on the same computer. The master process is connected to all other slave processes via the LAN or the Internet. Site '2' is a cluster of workstations at the university. This distributed computing environment allows the processing power to be scaled up easily, as long as enough communication bandwidth is available (to transmit the genotypes from the master to the slave processes).

4.7 Genetic Algorithm

Besides the representation of the genotype, a method to convert the genotype into a a phenotype, an evaluation module and a parallel evaluation method, two more ingredients are required to evolve virtual creatures: genetic opera-

tors and a genetic algorithm.

4.7.1 Genetic operators

Genetic operators create new genetic material out of existing genetic material. Usually two types of genetic operators are used: mutation and recombination. Mutation leaves most of the genetic material intact, but changes (mutates) several values of the genotype. Mutation is traditionally performed by flipping the state of bits in the bit string. Recombination combines the genotype of two individuals to create a new genotype. The most common recombination operator is crossover. In the case of a bit string, the two bit strings are cut at one or more locations and pieces of genetic material are exchanged between the two genotypes.

In our case, mutation is performed by calling the mutation function associated with the evolvable object you want to mutate. This causes the evolvable object to mutate it's members (which can include other evolvable objects) with certain probabilities. Possible mutations include flipping a bit (for integers), setting an index to a new random index, adding a random value from a gaussian distribution to a real value and adding and removing nodes, connections and neurons. The mutation function of each evolvable object checks whether it is still valid after mutation.

Our crossover operator lays the nodes of the creatures along side each other and swaps a number of consecutive nodes between the two creatures. All connections internal to the nodes of the creatures which are exchanged are also swapped. This crossover operator results in the swapping of entire nodes (descriptions of body parts, local neuron networks and internal connections). Note that this operation happens at the top level of our evolvable object hierarchy (the evolvable object called 'creature'). All other evolvable objects have crossover function associated with them that do nothing.

We use a second type of recombination method called grafting [Sim94a]. We cut a number of nodes loose from a genotype (including all internal connections and neurons) and paste it to another genotype. This can result in grafting for instance a limb of one creature to another. Grafting, like crossover, happens at the top level of our evolvable object hierarchy.

4.7.2 Genetic Algorithm

We use a relatively simple genetic algorithm to drive the evolution. First of all, the best performing genotype of a generation is always copied to the next generation (elitist model). The rest of the genotypes are created by mutating or recombining genotypes of selected individuals. 30% of the new genotypes are generated by mutating the genotype of some selected creature, 40% by crossing over the genetic material of two selected creatures, and 40% by grafting. Individuals are selected proportionally to their fitness. E.g. if an individual has a fitness twice as high as another individual, it has twice as high a chance of being selected as a source of genetic material for the next generation.

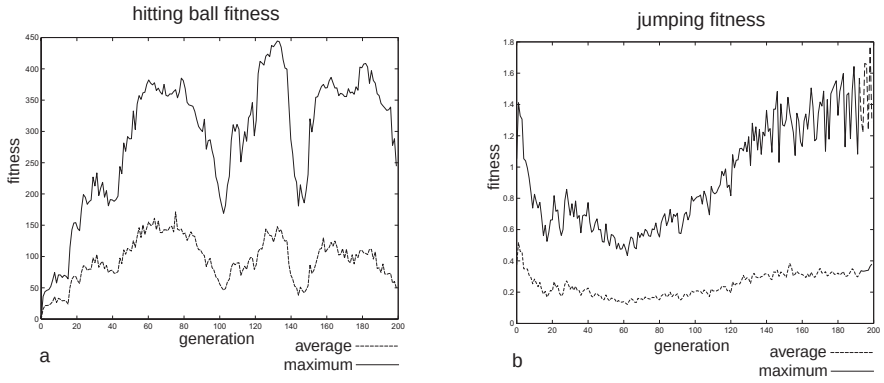

Figure 4.6: a) Fitness of the creatures during a *hit ball* evolution. b) Fitness of the creatures during a *jump* evolution.

We can reduce the selective pressure by adding multiple s of the average fitness to every individuals fitness. We slowly reduce s each generation. I.e. for generation 1 s would be 0.5 (adding half the average fitness to every individuals fitness), and for generation n, s would be 0.0, causing the creatures to be selected according to their true fitness value. A low selective pressure helps to prevent premature convergence to a local optimum in the presence of so called super individuals which perform much better than most other individuals (see e.g. [Mic96]).

4.8 Results

We ran the genetic algorithm to evolve creatures to perform the tasks described in section 4.5. A typical run (100-200 generations, population size 300) required anywhere between 2 and 6 hours running 8 to 12 processors (depending on what machine were available). Animation of the creatures evolved are on the animation CDROM and frames from those animations are present in the full color appendix.

4.8.1 Hitting a ball

The genetic algorithm was quite successful at evolving creatures which could hit a ball at a fixed location. We used a population size of 300 creatures (as with all evolutions) and evolved the creatures for 200 generations. Every generation, we moved the ball a little bit further away from the root body part of the creatures, slowly complicating the task.

The effect of slowly making the task more difficult is clearly visible in figure 4.6a, which shows the maximum and average fitness of the creatures for each generation. The creatures learn how to perform the task (the fitness increases), but gradually it gets more and more difficult and the creatures get worse at

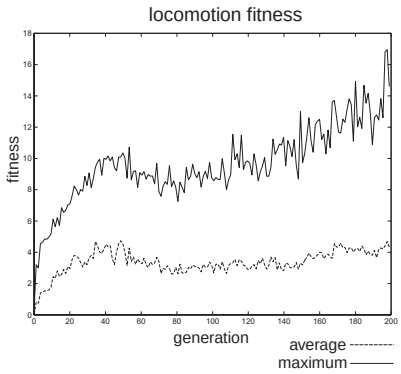

Figure 4.7: Fitness of the creatures during a *locomotion* evolution.

the task (the fitness decreases). Then some invention is done by the genetic algorithm (e.g. adding an extra body part to the creature such that it can reach further), and the fitness rises quickly again. This process happens multiple times during the evolution.

4.8.2 Hitting a ball which is not always at the same location

We were not able to evolve creatures for this task. During the first generations the evolved creatures were be able to hit the ball (because the task was still very simple), but as the ball moved further away, the genetic algorithm was unable to evolve creatures which could perform the task. GRAPH

4.8.3 Jumping straight up

We evolved creatures to jump straight up using two approaches. In one approach, gravity was set to a fixed value, in the other we slowly increased gravity each generation. Figure 4.6b shows the fitness of the creatures which were evolved using the latter approach for each generation. The effect of increasing the gravity is visible: the fitness is very high initially (gravity near 0), but as gravity increases, fitness drops, until some discovery is done after which fitness start to increase again.

We performed three evolutions with identical parameters (gravity, population size, number of generations, strength of the creatures). This resulted in nearly identical approaches to jumping every time (see the jumping 1 animation). When we changed the allowed shape of the bodies, different approaches emerged, of which one is shown in the jumping 2 animation.

4.8.4 Jumping away from the origin

We performed one evolution experiment where we taught the creatures to jump away from the origin. We specified a low gravity for the evolution to

make the task relatively simple. The approach which emerged, shown in the jumping 3 animation, consists of two steps. First the creature jumps as high as it can, then it starts moving erratically which, upon landing, results in velocity tangential to the surface.

4.8.5 Locomotion

We were not very successful at evolving creatures which could walk or otherwise move in a certain direction. Using the same parameters (creature strength, gravity) as for jumping resulted in creatures which would only move a few meters (the creatures have a size in the order of meters). In order to get at least some results, we increased the creatures' strength, disabled inter body part collisions (thus allowing the creatures to move more freely), and set the vector of gravity such that it pulls the creatures towards the x-axis a little. This resulted in creatures such as those shown in the move 1, 2 and three animations. Figure 4.7 shows a graph of the fitness of the creatures for one of the locomotion evolutions.

4.9 Discussion and conclusion

We have evolved creatures for a number of tasks with some success. The results of our evolution experiments were not as spectacular as those of Sims ([Sim94a] and [Sim94b]), but they are a good first step to building our own environment for evolving virtual creatures. We think that the results were not as good as expected or hoped for is mainly due to five reasons:

- The crossover operator. The current crossover operator only exchanges entire *nodes* (descriptions of body parts) between genotypes. This causes all information contained inside the node (local controller network, shape, joint description) to be kept inside the same node; it is not possible to exchange such information between nodes. We expect the genetic algorithm to benefit from a crossover operator which can exchange information at the sub-node level. For example, if an useful local controller network is discovered in some node, this network could be spread to other node via the crossover operator. This is currently not possible. Note however, that the crossover operator we currently use is the same as described in [Sim94a].
- Slow feedback from the evolution experiments. It typically took several hours to evolve a population of 300 individuals for 100 to 200 generations. These evolutions were usually done at night because of the availability of machines, so this implies 1 evolution per day. This made it difficult to enhance or tweak the program. Every change that is made requires an evolution experiment to verify whether the changes had any positive effect on the result.
- Parameter tuning. There are a lot of parameters inside the model of the creatures and the world, e.g. the amount of gravity, the physical proper-

ties of the bodies and the maximum size of the joints forces, which should be fine tuned to achieve optimal (read 'most spectacular') results. The slow feedback from the experiments and lack of time prevented us from doing so.

- The simulator. The simulator used to evolve the creatures still had some problems and was very experimental. As mentioned in section 3.12, the simulator would benefit from a total rewrite.
- Creature controllers. The neural network used to control the creatures' joint forces is very primitive; some might argue that it is not even a neural network. We tried however to stick as closely to the description Sims gave in [Sim94a] in an attempt to reproduce the results reported. We were hoping to implement another type of enhanced controller and compare the results of such a controller with the one Sims used, but lacked the time to do so.
- The number of generations evolved. From the graphs in section 4.8 we conclude that we did not let the evolutions last long enough. The average fitness and maximum fitness of the creatures were still increasing significantly each generation when we stopped the evolution. This suggest that if we had let the evolution continue for more generations the results would have improved.

We think the most important problem of the five listed above are the creature controllers. We were able to successfully evolve creatures for tasks which did not depend on any complicated processing of (sensory) information (such as hitting a ball which is always at the same location, or jumping), but unable to evolve creatures for tasks which did (hitting a ball which is not always in the same location, and locomotion to a lesser degree). The neural network and the way it is evolved should obviously be improved.

A problem with our genetic algorithm was the large number of mutation and recombination probability parameters. First of all we have three parameters which specify the probability of a certain genetic operator (mutation, crossover, grafting) being used. Secondly we can specify the probability of mutation occurring for a lot of members of the evolvable objects. Although this gives us a lot of control over which part of the genotype mutate and how often, it also presents us with the task of specifying reasonable probabilities. This could be done by using a meta-GA, a genetic algorithm that evolves (the parameters of) another genetic algorithm. This is unfeasible for our problem however, because of the long time it takes to do even *one* evolution, let alone running a meta-GA on top of our GA. We specified mutation parameters which seemed reasonable to us (compared to mutation probabilities reported in literature), and tweaked those a little, observing little change in the final results of each evolution.

The state space searched by our genetic algorithm is very large, only limited by the amount of memory and processing time available. It is almost unavoidable for the evolution to get lured into in a local minimum when a small (300) population size and a limited number of generations is used. It is interesting to

see we got nearly identical results for three jumping evolutions (in animation jump1). Even though the initial conditions were the same, one would expect different results from each evolution when such a large state space is available. In case of the three jump evolutions, either the evolution got trapped in the same local minimum every time or the task was so simple that the global optimum was very obvious.

Chapter 5

Discussion, Conclusion and Future Work

5.1 Discussion and Conclusion

During the work performed for this thesis we have studied modern algorithms and methods for collision detection and rigid body simulation. We implemented an efficient collision detection package and an impulse based rigid body simulator. We have used the rigid simulator to evolve virtual creatures much like [Sim94a] and [Sim94b].

We looked into several narrow phase collision detection algorithms such as [GJK88], [Lin93] and [Mir97], all of which have near $O(1)$ time complexity for situations when objects do not move much relative to each other. This means that the speed of the collision detection algorithm is almost independent of the complexity of the geometry.

We implemented the Lin-Canny algorithm and it's superior descendant, the V-Clip algorithm. We verified the theoretical time complexity properties of the algorithm empirically by performing a number of experiments where we varied object geometry complexity and the number of objects in the scene. Our measurements indicate that the theoretical time complexity properties of the algorithms are correct: the time complexity lies between $O(1)$ and $O(\sqrt{n})$ (n is the number of features in the object geometry) depending on the relative movement of the objects.

We made a minor extension to the V-Clip algorithm such that it can track the closest points between two *features*. These features can be any pair of features laying on two different objects, but this extension will mostly be used to track points on features which were previously closest.

To complement the V-Clip algorithm, we implemented the hierarchical hashing algorithm [Mir96], which is used to cull most pairs of objects which can not collide or intersect during a certain period. We performed experiments to verify the predicted $O(n + c)$ (where n is the number of objects and c the number of possibly colliding objects) time complexity of the combination of the hierar-

chical hashing and Lin-Canny-like algorithms. The experiments showed that the $O(n + c)$ time complexity is correct.

Before implementing the rigid body simulator, we studied (constrained) rigid body dynamics, and looked into several rigid body simulation methods. The main difference between these methods is how they handle contact. The methods examined were based on penalty / spring forces, LCP and (micro-)impulses. The penalty methods were rejected because they cause stiff integration problems and do not model the real world accurately (see for instance [Mir96]). LCP methods handle contact very efficiently if contact geometry does not change. The LCP algorithm is not guaranteed to terminate if static friction is present in the system. They are also more difficult to implement. We selected the impulses based method because it is relatively simple to implement and handles contact efficiently in most situations.

We extended the simulator as described in [Mir96] in an attempt to improve it's contact handling. We verified empirically that the extension was successful in terms of improving stability of the simulation of bodies which are at rest, but it is computationally more expensive.

We have performed a number of different simulations to demonstrate the capabilities of our simulator.

Using the simulator, we have evolved virtual creatures like those described in [Sim94a]. We were not as successful as Sims. We hypothesize this is mainly due to the creature controller networks and the lack of time to do parameter tuning. Since we attempted to implement the controllers the same way Sims did, either something is not functioning correctly in our implementation, or Sims did not describe (the way he evolved) his controllers accurately.

5.2 Future Work

The V-Clip collision detection algorithm is a very fast and accurate algorithm to determine the distance between two objects. It is unfortunate however that it can only handle convex objects. The prove of the central theorem (2) on which both Lin-Canny and V-Clip are based breaks if the geometry of the objects is not convex. Even though non convex objects can be split into several convex objects, it would a great improvement if the same performance would be possible for non-convex objects.

The simulator could be improved by rewriting it from scratch and incorporating better methods for contact handling. We have experimented a little with a hybrid micro-impulse / penalty force method where the maximum magnitude of the penalty forces is derived from the micro-impulses using a heuristic, and this method looks promising if combined with an implicit integrator to handle the stiffness introduced by the springs. This would require another algorithm (see e.g. [Lil93]) to compute the accelerations of constrained bodies, because the Jacobian matrix required by an implicit integrator is hard to compute using the Featherstone algorithm.

We have investigated a method to do the rigid body simulation in parallel, based on dividing the bodies into independent groups which are simu-

lated together, and timewarp. We lacked the time to implement this idea and it we will certainly investigate it further. Although conceived independently, [Mir00] uses many of the same ideas to speed up rigid body simulation on a *single* processor.

We experimented with real time interactive simulation in a virtual environment (the CAVE). The simulation results were very convincing to the participants. The author will certainly try to investigate such experiments further.

The author is looking forward to examine methods to simulate mass-spring models. Such models describe deformable bodies using number of vertices (representing mass) which are connected by springs with varying properties. When combined with an appropriate implicit integrator, such models could be used to simulate cloth or tissue of arbitrary stiffness (e.g. from bones to fat). It would be interesting to evolve creatures built not from rigid but deformable bodies.

Work will have to be done to improve the creature controllers since this seems to be the major problem keeping us from evolving better performing creatures. Implementing any type of modern evolvable controller would probably improve the results a lot.

A very interesting step forward would be to incorporate growth and development into the creature model and to let the creatures exist in a continuous long lasting virtual world instead of simulating them only for a limited time.

Appendix A

Animations

This appendix lists the animations present on the CDROM and the full color appendix. The CDROM contains all animations in multiple formats (avi, mpeg1, mpeg2). The full color appendix contains only a few frames of a number of animations. The locations of the animations are listed in table A.1. Table A.1 also lists which frames are present in the full color appendix.

A.1 Collision detection

All collision detection animations were made using the bounce demonstration program (section 2.6). The frames were rendered using OpenGL, grabbed, stored in separate pictures and compressed into an animation file.

Collision Detection Animation 1

Collision Detection Animation 1 demonstrates object geometry (section 2.2) and the V-Clip algorithm (section 2.4). The first 15 seconds of the animation show two objects drawn as wireframes while the V-Clip algorithm tracks the closest features and points between the two objects. The features are drawn as bold white points and lines, while a thin white line connects the closest points between the two objects. The next 15 seconds of the animation shows the two objects as solids, with the V-Clip algorithm still in action. The tracking of features over the two surfaces is clearly visible.

Collision Detection Animation 2

This animation demonstrates the hierarchical hashing algorithm. The first 7 seconds show the V-Clip algorithm being turned on and off as the distance between the objects varies. The next 5 seconds the bounding boxes are shown and after that the hashed tiles are shown. See section 2.3 for a detailed description of the algorithm.

Table A.1: Animations. Replace 'fmt' with the appropriate format extension ('avi', 'mpg' or 'm2v').

name	CDROM	full color appendix
Collision Detection Animation 1	/fmt/cd/anim1.fmt	frames 191 and 641
Collision Detection Animation 2	/fmt/cd/anim2.fmt	-
Collision Detection Animation 3	/fmt/cd/anim3.fmt	-
Collision Detection Animation 4	/fmt/cd/anim4.fmt	frames 10 and 260
Friction animation	/fmt/rbs/friction.fmt	-
Restitution animation	/fmt/rbs/rest.fmt	-
Brick wall stability	/fmt/rbs/bwstab.fmt	frames 1, 1001, 2001, 3001, 4001 and 4691
Brick wall smashed by car	/fmt/rbs/bwcar.fmt	frames 60, 80, 100, 120, 140, 160, 180, 200, 220
Single brick wall smashed by demolition chain	/fmt/rbs/bwchain1.fmt	-
Double brick wall smashed by demolition chain	/fmt/rbs/bwchain2.fmt	frames 30, 50, 70, 90, 110 and 130
Coins	/fmt/rbs/coins.fmt	frame 83
Dominos	/fmt/rbs/dominos.fmt	frames 75, 395, 520 and 640
Marble	/fmt/rbs/marble.fmt	frames 52 and 536
Hitting ball	/fmt/vc/ball.fmt	frames 622, 643 and 660
Jumping 1	/fmt/vc/jump1.fmt	frames 1, 10 and 27
Jumping 2	/fmt/vc/jump2.fmt	frames 1, 42, 55, 65, 98 and 163
Jumping 3	/fmt/vc/jump3.fmt	-
Locomotion 1	/fmt/vc/move1.fmt	1, 52, 87, 121, 178 and 241
Locomotion 2	/fmt/vc/move2.fmt	-
Locomotion 3	/fmt/vc/move3.fmt	-

Collision Detection Animations 3 and 4

These two animation show the effect of the hierarchical hashing algorithm on the number of pairs for which the V-Clip algorithm is invoked. Both animation start with a run of the bounce program with the hashing algorithm turn *off*, followed by an identical run with the hashing algorithm turned *on*. We used 8 objects in animation 3, and 25 objects in animation 4. Animation 3 shows the scene both with and without geometry.

A.2 Rigid Body Simulation

All rigid body simulation animation were made by simulating the scene using our simulator and storing the results in an animation file. This file was later imported into a 3d rendering package, where the camera, lights, materials and all other rendering properties were set, after which the animation was rendered and compressed into an animation file.

Friction

This animation shows 7 cubes sliding down a sloped surface. The friction parameters of the cubes are varied from no friction at all to just enough friction to stay put.

Restitution

The restitution animation shows 7 spheres bouncing up and down. The restitution parameters of the balls varies from 0 to 1.

Brick wall stability

The brick wall stability animation shows simulations described in section 3.11.1. The brick wall on the left is simulated with the collision epsilon set to 10% of the size of the bodies and with *standard* contact handling; the brick wall on the right is simulated with the collision epsilon set to 10% of the size of the bodies and with *improved* contact handling (as described in section 3.10). The recordings of the simulations were merged inside the 3d rendering package. Both simulations had a length of more than 3 minutes, but are played back 10 times as fast.

Brick wall collisions

Three simulations of bodies colliding with brick walls have been performed. A car driving down a slope and crashing into the wall, a demolition ball smashing a single brick wall and a demolition ball smashing a double brick wall.

Coin toss

The coin toss animation shows the simulation of 18 coins.

Dominos

The domino track animation shows the results of simulating 150 domino bricks falling over. The dominos are arranged along a tunnel, stairs, a tower and a diving board.

Cartrack

The cartrack animation shows a car with damped springs between it's wheel and chassis driving over two obstacles and through a looping.

Marble

This animation shows the simulation of a marble rolling through a track of sloped ridges, trapdoors and gearwheels. The simulation is played back at 8 speed, because it lasts about 3 minutes.

A.3 Virtual Creatures

Just like the rigid body simulation animations above, the virtual creatures animations were made by importing the simulation results into a 3d rendering package. In some animations multiple creatures are shown alongside each other; in that case multiple scenes were merged inside the 3d rendering package.

Hitting ball

This animation shows the best performing creatures of several generations for the hit ball task.

Jumping 1

The jumping 1 animation shows the final result of three evolutions for the jump straight up task. A similar approach emerged from each evolution.

Jumping 2

These creatures were evolved using the same task and parameters as those shown in the jumping 1 animation, except they had to use spheres as body parts. Creatures from different generations are shown (the earliest generation shown up front, the newest in the back).

Jumping 3

A creature evolved for the jumping away from the origin task.

Locomotion 1, 2 and 3

These animations show the creatures which were evolved for the locomotion task.

Bibliography

[Bar90] D. Baraff *Curved surfaces and coherence for non-penetrating rigid body simulation* Computer Graphics, Annual Conference Series, SIGGGRAPH '90 Proceedings, August 1990, pp. 19-28.

[Bar92] D. Baraff. *Dynamic Simulation of Non-Penetrating Rigid Bodies.* PhD thesis, Department of Computer Science, Cornell University, March 1992.

[Bar94] D. Baraff. *Fast Contact Force Computation for Nonpenetrating Rigid Bodies*. Siggraph Conference Proceedings, ACM Press, 1994.

[Bar96] D. Baraff. *Linear-time dynamics using lagrange multipliers*. Siggraph Conference Proceedings, ACM Press, 1996.

[BDH96] C.B. Barber, D.P. Dobkin, and H.T. Huhdanpaa, *The Quickhull algorithm for convex hulls* ACM Transactions on Mathematical Software, vol. 22, pp. 469-483, Dec 1996.

[CR98] A. Chatterjee and A. Ruina *A New Algebraic Rigid Body Collision Law Based On Impulse Space Considerations* Journal of Applied Mechanics. 1998.

[CSD93] C. Cruz-Neira, D.J. Sandin, and T.A. DeFanti. *Surround-screen projection-based virtual reality: The design and implementation of the CAVE*. SIGGRAPH'93 pages 135-142. ACM SIGGRAPH, August 1993.

[DLMP95] J.D. Cohen, M.C. Lin, D. Manocha and M.K.Ponamgi. *I-Collide: An interactive and exact collision detection system for large-scaled environments*. ACM Siggraph, ACM Siggraph, April 1995.

[Gol89] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning* Addision-Wesley, 1989.

[GLD93] S. Gull, A. Lasenby, C. Doran. *Imaginary Numbers are not Real. The Geometric Algebra of Spacetime* Cambridge University, 1993.

[Fea87] R. Featherstone. *Robot dynamics algorithms* Kluwer Academic Publishers, 1987.

[dG90] H. de Garis. *Genetic Programming: Building Artificial Nervous Systems Using Genetically Programmed Neural Network Modules* Proceedings of the 7th International Conference on Machine Learning. 1990, pp.132-139.

[GvL96] G.H. Golub C.F van Loan. *Matrix Computations, third edition* The John Hopkins University Press, 1996.

[GJK88] E. G. Gilbert, D. W. Johnson and S. Sathiya Keerthi. *A fast procecure for computing the distance between complex object in three dimensional space.* IEEE Journal of Robotics and Automation. 4(2):192-203, April 1988.

[Hah88] J. K. Hahn. *Realistic animation of rigid bodies*. Computer Graphics, 22(4):299-308, August 1988.

[HLCGM97] T.C. Hudson, M.C. Lin, J. Cohen, S. Gottschalk, D. Manocha. *V-Collide: Accelerated Collision Detection for VRML*. In Proceedings of VRML 97.

[Hol75] J.H. Holland *Adaptation in natural and Artificial systems*. University of Michican Press, 1975.

[Jef85] D. R. Jeffeson. *Virtual Time*. ACM Transactions on programming languages and Systems, Vol 7, No 3, July 1985

[Kui99] J. B. Kuipers. *Quaternions and Rotation Sequences* Princeton University Press, 1999

[Lin93] M. C. Lin. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, University of California, Berkeley, December 1993.

[Lil93] K. W. Lilly. *Efficient Simulation of Robotic Mechanisms* Kluwer Academic Publishers, Norwell, 1993.

[LP00] H. Lipson and J.B. Pollack *Evolution of Machines* http://www.demo.cs.brandeis.edu/golem

[NM93] J.T. Ngo and J. Marks. *Spacetime constraints revisited* Computer Graphics, Annual Conference Series 1993, pp.343-350.

[MK86] J.L. Meriam, L.G. Kraige. *Engineering Mechanics Volume 2: Dynamics* John Wiley & Sons, Inc., New York, 1986.

[Mic96] Z. Michalewicz *Genetic Algorithms + Data Structures = Evolution Programs, Third, Revised and Extended Edition* Springer Verlag 1996.

[Mir96] B. Mirtich. *Impulse-based dynamic simulation of robotic systems*. Phd thesis, University of California, Berkeley, Fall 1996.

[Mir97] B. Mirtich. *V-Clip: Fast and Robust Polyhedral Collision Detection*. Technical Report TR97-05. Mitshubishi Electric Research Lab, Cambridge, MA, July 1997

[Mir98] B. Mirtich. *Rigid Body Contact: Collision Detection to Force Computation*. IEEE International Conference on Robotics and Automation, May 1998

[Mir99] B. Mirtich. *Comparing Diffuse and True Coevolution in a Physics-Based World*. 1999 Genetic and Evolutionary Computation Conference (GECCO).

[Mir00] B. Mirtich. *Timewarp Rigid Body Simulation*. Computer Graphics, Annual Conference Series 2000, pp. 193-200.

[MW88] M. Moore and J. Wilhelms. *Collision detection and response for computer animation*. Computer Graphics, 22(4):289-298, August 1988.

[Over94] M. Overmars. *Point location in fat subdivisions*. Information Processing Letters, 44:261-265, 1992.

[Pau95] E. Paulos. *Parallel Impulse Based Simulation*. http://robotics.eecs.berkeley.edu:80/~paulos/IMP/

[PF93] M van de Panne and E.Fiume *Sensor Actuator Networks*. Computer Graphics, Annual Conference Series 1993, pp.335-342.

[PTVF92] W.H. Press, S.A. Teukolsky, W.T. Vettering and B.R. Flannery. *Numerical Recipies in C* Cambrige University Press, Cambridge.

[Rey94] C. Reynolds *Competition, Coevolution and the Game of Tag* Artificial Life IV Proceedings, MIT Press, 1994.

[Sim94a] K. Sims *Evolving Virtual Creatures* Computer Graphics, Annual Conference Series, SIGGGRAPH '94 Proceedings, July 1994, pp. 15-22.

[Sim94b] K. Sims *Evolving 3D morphology and Behavior by Competion* Artificial Life IV Proceedings, MIT Press, 1994, pp. 28-39.

[XTu96] X. Tu *Artificial Animals for Computer Animation* PhD Thesis, University of Toronto.