University of Oslo
Department of Informatics

Simulation of Rigid Body Dynamics

Ståle Waage Pedersen

25th October 2003

Preface

This thesis documents my work on the Cand. Scient. degree in Computer Science at the University of Oslo, faculty of mathematics and natural sciences, department of informatics, direction of Computational Mathematics.

I would like to thank Trond Gaarder for good cooperation in the early phase of this thesis and several helping hints along the way.

Further i would thank my supervisor Hans Petter Langtangen for useful feedback on the writing of this thesis.

Abstract

In this thesis the problems of simulating rigid body dynamics are discussed and a library is provided that can be reused without dealing with the more complicated problems of rigid body simulation as collision detection and contact handling.

Contents

List of Figures

Chapter 1

Introduction

The difference between what we can model and visualize on a computer versus what we can physically simulate has become quite large. There are many good techniques for creating high-quality images for complex models, but our ability to perform realistic physical simulation on these models lags far behind our ability to visualize them.

In this thesis the problems of simulating rigid body dynamics are discussed and a library is provided that can be reused without dealing with the more complicated problems of rigid body simulation as collision detection and contact handling.

1.1 Rigid Body

So what is a rigid body? A rigid body is an object that does not deform under movement or collision. You can imagine a solid steel ball which is impossible to deform. There are no such thing as a perfectly rigid object in the real world. All objects deform somewhat on impact and when accelerating. But the deformation is often so small that we can ignore it, and the result will still be satisfactory.

1.2 Rigid Body Simulation

In computer graphics and robotics applications, a major concern is modeling systems and object motions over time. Since rigid body objects are impenetrable, it is important that the simulations correctly handle the issues of collision and contact between object accurately.

Rigid body simulation with non-interpenetration constraints is important for the following reasons:

- Rigid body simulation can help us more easily understand and visualize complicated mechanical systems and processes.
- A simulated physical environment can be used as a natural intuitive mean of interaction with many design and modeling tasks. The ability to interactively move 3D objects could greatly simplify many computer-aided design and layout systems.
- Rigid body simulation can be used to perform experiments and test hypothesis in situations for which real-world experiments would be difficult, costly, or impractical to perform.
- Realistic simulation is an extremely powerful method of creating realistic computer animations.

1.3 Simulator

A simulator can be seen as an advanced calculator. We can tell the simulator which objects it should simulate and give it their mass properties, positions ant their initial velocities. This information is called the initial state or initial configuration of the simulator. The simulator also needs to know about the general forces as gravity and air resistance. When the simulator has all the initial information it needs, it is the simulators task to calculate the final end state. This end state could be after 5 seconds or it can go on as long as we want. The simulator usually does not calculate the end state in a single calculation, it frequently calculates several in between states also called time-steps before it reaches its end state.

A simulator is frequently used in animations and interactive applications. Here the simulator is not given an end state. It will continue until it receives a stop signal. The simulator only returns numbers as position and orientation, the visualisation is created by another program or task which receives the numbers from the simulator.

Interactive applications or animations has something called a frame rate. Frame rate is measured in frames per second (fps). If the goal is to have a realtime simulator the average frame rate should be no less than 30 fps.

1.4 Dynamic Simulation

Dynamics can be defined most easily in terms of a closely related field, kinematics. Kinematics is the study of movement over time. Kinematics does not concern itself with what is causing movement or how things get where they are in the first place, it just deals with the actual movement itself. Dynamics, on the other hand is the study of the causes of motion. A dynamic system is the study of systems governed by ordinary differential equations including the trajectory of the system, stability, and periodicity. A dynamic simulator try to simulate this.

Dynamic simulation is often divided in two categories called forward kinematics and inverse kinematics. In a system where we have a initial and an end state and we want to calculate backward from end to start, is called inverse kinematics. It is commonly used when creating animations, because it is easy to control items in the simulation and alter the end or initial conditions to get the desired result.

With forward kinematics only a initial condition is known. When the simulations start it is only possible to observe what happens. Forward kinematics is very suitable in interactive applications such as flight, car and weather simulations, in robotics and CAD design.

1.5 Collision Detection

It is not possible to simulate rigid body dynamics without a decent collision detection routine. Without a collision detection routine it is only possible to simulate the movement of rigid body objects, not interaction between them.

Although collision detection is considered a different subject, it is heavily interleaved with rigid body dynamics simulation. According to Brian Mirtich [Mir96] the most difficult aspect of rigid body simulation is contact modeling. Two subproblems of contact modeling are detecting contacts and computing contact forces. Much of the time spent on this thesis were on creating a collision detection routine. A thorough introduction to the problems of collision detection are covered in Chapter 4, Collision Detection.

1.6 Programming Language

The primary language in this thesis will be Java[Gos96], with the use of Java3D[Mic97] when dealing with graphics. According to [Gal01] over half of all U.S. developers use Java and this share was projected to rise with an additional ten percent during 2002.

Though Java is the most popular programming language in the world, it has not been widely accepted by scientific programmers yet. The common perception of Java as slow, is most likely the sole reason for this. But with the development of Java the last years it is interesting to see how Java now performs, with and without 3D hardware support.

A technical introduction to Java is found in chapter 8, Java.

1.7 Where to use Rigid Body Dynamics

Rigid body dynamics has been used for many years in robotics and other simulation fields. Several car manufactures use it when developing cars and car parts. It has also been used within virtual reality research.
Lately there has been a lot of discussion around next generation games. With the ever evolving development of hardware, there will be other areas than graphics that is likely to dominate future games. One of those areas will probably be physics. With the use of rigid body dynamics it is possible to make games and animations look even more natural than before.

1.8 Thesis Organization

Chapter 1 This chapter.

Chapter 2 Covers the theory of rigid body dynamics.

Chapter 3 Introduction to the implementation of the theory that is covered in chapter 2. Quaternions and its use is also covered.

Chapter 4 Describes collision detection theory and covers implementation of a technique.

Chapter 5 Describes the implementation of the rigid body library.

1.8 Thesis Organization

Chapter 6 Covers the theory of contact handling and an implementation.

Chapter 7 Covers ordinary differential equations.

Chapter 8 A short introduction to Java technology.

Chapter 9 Conclusion.

Chapter 2

Rigid Body Dynamics Theory

This section is intended as an introduction to the theory of simulating rigid body motion. The reader is assumed to have knowledge in linear algebra, calculus and some understanding of classical mechanics.

The "rigid body" part of rigid body dynamics refers to constraints we place upon the objects we are simulating. A rigid body's shape does not change during simulation. Or in other words, no element of matter in a body is able to translate and rotate with respect to any other element of matter in that body. The reason we use rigid bodies is actually just that, they do not deform under simulation. This means that they have some properties that make their motion easier to deal with. One is that their center of mass is fixed. When a rigid body is rotating, every mass element within it has angular momentum with respect to the center of mass.

2.1 Kinematics

First let us define a body coordinate system, in which all our dynamic variables can be specified. Because a rigid body can only undergo rotation and translation, we define the shape of a rigid body in terms of a fixed and unchanging space called body space. To locate the body in world space we will use a vector $x(t)$, which describes the translation of the body. To describe the rotation of the body, we will use a 3x3 rotation matrix $R(t)$.

2.1.1 Mass

In general, people think of mass as a measure of the amount of matter in a body. We can also think of mass as a measure of a body's resistance to motion or a change of motion. The greater the body's mass, the harder it will be to set or change its motion. Temporarily we imagine that the body is made up of a large number of small particles. The particles are indexed from 1 to N. The mass of the i'th particle is m_i, and each particle has a (constant) location r_{0i} in body space. The location of the i'th particle in world space at time t, denoted $r_i(t)$.

2.1.2 Center of mass

The center of mass is the point through which any force can act on the body without resulting in a rotation of the body. The center of mass is used to describe the position of the body in world space. The position of the center of mass is calculated with a weighted sum of every mass element, m_i, in the body:

$$r_{cm} = \frac{\sum r_i m_i}{\sum m_i} = \frac{\sum r_i m_i}{M}$$

When the mass is continuously distributed throughout its volume, this sum becomes the integral:

$$r_{cm} = \frac{\int r\rho(r)dV}{\int \rho(r)dV} = \frac{r\rho(r)dV}{M}$$

In this case, each mass element is calculated by multiplying a volume element dV by a three-dimensional density function $\rho(\gamma)$:

$$m_i = \rho(r)dV$$

2.1.3 Velocity

We call $x(t)$ and $R(t)$ the position and orientation of the body at time t. Now we need to define how the position and orientation change over time.

If at time t_0 a body's position is x_1, and at time t_1 its position is x_1, then its average velocity between t_1 and t_2 is:

$$v_{ave} = \frac{x_1 - x_0}{t_1 - t_0} = \frac{\Delta x}{\Delta t}$$

If we let $\Delta t \rightarrow 0$, the limit is the instantaneous velocity of the particle, which is the true velocity of the particle at any time, t, and is equal to the derivate of its position with respect to time:

$$v = \lim_{\Delta t \to 0} \frac{\Delta x}{\Delta t} = \frac{dx}{dt}$$

Similarly, if its velocity changes from one instant to another, it is said to be accelerating:

$$a = \lim_{\Delta t \to 0} \frac{\Delta v}{\Delta t} = \frac{dv}{dt}$$

2.1.4 Angular Velocity

In addition to translation, a rigid body can also undergo rotation. The amount of rotation the body experiences per unit time is called its angular velocity, given by:

$$\lim_{\Delta t \to 0} \frac{\Delta \theta}{\Delta t} = \omega$$

Where $\Delta\theta$ is a very small rotation (in radians) and ω is the angular velocity about the center of mass.

Actually, finite rotations, no matter how small, cannot be considered vectors because they are not commutative (but that is out of scope of this thesis).

If a vector r is rotating at a constant angular velocity, then its time derivative with respect to the fixed world frame is:

$$\frac{dr}{dt} = \frac{\partial r}{\partial t} + \omega \times r$$

If the length is not changing, then the derivative simplifies to:

$$\frac{dr}{dt} = \omega \times r$$

Using this relationship, the time derivative of R is:

$$\frac{dR}{dt} = \omega^* R$$

where the antisymmetric matrix:

$$\omega^* = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$$

takes the place of the cross product [BW97]

2.2 Forces and Torques

The location of the particle the force acts on, defines the location at which the force acts. We will let $F_i(t)$ denote the total force from external forces acting on the ith particle at time t. Also, we define the external torque $\tau_i(t)$ acting on the ith particle as

$$\tau_i(t) = ((r_i(t) - x(t)) \times F_i(t)$$

Torque differs from force in that torque on a particle depends on the location $r_i(t)$ of the particle, relative to the center of mass $x(t)$. We can intuitively think of the direction of $\tau_i(t)$ as being the axis the body would spin about due to $F_i(t)$, if the center of mass were held firmly in place.

2.2.1 Momentum

Newton's first law of motion states that a body remains stationary or maintains a constant velocity unless acted on by an external force. This is also known as the law of conservation of linear momentum. The linear momentum vector, p, of a body is calculated by multiplying its velocity v, by its mass, m:

$$p = mv$$

The rate of change and momentum with respect to time is equal to the sum of all to forces (the net force) on this body:

$$F_{net} = \sum F_i = \frac{dp}{dt} = m\frac{dv}{dt} = ma$$

- this is better known as Newton's second law of motion.

2.2.2 Angular Momentum

When a body is moving relative to a point of reference and its motion is not directly toward or away from that point, it is said to have angular momentum with respect to that point. The angular momentum vector $L(t)$ of a rigid body is defined by $L(t) = I(t)\omega(t)$, where $I(t)$ is a 3x3 matrix called the inertia tensor. The inertia tensor $I(t)$ describes how the mass in a body is distributed relative to the body's center of mass. The inertia tensor will be described in more detail shortly. The relationship between $L(t)$ and the total torque $\tau(t)$ is:

$$\dot{L}(t) = \tau(t)$$

2.2.3 Inertia tensor

The inertia tensor $I(t)$ is the scaling factor between angular momentum $L(t)$ and angular velocity $\omega(t)$. In the same way mass can be seen as a measure of a body's resistance to motion, moments of inertia is a measure of a body's resistance to rotational motion.

We know that the body's total angular momentum (in world space) about its center of mass is the sum of all the elements/parts of the body:

$$L(t) = \sum r'_i \times p_i = \sum r'_i \times (m_i v_i)$$

where r'_i is the vector from $x(t)$ to r_i. Since the velocity of m_i is given by:

$$v_i = \omega \times r'_i$$

we can write:

$$L(t) = \sum m_i r'_i \times (\omega \times r'_i) = -\sum m_i r'_i \times (r'_i \times \omega) = -\sum m_i r^*_i r^*_i \omega$$

where:

$$r^*_i = \begin{bmatrix} 0 & -r'_{iz} & r'_{iy} \\ r'_{iz} & 0 & -r'_{ix} \\ -r'_{iy} & r'_{ix} & 0 \end{bmatrix}$$

Substituting and multiplying through gives:

$$L(t) = \sum \begin{bmatrix} m_i \left(r'^2_{iy} + r'^2_{iz}\right) & -m'_i r_{ix} r'_{iy} & -m_i r'_{ix} r'_{iz} \\ -m_i r'_{ix} r'_{iy} & m_i \left(r'^2_{ix} + r'^2_{iz}\right) & -m_i r'_{iy} r_{iz} \\ -m_i r'_{ix} r_{iz} & -m_i r'_{iy} r_{iz} & m_i \left(r'^2_{ix} + r'^2_{iy}\right) \end{bmatrix} \omega$$

$$= \begin{bmatrix} \sum m_i \left(r'^2_{iy} + r'^2_{iz}\right) & \sum -m'_i r_{ix} r'_{iy} & \sum -m_i r'_{ix} r'_{iz} \\ \sum -m_i r'_{ix} r'_{iy} & \sum m_i \left(r'^2_{ix} + r'^2_{iz}\right) & \sum -m_i r'_{iy} r_{iz} \\ \sum -m_i r'_{ix} r_{iz} & \sum -m_i r'_{iy} r_{iz} & \sum m_i \left(r'^2_{ix} + r'^2_{iy}\right) \end{bmatrix} \omega$$

This symmetric matrix of sums is called the inertia tensor, I, where:

$$I(t) = \begin{bmatrix} I_{xx} & I_{yx} & I_{zx} \\ I_{xy} & I_{yy} & I_{zy} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix}$$

The diagonal elements are called the moments of inertia, and the off-diagonal elements are called the products of inertia. Since r'_i has been specified with respect to world coordinates, the inertia tensor depends on the body's orientation and must be recalculated every time the body rotates.

However, by diagonalizing the inertia tensor we can avoid to re-evaluate the integrals after every rotation. Diagonalization of a matrix involves changing to a basis in which all the off-diagonal elements become zero. This basis is unique and consists of the eigenvectors of the matrix. The diagonal elements with respect to this basis are called the eigenvalues of the matrix.

The normalized eigenvectors of the inertia tensor are called the principal axes of the rigid body, and the eigenvalues are called the principal moments of inertia. With respect to a body's principal axes, the inertia reduces to:

$$I_{body} = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$$

Where the moments of inertia are calculated with the use of body spaced coordinates. For rigid bodies, these integrals need to be calculated only once, and the inertia tensor in world space is given by:

$$I(t) = R(t) I_{body} R(t)^T$$

[BW97]

Chapter 3

Simulating Rigid Body Dynamics

3.1 Mass Properties

Rigid body systems requires several parameters describing the mass distribution of rigid bodies: the total mass (a scalar), the location of the center of mass (3 parameters), and the moments and products of inertia about the center of mass (6 parameters).

A rigid body comprising N parts, $B_1, ...B_N$, each a uniform density polyhedron. There are no restrictions on the convexity or genus of the polyhedral, nor on the shape of the bounding faces. For each polyhedron B_i, either its density p_i or mass m_i is specified, and the geometries of all of the polyhedral are specified relative to a single reference frame

The mass and center of mass is relatively easy to implement. The moments and products of inertia is more difficult to calculate. For simple uniform geometries there are specific formulas.

3.2 Rigid Body Equations of Motion

The state of a rigid body is its position $x(t)$, orientation $R(t)$, momentum $P(t)$ and angular momentum $L(t)$. The mass M of the body and the body-space inertia tensor I_{body} are constants, which we assume we know when the simulation begins.

The inertia tensor $I(t)$, angular velocity $\omega(t)$ and velocity $v(t)$ are computed by:

$$v(t) = \frac{P(t)}{M}, \quad I(t) = R(t)I_{body}R(t)^T, \quad and \quad \omega(t) = I(t)L(t)$$

3.2 Rigid Body Equations of Motion

The derivatives for position, orientation, momentum and angular momentum are:

$$\frac{d}{dt}(x(t)) = v(t)$$
$$\frac{d}{dt}(R(t)) = \omega(t) * R(t)$$
$$\frac{d}{dt}(P(t)) = F(t)$$
$$\frac{d}{dt}(L(t)) = t(t)$$

Using this datatypes we can represent a rigid body as:

```
class RigidBody3D {

  // The world coordinate position of the body
  private MyVector3f position;          // x(t)

  // The linear velocity of the body
  private MyVector3f velocity;          // v(t)

  // Mass of the body
  private float mass;                   // M

  // The orientation of the body
  private MyQuat4f orientation;         // R(t)

  // The angular velocity of the body
  private MyVector3f omega;             // w(t)

  // The inertia tensor of the body in body coordinates
  private MyMatrix3f inertia;           // I(t)

  // Linear moment
  private MyVector3f P;

  // Angular moment
  private MyVector3f L;
}
```

Here we have used a quaternion to represent the rotation instead of a matrix. A quaternion is better suited as a rotation representation than a matrix. The reason for this and a introduction to quaternions is provided in 3.3.

For each timestep we need solve the equations of motion using numerical integration techniques. The equations of motion can be solved

by the use of ordinary differential equations (ODE). A introduction to ODE can be found in section 3.4.

A simple Euler implementation of the equations of motion:

```
public void update(float dt) {
    // velocity = P / mass
    velocity.setScaleAdd(getInverseMass(), P);

    // position += dt*velocity + position
    position.scaleAdd(dt, velocity, position);

    // q'(t) = 0.5 * omega(t) * orientation(t) * dt
    // tmpQuat = q'(t)
    RBDQuat4f tmpQuat = new RBDQuat4f();
    tmpQuat.mul(omega, orientation);
    tmpQuat.scale(0.5f * dt);

    // orientation += q'
    orientation.add(tmpQuat);

    // must normalize orientation to prevent drift
    orientation.normalize();
}
```

As described in the ODE section the Euler method is not suitable to be used for a numerical solver since it not stable. A more efficient and stable method would be midpoint Euler or the Runge-Kutta method.

3.3 Quaternions

During simulation a rotational (3x3) matrix will suffer from numerical drift. When this happens the rotational matrix will not only rotate the body, but also scale and share it. To prevent this the rotational matrix must frequently be reorthogonalized. This is an expensive operation and should be avoided. A solution is to use *unit quaternions*, a type of four element vector, normalized to unit length. A quaternion $\mathbf{q} \in H$ is given by:

$$\mathbf{q} = s + v_x\mathbf{i} + v_y\mathbf{j} + v_z\mathbf{k}$$

Since quaternions have only four parameters, only one extra variable being used to describe the three degrees of freedom; therefore, the degree of redundancy is noticeably lower for quaternions than rotation

matrices. This means that quaternions experience far less drift than rotation matrices. If the quaternion loose its unit magnitude it can be easily correctable by renormalizing the quaternion to unit length. Because of this it is desirable to represent the rotation of a body by a quaternion $q(t)$. We still need the rotation matrix to calculate the inverse inertia tensor, but it will be computed from the quaternion. We will write the quaternion has the pair:

$$[s, v]$$

Some quaternion definitions:
Conjugate:

$$\mathbf{q}^* = [s, -v] = s - v_x\mathbf{i} - v_y\mathbf{j} - v_z\mathbf{k}$$

Inverse:

$$\mathbf{q}^{-1} = \frac{\mathbf{q}^*}{\mathbf{q}^*\mathbf{q}}$$

A rotation of θ radians about a unit axis u is represented by the unit quaternion

$$[cos(\theta/2), sin(\theta/2)u]$$

Rotate a vector v with a quaternion q, we define the vector as a pure quaternion $p = [0, v]$. The rotated is then:

$$\left[0, \mathbf{v}^{'}\right] = \mathbf{p}^{'} = \mathbf{qpq}^*$$

To use a quaternion to represent rotation we need a formula for $\dot{\mathbf{q}}(t)$. It is a lengthy derivation, so the only the formula is given here:

$$\dot{\mathbf{q}}(t) = \frac{1}{2}\omega(t)\mathbf{q}(t)$$

where the multiplication $\omega(t)\mathbf{q}(t)$ is shorthand for multiplication between the quaternions $[0, \omega(t)]$ and $\mathbf{q}(t)$.

We know all we need to effective use quaternion as a representation of rotation.

3.4 Ordinary Differential Equation

3.4.1 Initial Value Problem

Differential equations describe the relation between an unknown function and its derivatives. To solve a differential equation is to find a

function that satisfies the relation, typically while satisfying some additional conditions as well. Here we will be concerned primarily with a particular class of problems, called initial value problems. In a initial value problem, the behavior of the system is described by an ordinary differential equation (ODE).

Standard introductory differential equation courses focus on symbolic solutions, in which the functional form for the unknown function is to be guessed.

We will exclusively be concerned with numerical solutions, on which we take discrete time steps starting with the initial value $f(t_0)$. To take a step, we see the derivate function f to calculate an approximate change in f, Δf, over a time interval Δt, then increment f by Δf to obtain the new value.

Numerical methods operate by performing one or more of these derivative evaluations at each time step.

3.4.2 Euler Method

One of the oldest and simplest numerical methods is Euler's method. Euler's method is derived from Taylor's sentence. When we want to find the value of $f(t_0 + h)$, we can do this in N steps. These steps are all equal, which means we have:

$$h = \frac{t_1 - t_0}{N}$$

where h is the stepsize parameter. We can find approximations of $f(t_i)$ for all t_i:

$$t_i = a + ih$$

where $i = 1, 2, 3, ..., N$. To find the value of $f(t_{i+1})$ we can use the Taylor sentence:

$$f(t_{i+1}) = f(t_i) + (t_{i+1} - t_i) f^{'}(t_i) + \frac{(t_{i+1} - t_i)^2}{2!} f^{''}(\epsilon_i)$$

For some value $t_i < \epsilon_i < t_{i+1}$. We can write this as:

$$f(t_{i+1}) = f(t_i) + hf^{'}(t_i) + \frac{h^2}{2!} f^{''}(\epsilon_i)$$

We get the Euler's method by ignoring the last part. We use the remaining part to calculate $f(t_{i+1})$. From this we get:

$$w_{i+1} = w_i + hf(t, w_i)$$

Here w_i is an approximation to $f(t_i)$, w_{i+1} is the approximation we want to find for $f(t_{i+1})$.

Though simple, Euler's method is not accurate nor stable. Shrinking the stepsize will slow the rate of drift, but never eliminate it. Finally, Euler's method is not even efficient. Most numerical solution methods spend nearly all their time performing derivative evaluations, so the computational cost per step is determined by the number of evaluations per step. Though Euler's method only requires one evaluation per step, the real efficiency - as well as on the cost per step. More sophisticated methods, even some requiring as many as four or five evaluations per step, can greatly outperform Euler's method because their higher cost step is more than offset by the larger stepsizes they allow.

3.4.3 Runge-Kutta

A fourth order Runge-Kutta is another numerical method to solve a differential equation. As Euler's method we have N steps, Runge-Kutta is also derived from Taylor's sentence, but with 2 variables instead of 1 as the Euler method. The differential equation of a fourth order Runge-Kutta is:

$$\begin{array}{l} k_1 = hf(t_i, w_i) \\ k_2 = hf(t_i + h/2, w_i + k_1/2) \\ k_3 = hf(t_i + h/2, w_i + k_2/2) \\ k_4 = hf(t_i + h, w_i + k_3) \end{array} \quad w_{i+1} = w_i + (k_1 + 2k_2 + 2k_3 + k_4)/6$$

The difference between Euler and a fourth order Runge-Kutta has a much smaller local truncation order error. Runge-Kutta has a local truncation order error on $O(h_4)$, where Euler's method as $O(h)$. Since h is small $h \ll 1$ the Runga-Kutta error is small and we can use a larger stepsize than with the Euler method.

Chapter 4

Collision Detection

Even though *collision detection* (CD) does not have anything to do with rigid body dynamics it is a vital aspect of the simulation. According to[Mir98], collision detection and resting contact simulation are the two bottle-necks in rigid body dynamics simulation. Therefore it is very important to use an as efficient and accurate CD routine as possible when simulating rigid body dynamics. Collision detection is part of what is often referred to as collision handling, which can be divided into three major parts: collision detection, collision determination and collision response. The result of collision detection is a boolean saying whether two or more objects collide, while collision determination finds the actual intersection between objects; finally, collision response determines what actions should be taken in response to the collision.

Most of the collision detection algorithms between polyhedrals is specified by a boundary representation. More sophisticated algorithms cache witnesses that are used to verify disjointness of penetration in constant time.

Collision detection has a wide variety of uses:

- Robotics
- CAD design
- Medicine research
- Games, demos, animation movies
- Scientific simulation

There are four different classes of collision detection. Bounding volume is described more thoroughly in the next chapters. Spatial subdivision include some techniques as BSP[FKN80] and octrees[ABJN85].

BSP trees is very popular within games since it is very fast can remove triangles based on the position and direction of the viewer.

Feature based collision detection is also called closest feature tracking. The algorithms are tracking the two closest geometric sizes. These algorithms are very effective and fast. Lin-Canny[Lin94] was the first that created these algorithms. Later, Brian Mirtich created V-Clip[Mir98] which is a further development of the work of Lin-Canny.

Simplex based collision detection try to compute the minimum distance between convex polyhedra. If the minimum distance is found, and the acceleration and velocity is given the distance can be used to estimate a lower bound on the time of impact. The most famous of these algorithms is GJK[GJK88].

4.1 Bounding Volume

To provide a simple intersection test and make more efficient rejections a *bounding volume* (BV) is frequently used. BV is a closed volume that contains a set of objects/polygons, it is often built as a hierarchy of bounding volumes. There are four bounding volumes that are commonly used; the sphere hierarchies[Hub96], the *axis-aligned bounding box* (AABB)[van97], the discrete oriented polytype (k-DOP)[KHM+98], and the *oriented bounding box* (OBB)[GLM96].

The reason for several bounding volumes is that they have different attributes. OBB and k-DOP have a tighter fit than sphere and AABB bounding volumes, but they have a more costly detection/rejection test.

4.2 Cost Function

There is a function t, that give a description of the performance of a collision detection algorithm. It was first used to evaluate the performance of CD algorithms by Gottschalk et al.[GLM96]

$t = n_v c_v + n_p c_p + n_c c_u$

- n_v : number of BV/BV overlap tests
- c_v : cost for a BV/BV overlap test
- n_p : number of primitive pairs tested for overlap
- c_p : cost for determine whether two primitives overlap

- n_u : number of BVs updated due to the model's motion
- c_u : cost for updating a BV

Choice of Bounding Volume:

- It should fit the original model as tightly as possible (to lower n_v and n_p)
- Testing two volumes for overlap should be as fast as possible to lower c_v

Primitives like spheres and AABBs do very well with respect to the second constraint, but they have a poor fit with some primitives like long-thin oriented polygons. OBBs and k-DOP provide tight fits, but checking for overlap between them is relatively expensive.

Hierarchal Decomposition

There is no hierarchal representation that gives the best performance all the time. When two objects are far apart, hierarchal representation based on spheres and AABBs work well. But, when two models are in close proximity with multiple number of closest features, the number of pair-wise bounding volume tests, n_v increases, sometimes also leading to an increase in the number pair-wise primitive contact tests, n_p. In this case a OBBTree will provide a smaller n_v and n_p. With an improved algorithm to check for overlap provided by Gottschalk et al.[GLM96] the cost is less than an order more costly than compared to sphere and AABB trees.

4.3 Separating Axis

The *separating axis theorem*[Got96] is heavily used for fast rejection tests for convex, disjoint polyhedra. Two polyhedra A and B, there exists a separating axis where the projections of the polyhedra, which form intervals on the axis, are also disjoint. If A and B are disjoint, then they can be separated by an axis that is orthogonal to: a face of A, a face of B, or an edge from each polyhedron.

4.4 Axis Aligned Bounding Box

An axis aligned bounding box (AABB), is a box whose faces have normals that coincide with the standard basis axes. AABB is the simplest bounding volume to create. Take the minimum and maximum extents of the set of polygon vertices along each axis and the AABB is formed.

4.4.1 Building a AABB

Creating a AABB bounding volume is very simple. Take the minimum and maximum extents of the set of polygon vertices along each axis and the AABB is formed.

4.4.2 AABB Intersection Test

Since the AABB is aligned with the main axis directions there is sufficient to describe the volume with two points. A Simple test is described below:

```
tab[] = [x, y, z];
for(int i = 0; i < 3; i++) {
  if (a_min[i] > b_max[i] || b_min[i] > a_max[i])
    return(DISJOINT)
return(OVERLAP)
```

4.5 Oriented Bounding Box

An oriented bounding box (OBB) is a rectangular bounding box with an arbitrary orientation in space. It is an AABB that is arbitrary rotated to fit the volume as best as possible.

4.5.1 Building an OBB

Gottschalk et al.[GLM96] showed that a tight-fitting OBB enclosing an object can be found by computing an orientation from the triangles of the convex hull.

First we compute the convex hull of all the triangles in the object. If the vertices of the i'th triangle are the points $\mathbf{p}^i, \mathbf{q}^i$, and $\mathbf{r}^i$, then area of the i'th triangle in the convex hull is:

$$A^i = \frac{1}{2}\left|\left(\mathbf{p}^i - \mathbf{q}^i\right) \times \left(\mathbf{p}^i - \mathbf{r}^i\right)\right|$$

Let the surface area of the entire convex hull be denoted by:

$$A^H = \sum_i A^i$$

Let the centroid of the i'th convex hull triangle be denoted by:

$$\mathbf{c}^i = \frac{\mathbf{p}^i + \mathbf{q}^i + \mathbf{r}^i}{3}$$

Let the centroid of the convex hull, which is a weighted average and the triangle centroids (the weights are the areas of the triangles), be denoted by:

$$\mathbf{c}^H = \frac{\sum_i A^i \mathbf{c}^i}{\sum_i A^i} = \frac{\sum_i A^i \mathbf{c}^i}{A^H}$$

Now we can compute a 3×3 covariance matrix, C, whose eigenvectors are the direction vectors:

$$\mathbf{C}_j k = \sum_{i=1}^{n} \frac{A^i}{12A^H} \left(9\mathbf{c}_j^i \mathbf{c}_k^i + \mathbf{p}_j^i \mathbf{p}_k^i + \mathbf{q}_j^i \mathbf{q}_k^i + \mathbf{r}_j^i \mathbf{r}_k^i\right) - \mathbf{c}_j^H \mathbf{c}_k^H$$

After computing C the eigenvectors are computed and normalized. Then we project the points of the convex hull onto the eigenvectors to find the minimum and maximum along each direction. We then use this to calculate the center and half-length of the OBB.

4.5.2 OBB Intersection Detection

To check intersection between two OBBs, A and B, a fast algorithm introduced by Gottschalk [Got96] that uses the separation axis theorem, and is about an order faster than previous methods.

The test is done in the coordinate system formed by OBB A's center and axes. B is placed relative to A by rotation B and translation T. The half-dimensions (radii) of A and B are a_i, and b_i, where $i = 1, 2, 3$. The axes of A and B are the vectors $\mathbf{A}_i$ and $\mathbf{B}_i$, for $i = 1, 2, 3$. These will be referred as the 6 box axes.

According to the separating axis theorem, it is sufficient to find one axis that separates A and B to be sure that they are disjoint. Fifteen axes has to be tested: Three from the faces of A, three from the faces of B, and $3 \cdot 3 = 9$ from combinations of edges from A and B.

As a consequence of the orthonormality of the matrix $\mathbf{A} = (\mathbf{A}^1, \mathbf{A}^2, \mathbf{A}^3)$, the potential separating axes that should be orthogonal to the faces of

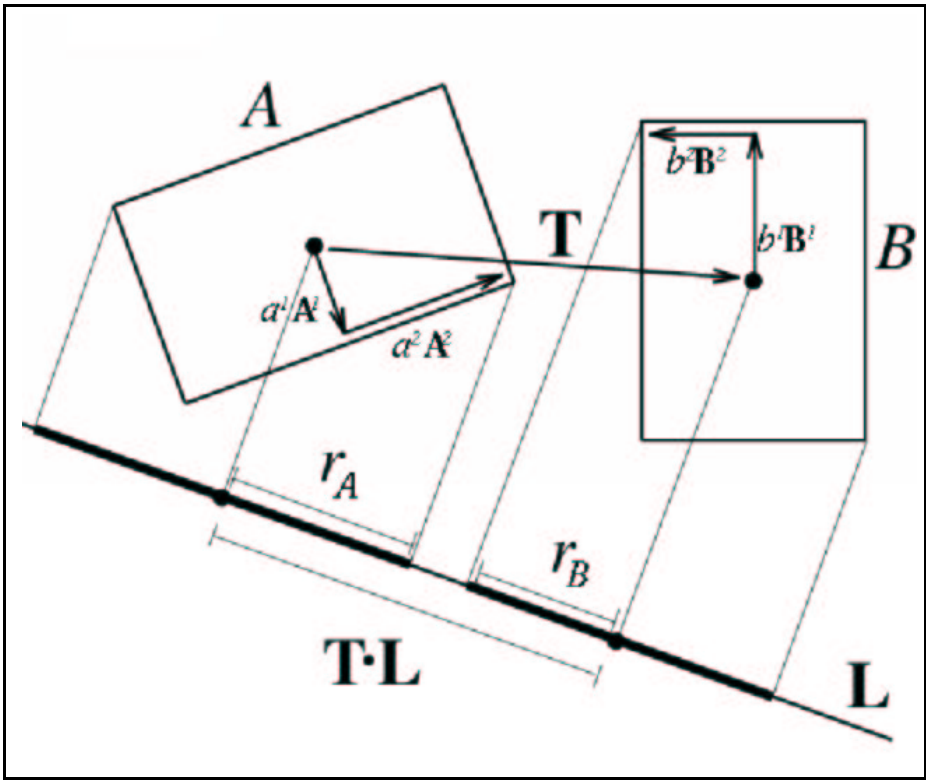

Figure 4.1: *The separation axis for two OBBs. They are disjoint since the projection of their radii on the axis determined by L are not overlapping. (Illustration after Gottschalk et al. [GLM96])*

A are simply the axes $\mathbf{A}^1, \mathbf{A}^2$, and $\mathbf{A}^3$. The same holds for B. The remaining nine potential axes formed by one edge each from both A and B, are then $\mathbf{c}^{ij} = \mathbf{a}^i \times \mathbf{b}^j$.

The centers of each box projects onto the midpoint of its interval. By projecting the box radii onto the axis, and summing the length of their images, we obtain the radius of the interval. If the axis is parallel to the unit vector $\mathbf{L}$, then the radius of box A's interval is:

$$r_A = \sum_i \left| a_i \mathbf{A}^i \cdot \mathbf{L} \right|$$

$$r_B = \sum_i \left| b_i \mathbf{B}^i \cdot \mathbf{L} \right|$$

The placement of the axis is immaterial, so we assume it passes through the center of box A. The distance between the midpoints of the intervals is $|\mathbf{T} \cdot \mathbf{L}|$. So, the intervals are disjoint if:

$$|\mathbf{T} \cdot \mathbf{L}| > \sum_i \left| a_i \mathbf{A}^i \cdot \mathbf{L} \right| + \sum_i \left| b_i \mathbf{B}^i \cdot \mathbf{L} \right|$$

The last step is due to the fact that the columns of the rotation matrix are also the axes of the frame of B. After simplifying all the terms, this axis test looks like:

$$|\mathbf{T}_3\mathbf{R}_{22} - \mathbf{T}_2\mathbf{R}_{32}| > a_2 |\mathbf{R}_{32}| + a_3 |\mathbf{R}_{22}| + b_1 |\mathbf{R}_{13}| + b_3 |\mathbf{R}_{11}|$$

4.6 OBBTree

OBBTrees was first presented at SIGGRAPH 96 by Gottschalk et al. [GLM96] with a paper called "OBB-Tree: A Hierarchal Structure for Rapid Interference Detection". This paper have strongly influenced further research on CD algorithms. As the name OBB-Tree algorithm implies, the bounding volume used is the oriented bounding box, the OBB. OBBs converge much faster to the underlying geometry they are holding than AABBs (axis-aligned bounding box) and spheres. OBBTrees have been widely used both in scientific simulations and in games.

4.7 Sweep and Prune

The sweep and prune technique was developed by M. Lin[Lin94] and exploits that objects undergo small changes in their position and orientation from frame to frame. Lin proved that the bounding box problem can be solved in $O\left(n \log^2 n + k\right)$ time (where k is the number of

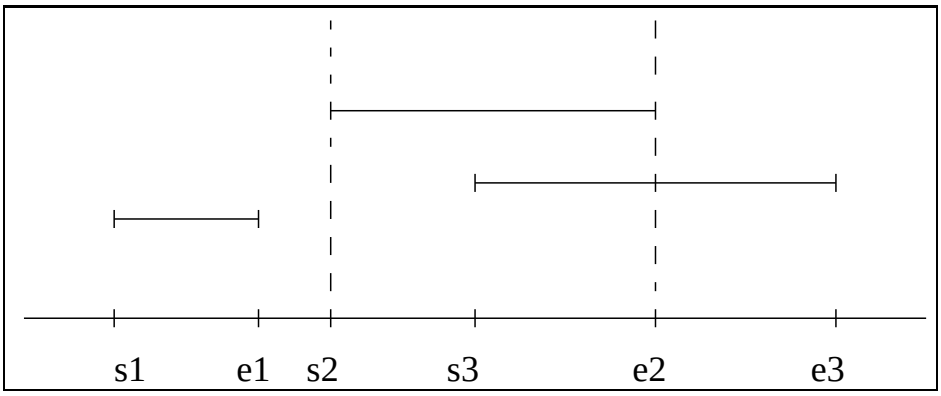

Figure 4.2: *Graphical view of Sweep-and-prune for one axis. When s3 is encountered s2 is already in the active list. There is an overlap between body 2 and 3 in at least one axis.*

pairwise overlaps), but it can be improved by exploiting coherence and so be reduced to $O(n+k)$.

If two AABBs overlap, then all three one-dimensional intervals in each axis direction must also overlap. Start and endpoints of the AABBs in each axis is stored in three lists, these values are stored in increasing order. This list is then swept from start to end. When a start-point is found it is stored in a active list. When its endpoint is found it is removed from the active list. If another start or endpoint is encountered when the start-point is in the active list we have overlap. A graphical example of how sweep and prune works is shown in figure 4.2.

This procedure would take $O(n \log n)$ to sort all the intervals, plus $O(n)$ to sweep the list, and $O(k)$ to report overlapping intervals. But since the lists are not expected to change very much from frame to frame, a bubble sort or insertion sort can be used with great efficiency after the first pass has taken place. It was shown in[SH76] that these sorting algorithms sort nearly-sorted lists in an expected time of $O(n)$.

4.8 Hierarchical Collision Detection

A hierarchy that is commonly used in the case of collision detection algorithms is a data structure called a k-ary tree, where each node may at most have k children. Most algorithms use $k = 2$, a binary tree. The root node is a BV that encloses the whole object. Each internal node enclose all of its children in its volume, and at each leaf, there are one

or more primitives.

There are three ways of creating a hierarchy: a $bottom - up$, $top - down$, and a $incrementaltree - insertion$.

The bottom-up method start with creating a BV for each primitive and then merge two and two nodes that are close in proximity together with a parent node until there are only one root node left.

The incremental tree-insertion method starts with an empty tree. Then all other primitives and their BVs are added one at a time to this tree.

The top-down approach is the method that is most frequently used. It starts by finding a BV for all the primitives to the object which is the root of the tree. Then a divide-and-conquer method is applied where the BVs is split into two (or k) parts. There is then created a BV based on the primitives in each part. This method is followed until there are only one primitive left in each part.

The big challenge is to find a satisfactory bounding volume and a hierarchy construction method that create balanced and efficient trees.

4.9 Collision Report

To make the collision detection routine useful there must be a way to get some confirmation that overlap is found. The most easy solution is to just return a boolean that say if there are any overlaps or not. This works for many situations, but since we want to use it in a rigid body simulation this is not sufficient. To simulate contacts in a rigid body simulation we minimum need the contact point and contact normal. This is more thoroughly explained in chapter 7.

To find the contact point and contact normal has an enormous negative impact on performance, but there is no way around it. Further discussion is found in the implementation section 5.2

Chapter 5

Collision Detection Implementation

When deciding which CD algorithm to use I wanted a better solution than the first one Trond Gaarder and I wrote in Jan 2002. That was an object specific algorithm which used simple geometry observations to check for collisions. It worked well for spheres and simple surfaces, but when adding cubes and thetrahedrons it took too much resources and was not usable.

A new implementation of a collision detection routine had to aviod the problems that were encountered in the previous attempt. A set of requirements were set to avoid earlier problems.

Requirements

- Generic collision detection algorithm. To be flexible, the CD algorithm can not be object specific. This means we should check for collisions on a triangle level.
- Fast. Since there will be a lot of computations regarding the rigid body simulation it is desirable that the CD algorithm use as little CPU time as possible.
- Not too complex. Since the timeframe of the thesis is limited, and the focus should mostly be on rigid body simulation it must be possible to implement it within a timeframe of 6-9 months.

BSPTrees [FKN80] was considered since its widely used especially in computer games. The problem with BSPTree is that it generates its collision tree from the user position and camera direction. This works

very well in FPS (first person shooter) games, but not when doing simulations.

Other techniques that caches withnesses like V-Clip[Mir98], was not chosen because these are generally more complicated than boundary representation techniques.

The decision fell on OBBTrees. The OBBTree design is clean and not too complex. The algorithm is widely accepted as one of the fastes around. It has been criticised to use too much memory, but since we would not generally simulate with models with more than +20K of triangles this was not a big issue.

The OBB algorithm performs very well when deciding overlap between objects that are close in proximity, but is quite costly to check for overlap on every object in the simulation. What we also want is an algorithm that quickly can determine if two boundary repersentations overlap.If they do, we can then further check for overlap with the OBB algorithm.

The Sweep-and-Prune algorithm is perfect for this, and is used just for this purpose in some CD packages. Sweep-and-Prune is easy to understand, but difficult to implement. Even so, the sweep-and-prune algorithm will speed up the collision detection dramatically so it should therefore be implemented.

Figure 5.1 show the outline of the collision detection routine and how it is integrated with the rigid body simulator. The collision routine was upon creation called *Java Axis-aligned Object-oriented Collision-detection* (JAOC).

5.1 Implementation of OBBTree

An OBBTree need tree nodes that can be used to rapidly check for overlap and either discard it if it is disjoint or call its children to finally check triangle-triangle if it overlap. The node need to specify the rotation and translation based on its parent and a size. If it is a leaf node it also has a pointer to the triangle it is bounding. If it is not a leaf node it has pointers to two child nodes. One in the positive and one in the negative direction based on its center.

To generate the tree a triangle array is given to the root node, it calculates the translation and rotation of the OBB box that contain all the triangles. When there are more than one triangle in the triangle array a recursive method that calculates the translation and rotation

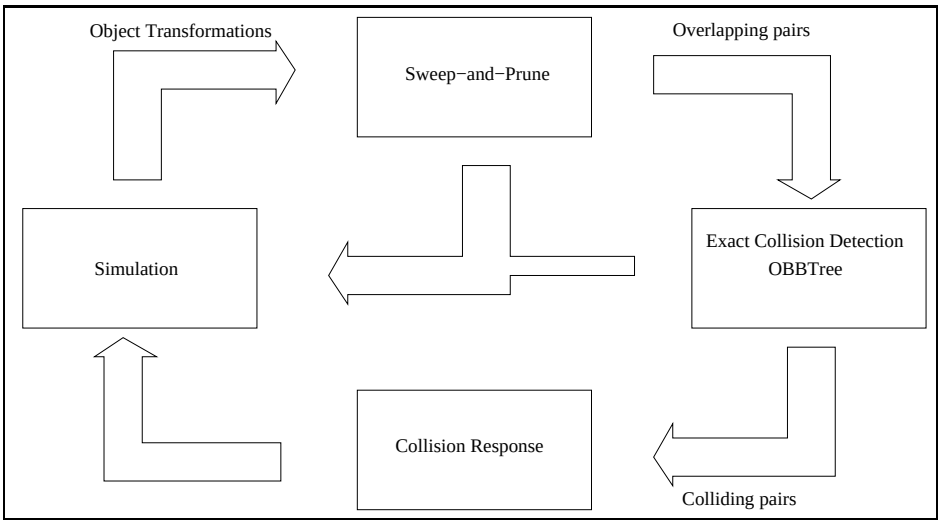

Figure 5.1: *Diagram of the collision detection system.*

of its children is called. The triangle array is divided in two new arrays with respect of the OBB node's center and the mean point to the triangles. The new arrays are now used when calculating the childrens attributes until the length of the array is 1. With this in mind we can create a OBB node object like this:

```
class OBBox {

  RBDMatrix3d rotation;
  RBDVector3d translation;
  RBDVector3d size;

  OBBox Pos;
  OBBox Neg;

  Triangle3d trp;
}
```

The Pos, Neg and trp objects are all references.

The calculation of translation and rotation is not an easy task (view the theory of generating a OBB tree). Several helper classes are needed, we use a class to help us calculate the areal and covariance matrix of a triangle. A class to accumulate all these values is also needed. A Triangle class is used for collision detection and generation of the tree.

To keep track of all the triangles and do collision detection we need a class to hold it all in place, a OBBTree class. The OBBTree contain all the geometric data of an object in 3d space. And has methods to check for contact with other OBBTrees. It must store the triangle array and manage the process of adding triangles to the tree. It also need to store the rotation and translation values of the whole object. Simplified outline of the OBBTree:

```
class OBBTree {

  OBBox root;
  Triangle3d tris[];

  RBDMatrix3d rotation;
  RBDVector3d translation;
}
```

The implementation is more complicated since we need to keep track of contact pairs, number of contacts, number of triangles, etc.

5.2 Collision Report Implementation

As mentioned we need the ability to find the contact point and the contact normal. Most current implementations only return a boolean indicating if two objects overlap. There are some implementations that find the triangles that overlap, but no overlap point. There are two reasons for this, it is difficult to implement and it has a huge performance impact.

One of the requirements of JAOC is that it should not only be used in rigid body dynamics, but also in other settings.

With this in mind the OBB collision detection routine was implemented with two methods that do almost do the same thing, one that return a boolean that indicates if contact is detected and one that find which triangles that overlap. The other method have a class called JAOCContact as a parameter. The contact point and contact normal is stored in this class. JAOCContact is shown here:

```
public class JAOCContact {

  private int bodyA;              // body containing vertex point
  private int bodyB;              // body containing face
  private RBDVector3d contact;    // contact point in world coord
  private RBDVector3d normal;     // contact normal
```

```
    private RBDVector3d edgeA;      // direction of A
    private RBDVector3d edgeB;      // direction of B
    private boolean vf;             // true if vertex/face contact
}
```

A contact has a reference to the bodies and a contact point. This point is given in world coordinates. A contact normal is also calculated, if the contact is a vertex/face contact, the contact normal is the normal of the face. If it is a edge/edge contact, the normal is given as the cross product of the two contact edges. If we have a vertex/face contact, the body that has the contact face is referenced by body B, the vertex point by body A. This make it easier to calculate an apply the impulse on the two bodies. The JAOCContact class stores the values we need to calculate the contact forces between the bodies. In chapter 6 the problem of calculating and appling the forces to the bodies is discussed.

5.3 Implementation of Sweep and Prune

The sweep and prune routine has references to all the objects in the simulation. When a object is loaded, the triangles are added to a AABB object which again has a reference to a OBBTree.

The AABB generates the tree upon initialization and is added to a AABB manager. The AABB manager has a list of all the AABB objects and a three dimensional list where the end and start points of the AABB in each direction are stored. Before the simulation starts the list is sorted. During simulation the three dimensional list is sweept and prunes as described earlier. When an overlap in all three dimensions is found the AABB collision pair more accurately check for overlap with the OBB routine.

If an overlap is found, it is reported to a list of overlapping pairs with object identifiers of the AABB's and the collision point. The list of overlapping pairs are later used by the physics routine to calculate the collision forces and reactions.

```
public class JAOCManager {

    // pointers to the three linked lists, one for each axis
    private EndPoint sweepList[];

    // AABBObjects pointers to a dynamic array of pointers.
    private JAOCObject AABBObjects[];
```

```
    // overlappingPairs contain all the pair of
    // AABBObjects that overlap. This list is used
    // by the OBB collision detection routine.
    public JAOCAABBCollisionReport overlappingPairs;

    // all contacts reported by the OBB routine
    public JAOCContactManager contacts;
}
```

The variable sweepList is a three linked list that contain the start and end point of the AABB's in each axis. It is this list that is sweept for each timestep to check for overlap. All overlaps are added to JAOCAABBCollisionReport object. The overlapping objects are kept in the list until the routine report them not to be overlapping. It is caching the overlaps from previous timestep, so already reported overlaps are not reported, only new and previously overlapping pair that do not longer overlap.

For each timestep the list of overlapping pairs is more accurately checked for overlap by the OBB routine. All overlapping pairs that are found by the OBB routine are added to the JAOCContactManager object that goes through the list and calculate the new constraints. Chapter 7 Contact, describes how contacts are handled in rigid body dynamics.

5.4 Performance

JAOC was written to be used in rigid body dynamics, but it can be used in anything from robotics, CAD design and games. The only requirements are that the objects must be specified by triangles and triangle points do not change position with respect to another during the simulation.

JAOC have different collision detection options buildt into it. It has two different methods of finding contacts. It can find the exact contact point and contact normal, or find the overlapping triangles without any specified contact point.

JAOC can also be set to only find the first overlap between two objects. This function can be used with the exact collision detection method or the overlapping triangle method. These options have a big impact on performance. The fastest is when JAOC is set to only find the first contact and the overlapping triangles. The slowest is when we want to find all contacts, their contact point and their contact normal.

It is difficult to measure the performance of collision detection routines and implementations since the routines are written with different designs and purposes.

Some different designs and purposes are listed here:

- Handle complex topology
- Highly accurate collision detection
- Handle many objects
- Handle deformation of objects during simulation

All these different designs make it difficult to compare different collision detection routines. The best way to test performance is to test it in a setting which would be representative for what the routine were built for.

5.4.1 Demo Test

The demo test was run on a Pentium 4, 2.4Ghz with 768mb RAM, running Debian Linux. All tests used the latest version of Java SDK, 1.4.2. The demo program test two torus objects which move through eachother. Each torus has 5000 triangles. All different options regarding contact report and number of contacts were tested.

- Finding all overlapping triangles: Average 8-10 overlaps found for each millisecond.
- Finding all overlapping triangles and the exact contact point and normal: Average 7-9 overlaps found for each millisecond.
- Finding the first overlapping triangle: Average time is 1-3 milliseconds.
- Finding the first overlapping triangle and the exact contact point and normal: Average time is 1-3 milliseconds.

When large objects are totally overlapping or close to a complete overlap there are very many reported contacts and the routine has to check almost every leaf node for overlap. This reduces JAOC's performance since checking all leaf nodes for overlap is costly. Since there are no interpenetration during rigid body simulation this is not a big problem. When large objects are in close proximity without contact or

when in contact, the JAOC performes very well. However there is a flipside, the demo test use 174MB of memory. This is more than expected, but it must be noted that the test was using double precision numbers. When using single precision numbers the memory footprint would be much smaller.

However the OBB routine is known to use a lot of memory. Each node stores the translation, rotation and size of the bounding volume. The node also have pointers to two child nodes and each leaf node has a pointer to a triangle. When we have N triangles there are $2 \times N - 1$ nodes in the OBBTree. This is the main reason of the high memory usage, but there are some solutions to this problem.

5.4.2 Optimization

During the implementation of JAOC several new ideas of improvements have emerged, the best and probably most memory efficient idea is described here:

Each leaf node in the OBBTree object is a OBB node which has a reference to a triangle. During the recursive collision queries, when checking two leaf nodes for overlap, the pair of bounding volumes is checked for overlap. Then, only if the OBB overlap, the triangles are checked for exact intersection. The idea is to simply skip the the first bounding volume test, and directly perform the triangle overlap test instead. A triangle-triangle overlap test is almost as fast as a OBB-OBB overlap test. Skipping the OBB-OBB test for leaf nodes should be efficient because:

- If the triangles overlap, we would have to perform the triangle overlap test anyway. In this case the OBB-OBB test prior to the triangle-triangle test can be skipped.
- If the triangle do not overlap the the test is roughly as fast as the OBB-OBB overlap test. There is also the possibility that the OBB-OBB test report overlap but the triangle-triangle overlap test would show that there are no overlap.

Does this mean that OBB nodes are not needed in leaf nodes anymore? No, because we still may have to collide a leaf node against a internal one. This mean that we need a triangle-OBB overlap test. Assuming we have this routine, we can remove the leaf nodes altogether

by storing the triangles in the parent nodes, replacing previous pointers to the leaf nodes we just discarded.

This reduces memory consumption considerably, instead of having $2 \times N - 1$ nodes, we now have $N - 1$ nodes. Moreover, replacing the OBB-OBB test with a more accurate triangle-OBB test actually leads to less tests since the OBB-OBB can report overlap wheras the triangle-OBB test does not.

The required triangle-box test has been derived by Tomas Akenine-Möller[AM01], and turns out to be roughly as fast as the standard OBB-OBB test.

With these improvements on the existing implementation it would use much less memory and be somewhat faster.

Another less drastic memory improvement would be to use quaternions instead of a 3x3 matrix to represent rotation of the OBB node. Using quaternions results in substancial memory savings, but need 13 more operations for each OBB overlap test. This is the usual trade of between memory and CPU time.

5.4.3 Conclusion

JAOC performes very well with large and small objects, objects that are in close proximity and objects that are far away. It has different methods of reporting contacts which makes it usable by many different applications. The problem is that the memory usage is high. But with the implementation of the ideas above the memory usage would be drastically reduced.

Chapter 6

Rigid Body Library

The development of the rigid body library has undergone several phases and changes. The requirements were set very early, and have not changed much during the development.

Requirements

- Use a modular architecture
- The architecture should be flexible and easily reusable. The basic components should be easily usable for different rendering engines.
- It should be suitable for interactive animations. This means speed is important, but not of the cost of extendibility and the modular architecture (see 1,2). This means accuracy may be sacrificed in exchange for more speed (if necessary).
- The dynamics system should not be purely impulse-driven, but support forces on the bodies.
- The numerical solvers used to solve the differential equations must prevent numerical drift.
- Implementation of collision detection and handling.
- Model friction.
- User manual for the library
- Examples of possible applications/demos/games using the library

6.1 Implementation

One of the goals of this library was that is should be easily reusable and extendable. The libraries have been written in an object oriented way. It could have been written without objects, but a modular and easily extensible library was one of the goals of this project. The library has been separated into three parts; Math, collision detection and rigid body simulation.

Math Library

The math library extends the existent vecmath library from SUN which is a part of Java3D[Mic97]. The vecmath library was chosen so our application could easily use Java3D as a visualization platform. The vecmath library is provided so that users who do not want to use the library against Java3D could easily do so. The math library is small with the classes RBDVector3*, RBDMatrix3* and RBDQuat4*. There are double and single precision version of the classes. Utility methods for finding eigenvectors and eigenvalues from a matrix[PFTV92] and quaternion to matrix calculation are provided to name a few.

Rigid Body Library

The rigid body library is responsible for the physics and the properties for all objects in the simulation. It has an integrated ODE solver which updates position and orientation for each timestep. The ODE solver could be implemented as a module of its own, but with the ODE integrated the API could still be clean and it is more efficient. A diagram of the rigid body library can be seen in figure 6.1.

Collision Detection Library

The collision detection library is JAOC. JAOC is responsible for detecting and reporting contacts. An outline of the library can be found in figure 6.2.

The contact manager works as a connection between JAOC and the rigid body module. The issue is where do we apply the contact forces on the objects? It can be implemented in both JAOC and the rigid body module without much problem. The contact manager is used by both but it is an integrated part in the JAOC routine. The current

6.1 Implementation

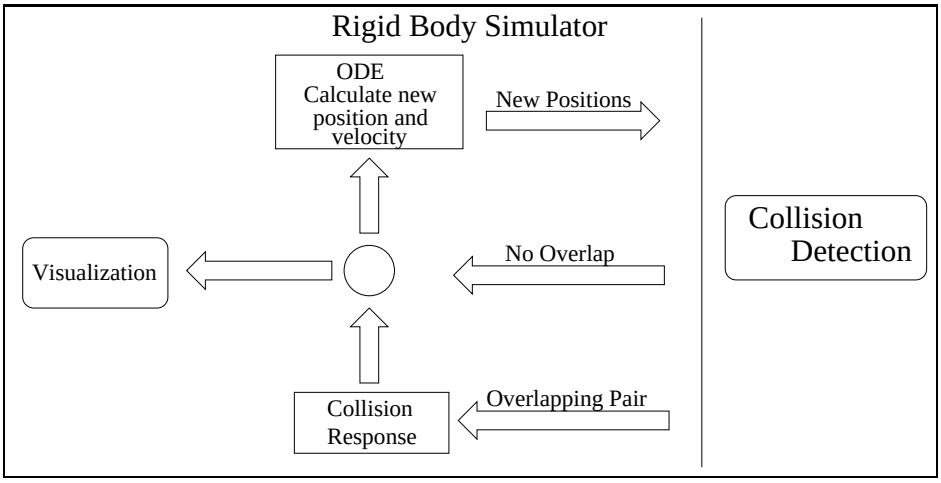

Figure 6.1: *Diagram of Rigid body simulator*

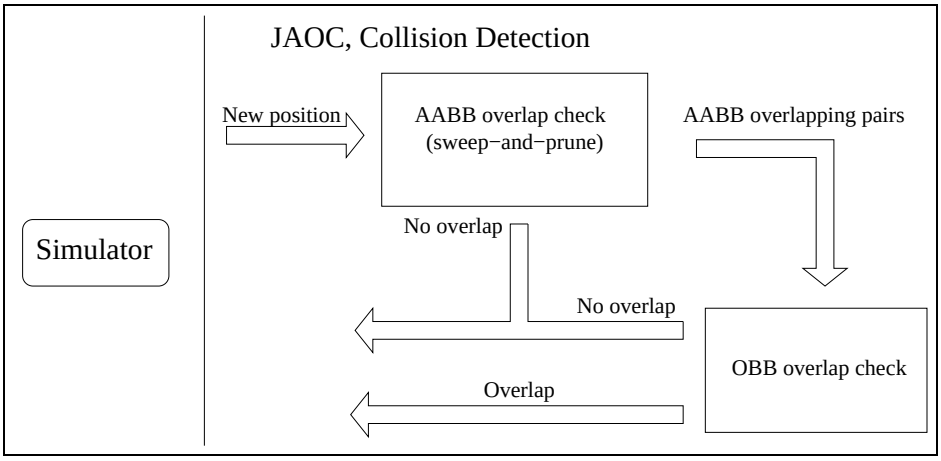

Figure 6.2: *Diagram of JAOC*

solution is to let the contact manager manage all the physics involving the impulse calculations.

Visualization

Java3D is used to visualize the objects, but it is not part of the rigid body library nor JAOC. The library is independent of visualization techniques and can easily be used by Java3D and GL4Java[Goe01].

6.1.1 Rigid Body Library

Each body is an object entity in world space and know of nothing other than itself. Ideally it should not need to contain any geometric data, just data that is needed to calculate the physical constraints. With the use of JAOC as the collision detection library the geometry can be stored in the JAOCObjects and not in the rigid body representation. The basis of a rigid body:

```
public class RigidBody3D {

  // The world coordinate position of the body
  private MyVector3f position;

  // The linear velocity of the body
  protected MyVector3f velocity;

  // Inverse Mass of the body
  private float invMass;

  // The orientation of the body
  private MyQuat4f orientation;

  // The angular velocity of the body
  private MyVector3f omega;

  // The inverse inertia tensor of the body in body coordinates
  private MyMatrix3f inertiaInv;

  // Velocity in body coordinates, used when calculating drag etc
  private MyVector3f bodyVelocity;

  // Linear moment
  private MyVector3f P;
```

```
  // Angular moment
  private MyVector3f L;

  // Restitution of body,
  // say how much energy is _lost_ during collision
  private float RESTITUTION;
}
```

The values of the rigid body object has been explained earlier in this thesis, so it is not covered here. The inertia tensor is not calculated by the object and must be calculated before object creation. A static utility class is provided to easily calculate the inertia tensor for most common geometric objects. The rigid body object stores the inverse inertia tensor instead of the tensor itself. All calculations involving the inertia tensor use the inverse. Therefore there is no reason to store the actual inertia tensor.

Other initial conditions for each rigid body is specified by assigning values to mass, position, orientation, linear moment and angular moment.

Mass is also a value we use the inverse instead of the value itself. This is mainly because it gives us the possibility to create bodies with infinite mass, eg. bodies that are stationary in the simulation.

6.1.2 Collision Detection Library

Here we will assume that JAOC detects all contacts between bodies and reports it in a decent way. All contacts are stored in a list in the JAOCContactManager class. A contact is stored in the JAOCContact class. JAOCContact:

```
class JAOCContact {

  private int bodyA;              // body containing vertex point
  private int bodyB;              // body containing face
  private RBDVector3d contact;    // in world coordinates;
  private RBDVector3d normal;     // contact normal
  private RBDVector3d edgeA;      // direction of A
  private RBDVector3d edgeB;      // direction of B
  private boolean vf;             // true if vertex/face contact
}
```

The JAOCContactManager store all JAOCContacts:

```
public class JAOCContactManager {

  private JAOCContact contacts[];
  private int numContacts;

  public JAOCContactManager() {
  }
  public void addContact(JAOCContact newContact) {
  }

  public JAOCContact getContact(int i) {
  }
}
```

The rigid body objects are stored in two different arrays. JAOCManager store the geometric values and RigidBodyManager store all the physic values. Now where should all the contacts be processed?

The JAOCContact object contain contact point and contact normal, but only the ids of the contact pair. It also need the rigid body objects. The contact processing could be implemented in the simulator, JAOC or even in a separate module. It was decided that the contact processing routine should be a part of the RigidBodyManager. For each timestep the simulator will give the RigidBodyManager the list of contacts to process from JAOCManager.

6.1.3 Rigid Body Manager

The rigid body manager controls all the bodies. It updates the timestep and calculate new velocities and positions with the use of an ODE. It is also responsible for the contact computation. A simple outline of the RigidBodyManager:

```
public class RigidBodyManager {

  private RigidBody3D bodies[];

  public RigidBodyManager() {
  }

  public void addBody(RigidBody3D body) {
  }

  public void processContacts(JAOCContactManager contacts) {
```

```
    }
}
```

To processContacts the JAOCContactManger object is given by the Simulator.

6.1.4 Simulator

The simulator is the main object in the simulation. It works as a connector between the collision manager and the rigid body manager. For each timestep it prune the contact manager for all contacts and send it to the rigid body manager that calculate the contact data.

All objects that is simulated has to be added to the simulator. The simulator pass the objects to the RigidBodyManager and JAOCManager.

```
public class Simulator {

  public RigidBodyManager rigidBody;

  public JAOCManager jaoc;

  public Simulator(boolean firstContact, boolean exactOverlap) {
  }

  public void newObject() {
  }
  // add physics data to the current object
  public void addRigidBody(RigidBody3D body) {
  }
  // add a triangle to the current object
  public void addTri(RBDVector3d v1, RBDVector3d v2, RBDVector3d v3,
                     int triId) {
  }

  public void update() {
  }
}
```

The booleans given to the constructor is further given to the JAOCManager constructor and set what kind of collision detection that will be used.

The update function updates the timestep and calls the RigidBodyManager which updates all the state variable of the rigid bodies. Then

the new positions and orientations is given to the JAOCManager and it checks for overlap. If any overlaps is found the JAOCContactManager object is given to the RigidBodyManager which processes all the contacts.

6.2 Example of use

This is a small example of how the library could be used to perform a rigid body simulation.

```
import org.rigidbodylibrary.math.*;
import org.rigidbodylibrary.jaoc.*;
import org.rigidbodylibrary.rbd.*;

public class SimulatorTest {

  public static void main(String[] args) {

    // create the simulator object
    Simulator sim = new Simulator(false, true);

    //use a rigid body util object to create a box and
    //add it to the simulator.
    BodyCreatorUtil bodyUtil = new BodyCreatorUtil();
    // create a box with width = height = length = 1 and mass 10.
    // Its position in world coordinate system is 2,0,0.
    int id1 = bodyUtil.newBox(1, 1, 1, 10,
                              new RBDVector3d(2, 0, 0), sim);
    int id2 = bodyUtil.newBox(1, 1, 1, 10,
                              new RBDVector3d(0, 0, 0), sim);
    // give one of the bodies a litte power
    sim.getBody(id2).addForces(
        new RBDVector3f( 2.0f, 0.0f, 0.0f), 1.0f);
    // then give the body a little spin
    RBDVector3f o = new RBDVector3f(-1.0f, -2.0f, 0.0f);
    o.scale((float)Math.sin(0.5));
    o.normalize();
    sim.getBody(id2).setOrientation(
        new RBDQuat4f(o.x, o.y, o.z, (float)Math.cos(0.5)) );

    // do the simulation
    while(true) {
      sim.update();
```

```
        }

    }
}
```

With utility classes as BodyCreatorUtil it is very easy to add bodies to the simulation. However it is not difficult to create a custom object. Only pass the triangles to the simulation manually with the addTri method.

The example above do not handle graphics, but that does not mean it is difficult to implement it. In the loop that call the update function, the position and orientation of the objects can be retrieved and used by the graphics objects. The objects can be created by the triangles added to the JAOCManager.

6.3 Summary

With a generic collision detection routine and proper contact handling implemented it is much easier to simulate rigid body dynamics. Since all bodies are objects entities it is very easy to add them to the simulation.

Chapter 7

Contact

In chapter 2 we introduced the equation of motion for a rigid body. When bodies are in contact we need to prevent them from inter-penetrate.

Lets consider a situation where a cube is falling onto a fixed floor. Since we are dealing with rigid bodies that are non-flexible, we do not want any inter-penetration at all. This means that at the instant the cube comes in contact with the floor, we must change the velocity of the cube.

Since we treat the bodies as totally rigid, the velocity has to be halted instantaneously to avoid inter-penetration.

This means we have two types of contact to deal with. When two bodies are in contact at some point p, and they have a velocity towards each other, this is called $colliding\ contact$. Colliding contact requires an instantaneously change in velocity. When two bodies are in contact at some point p, but the velocity between them are zero we say that the bodies are in $resting\ contact$.

In this chapter we will only look at what we do when we have found and reported a possible collision. A more in depth look at collision detection algorithms and some implementations are found in chapter 4, Collision Detection.

7.1 Colliding Contact

Contacts between polyhedral is either $vertex/face$ contacts or $edge/edge$ contacts. A vertex/face contact is when a vertex on one polyhedral is in contact with a face on another polyhedra. An edge/edge contact is when two edges contact; it is assumed that the two edges are not collinear.

For now, we will assume that the contact is frictionless and that the line of action of the impulse is normal to the surface of both objects. When we have a vertex/face contact, we use the normal to the face as the contact normal. When we have a edge/edge contact, we use the unit vector for each edge and compute the cross product between them, and use this vector as the contact normal.

Lets say we have got a contact point p, from the collision detection algorithm. The contact point is given in world space. We have two bodies A and B.

$\mathbf{p}_a(t)$ is the point on body A that satisfies $\mathbf{p}_a(t) = \mathbf{p}$. Similarly, we have the point $\mathbf{p}_b(t)$ for body B. Even though $\mathbf{p}_a(t)$ and $\mathbf{p}_b(t)$ are similar they do not have the same velocity. The velocity to $\mathbf{p}_a(t)$ is given by:

$$\dot{\mathbf{p}}_a(t) = \mathbf{v}_a(t) + \omega_a(t) \times (\mathbf{p}_a(t) - \mathbf{x}_a(t))$$

where $v_a(t)$ is the linear and $\omega_a(t)$ angular velocity for body A. $x_a(t)$ is the position of center of mass in world coordinates. Similar for B:

$$\dot{\mathbf{p}}_b(t) = \mathbf{v}_b(t) + \omega_b(t) \times (\mathbf{p}_b(t) - \mathbf{x}_b(t))$$

Now we need to calculate the relative velocity between body A and B. To get the relative velocity we use the contact normal:

$$v_{rel} = \mathbf{n}(t) \cdot (\dot{\mathbf{p}}_a(t) - \dot{\mathbf{p}}_b(t))$$

v_{rel} is a scalar and describes the velocity between the two objects. If v_{rel} is positive this means that the relative velocity at the contact point is in the positive $\mathbf{n}(t)$ direction and the bodies are moving apart. If the relative velocity is zero the bodies are neither colliding nor separating, they are resting. If the relative velocity is less than zero the bodies are colliding. If the relative velocity is less than zero we need to stop the bodies from penetrating.

The most obvious thing to do is to apply a force to both objects, but a force will not stop the bodies from penetrating because a force can not instantaneously change the velocity. Instead of using force we introduce a new quantity J, called an $impulse$ [MC95]. An impulse is a vector quantity, like a force, but it has its units of momentum. An impulse can be seen as a huge force integrated over a short period of time. We think of the time as infinitely small and the force almost infinitely large. As the force change the momentum over time, a impulse change the momentum instantaneously.

The collision model we will use is called "Newton's Law of Restitution for Instantaneous Collisions with No Friction." The impulse can be denoted as:

$$F\Delta t = J$$

Where $F \to \inf$ and $\Delta t \to 0$.

When two bodies collide, an impulse is applied between them to change their velocity. For frictionless bodies, the direction of the impulse will be in the normal direction, $\mathbf{n}(t)$. We can write the impulse J as:

$$\mathbf{J} = j\mathbf{n}(t)$$

Where j is an scalar that gives the magnitude of the impulse. We will adopt the convention that the impulse J acts positively on body A, $+jn(t)$, while body B is subject to: $-jn(t)$. Here the collision normal is the normal computed from the face of body B if it was an edge-vertex contact or with the cross product if it was an edge-edge collision.

Newton's Law of Restitution introduces yet another quantity, the coefficient of restitution, denoted by e or ϵ. ϵ must satisfy $0 \leq \epsilon \leq 1$. The coefficient of restitution tells us how much of the incoming energy is lost during the collision. If $\epsilon = 1$, no kinetic energy is lost. If $\epsilon = 0$ all kinetic energy is lost and the two bodies will be at resting contact at the contact point.

$$v_{rel}^{+} = -\epsilon v_{rel}^{-}$$

where v_{rel}^{+} is the relative velocity after the impulse has been applied, and v_{rel}^{-} is before the impulse has been applied.

We further denote v^{-} and ω^{-} as the velocities before the impulse has been applied and v^{+} and ω^{+} are the velocities after the impulse has been applied. We can now write the velocity after impulse for body A as:

$$\dot{\mathbf{p}}_a^{+}(t) = \mathbf{v}_a^{+}(t) + \omega_a^{+}(t) \times \mathbf{r}$$

Where $\mathbf{r} = \mathbf{p} - \mathbf{x}(t)$.

We can split up the two velocities to:

$$\mathbf{v}_a^{+}(t) = \mathbf{v}_a^{-} + \frac{j\mathbf{n}(t)}{M_a}$$

where M_a is the mass of body A.

$$\omega_a^{+} = \omega_a^{-}(t) + I_a^{-1}(t)\left(\mathbf{r}_a \times j\mathbf{n}(t)\right)$$

and I_a^{-1} is the inerta tensor of body A.

After solving for j we get:

$$j = \frac{-(1+\epsilon)\, v_{rel}^{-}}{\frac{1}{M_a} + \frac{1}{M_b} + \mathbf{n}(t) \cdot \left(I_a^{-1}(t)\left(\mathbf{r}_a \times \mathbf{n}(t)\right)\right) \times \mathbf{r}_a + \mathbf{n}(t) \cdot \left(I_b^{-1}(t)\left(\mathbf{r}_b \times \mathbf{n}(t)\right)\right) \times \mathbf{r}_b}$$

When we have calculated j we plug it in the equation and we have a collision response with the correct spin based on their incoming velocities and mass.

7.2 Resting Contact

Solving resting contact is hard because the only known methods of dealing with resting contact demands some sophisticated numerical software, known as quadratic solver.

Let us now say that we have n contact points, and all points are in resting contact. In other words the relative velocity v_{rel}, is zero or smaller than a numerical threshold. As with colliding contact, we have a contact force that acts normal to the contact surface. With colliding contact we had an impulse $jn(t)$ where j was an unknown scalar. For resting contact, there is some force $f_i n_i(t)$ at each contact point where f_i is an unknown scalar and $n_i(t)$ is the contact normal at the i'th contact point. What we want is to determine the value of all the f_i's. To determine the value they must all be calculated at the same time, since the force at the ith contact point may influence one or both of the bodies of the j contact point.

When calculating f_i we have three conditions. The contact forces must prevent inter-penetration, that is, the contact forces must be strong enough to prevent the two bodies from being pushed "towards" eachother. Second, we do not want the force to act like a "glue" and hold the bodies together. Third, we want the force at a contact point to become zero if the bodies begin to separate.

The first condition, prevent inter-penetration. For each contact point we use a expression $d_i(t)$ which describes the distance between two bodies at contact point i. If $d_i(t)$ is negative we have inter-penetration and if it is positive the bodies have broken apart. When the bodies are in contact we have $d_i(t) = 0$ (within numerical tolerances) for each contact point.

At a contact point i between two bodies A and B, we can construct a function for $d_i(t)$:

$$d_i(t) = n_i(t) \times (p_a(t) - p_b(t))$$

The same function can be used with edge/edge contacts since $n_i(t)$ points outwards from B towards A. At a contact point we have $d_i(t) = 0$, what we need it to keep $d_i(t)$ from decreasing at time t. This means we have $\dot{d}_i(t) > 0$.

$$\dot{d}_i(t) = n_i(t) \times (p_a(t) - p_b(t)) + n_i(t) \times (\dot{p}_a(t) - \dot{p}_b(t))$$

$\dot{d}_i(t)$ describes the separation velocity at time t. This is exactly what we called v_{rel} earlier. For resting contact we know that $\dot{d}_i(t)$ is zero, because the bodies are neither moving towards or away from eachother at the contact point. What we want to take a closer look at is $\ddot{d}_i(t)$. When we differentiate the equation above we get:

$$\ddot{d}_i(t) = \ddot{n}_i(t) \times (p_a(t) - p_b(t)) + 2\dot{n}_i(t) \times (\dot{p}_a(t) - \dot{p}_b(t)) + n_i(t) \times (\ddot{p}_a(t) - \ddot{p}_b(t))$$

Since $p_a(t) = p_b(t)$, we can write $\ddot{d}_i(t)$ as:

$$\ddot{d}_i(t) = n_i(t) \times (\ddot{p}_a(t) - \ddot{p}_b(t)) + 2\dot{n}_i(t) \times (\dot{p}_a(t) - \dot{p}_b(t))$$

The value of $\ddot{d}_i(t)$ describes how the two bodies are accelerating towards each other at the contact point. If $\ddot{d}_i(t)$ is positive the bodies are moving apart, if $\ddot{d}_i(t) = 0$ the contact remains. A negative $\ddot{d}_i(t)$ must be avoided. We have now the constraint

$$\ddot{d}_i(t) \geq 0$$

for each contact point.

The second constraint is simply that f_i must be positive. This since a force $f_i n_i(t)$ acts on body A, and $n_i(t)$ is the outwards pointing normal of B.

The third say that f_i must be zero if contact is breaking. We can express this as

$$f_i \ddot{d}_i(t) = 0$$

To find all the f_i which satisfy the three conditions, we can express each $\ddot{d}_i(t)$ as a function of the unknown f_i. We can write each $\ddot{d}_i(t)$ as

$$\ddot{d}_i(t) = a_{i1} f_1 + a_{i2} f_2 + \cdots + a_{in} f_n + b_i$$

We can write this with matrix syntax

$$\begin{pmatrix} \ddot{d}_1(t) \\ \vdots \\ \ddot{d}_n(t) \end{pmatrix} = \mathbf{A} \begin{pmatrix} f_1 \\ \vdots \\ f_n \end{pmatrix} + \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

where **A** is the $n \times n$ matrix of the a_{ij} coefficients.

$$\sum_{i=1}^{n} f_{n_i} i(t) \ddot{d}_{n_i}(t) = 0$$

can now be written as

$$f^T a = 0$$

We can now write the linear relation as:

$$a = \mathbf{A}f + b$$

We can now write the three constraints as:

$$\begin{aligned} \mathbf{A}f + b &\geq 0 \\ f &\geq 0 \\ f^T(\mathbf{A}f + b) &= 0 \end{aligned} \tag{7.1}$$

This is a LCP (Linear Complemtary Problem)[Bar94]. Since we ignore friction the matrix A is a PSD matrix. This means that there is a solution to the LCP. The problem is that LCP is NP hard. This means that the more contacts we have the solving time will increase exceptionally.

7.3 Implementation of Colliding Contact

When we have colliding contact, we are given several values from the collision detection routine. We have a reference to the two colliding objects and the contact point and contact normal. First we need do check if the the relative velocity v_{rel} between the bodies is zero or smaller than a numerical threshold. The body object have a method that calculate the velocity of a point in the body. A method to check for contact can then easily be written as:

```
public int calcRelativeVel() {
    relativeVelocity.sub(bodyA.getPointVelocity(collisionPoint),
                         bodyB.getPointVelocity(collisionPoint));
    int vrel = collisionNormal.dot(relativeVelocity);
```

```
    if (vrel > THRESHOLD)  // moving away
      return NO_COLLISION;

    else if (vrel > -THRESHOLD) // resting contact
      return COLLISION;         // treat it as a collision

    else
      return COLLISION;
  }
```

When a collision is confirmed by the relative velocity we need to find the impulse, $\mathbf{J} = j\mathbf{n}(t)$. We already have the contact normal, but we need to calculate j. j is defined to be:

$$j = \frac{-(1+\epsilon)\, v_{rel}^{-}}{\frac{1}{M_a} + \frac{1}{M_b} + \mathbf{n}(t) \cdot \left(I_a^{-1}(t)\left(\mathbf{r}_a \times \mathbf{n}(t)\right)\right) \times \mathbf{r}_a + \mathbf{n}(t) \cdot \left(I_b^{-1}(t)\left(\mathbf{r}_b \times \mathbf{n}(t)\right)\right) \times \mathbf{r}_b}$$

Calculation of j is, though lengthy, quite easy and will not be shown here, but rather a the method used to apply the impulses on the bodies as:

```
public void applyImpulseNoFriction() {
  MyVector3d J = new MyVector3d(getCollisionNormal(i));

  double j = getImpulse();        // getImpulse() calculates j
  J.scale(j);                     // J = n * j;

  bodyA.applyImpulse( J );        // positive impulse on body A
  J.negate();
  bodyB.applyImpulse( J );        // negative impulse on body B
}
```

Since the collision normal is calculated from body B, the positive impulse acts on body A, and the negative acts on body B. Then we apply the impulse on the body:

```
public void applyImpulse(MyVector3d impulse ) {
  // Apply the impulse to the body
  P.add(impulse);                              // P += impulse
  MyVector3d tmpVec = new MyVector3d();
  tmpVec.cross(collisionPoint, impulse);       // L += collPoint X impulse
  L.add(tmpVec);

  // recompute the auxiliary variables
  velocity.scale(getInverseMass(), P);         // v = P / mass
```

```
    omega.mul(getWorldInertiaInverse(), L);   // omega = Iinv * L
}
```

When the impulse is added the collision response routine is finished and the control is given to the simulator.

Chapter 8

Java

In this section we will briefly look at the programming language Java, and give an overview of the concepts involved.

8.1 Overview

Java is a relatively new programming language. It was first released in 95 by SUN Microsystems and is still developed further. Traditional Java applications are applications that are written in Java and then compiled into byte code form - called Java byte code. This byte code can be executed by a Java virtual machine (JVM) on any platform that Java supports. This means that the byte code is completely platform independent, and the JVM protects the underlying machine from illegal instructions and memory access. Java is an imperative object oriented programming language. Its syntax is very similar to C++. The details of Java can be found in the Java language report [GJS00].

8.2 Java Development Kit

The Java development kit (JDK) is the name of the official Sun toolkit to produce Java applications. It consists primarily of a compiler, java interpreter, core API, and the full documentation of the Java core API (javadoc). The latest version of JDK is version 1.4.1 which was released in the end of 2002. The official JVM follow the same version numbers.

8.3 Java Virtual Machine

With each version the speed of Java applications has improved significantly. Originally JVMs were simply byte code interpreters that executed one command after another. With JDK 1.1, Just-in-Time (JIT) compilation was introduced to the virtual machines. These compiled each class to machine code before the first time it was executed. This should have caused Java programs to become as fast as any statically compiled program.

Unfortunately, this was not the case, since traditional compiler technology relies heavily on inlining to gain their high performance. This was not easy in Java because each class in byte code form is completely independent of other classes. This flexibility is enhanced even more by the possibility of dynamically loaded classes. Since each class in JDK 1.1 and JDK 1.2 was compiled to machine code one at the time the consequence was that it was impossible to inline method across class boundaries or even within classes themselves if those methods were not private.

Several approaches have been used to solve this problem.

- Assume that all classes immediately presented are static and compile the classes as they were written in any other statically compiled language. This is the approach used by the Java static compilers.
- Allow the compiled machine code to be recompiled at run-time if the involved set of classes changes. This is the approach used by Hotspot virtual machines (JDK 1.3 and later).

8.4 Java Hotspot

One of the major innovations on Java is the development of Hotspot technology (JDK 1.3 and later), a kind of adaptive optimizer technology. [Gos01]

The idea behind this technology is that the application is analyzed at run-time and optimizations are performed based on these results.

The first times some piece of code is run, it is run interpreted, and when the system has gathered enough information about the run-time behavior of the application it aggressively compiles the sections of code that are run the most; the hotspots.

The idea is that it is better to use much time compiling the code that is run frequently rather than performing inferior compilation to it all as is usually done by JIT compilers.

This strategy introduces an initial overhead to the application as it start up (the so- called warm up period) while it analyzes run-time behavior and runs the optimizing compiler.

The optimizations that can be performed on the code can be much more aggressive than the optimizations performed with traditional static compilers, such as C++ compilers, because the compilation can be undone and redone at any time. This means that if the general run-time behavior changes then the machine code can, and probably will also change.

In general, the Hotspot virtual machine first attempts to in-line as much code as possible. Then the full range of traditional optimization techniques is applied to inlined code. Special optimizations for Java are:

- Array bounds check elimination.
- Elimination of checked type casts.
- Non-final methods (virtual methods) can be turned non-virtual if they are not derived or never used polymorphically.

8.5 Java Performance

From being just a simple byte code interpreter to the use of hotspot technology, Java performance has increased dramatically. The first version of Java (JDK 1.0), was approximately 20-40 times slower that C++. JDK 1.3 had a factor 0.7 - 4 slower that C++, and JDK 1.4 is from 0.5-3 slower than C++. This is based on tests[Mar02] where both the C++ and the Java code were tweaked as much as possible. Further, if the program uses 3D hardware, as most games today do, most of the execution time is spent on 3D hardware, which means the performance difference between Java and C++ becomes even less.

8.6 Java3D

8.6.1 Background

Java3D, a relatively new 3D API, tries to learn from the existing 3D graphics interfaces. It sits somewhere in the middle of the low level APIs like OpenGL and high level like VRML [VRM97].
Java3D provides a structured scene graph approach to representing objects in 3D space. On top of this, it provides a system for building custom behaviours that act according to a variety of stimulus within the API. Java3D also provides a lot of glue capabilities, such as generic input device support and 3D sound.
Java3D exists as part of the Java Media APIs [JMA]. The various APIs in this framework provide the capabilities for integrating with other multimedia and Internet technologies.
In order to provide a decent level of performance, Java3D uses a lot of native code provided by the operation system libraries. On a Solaris or Linux based machine, this means making use of OpenGL rendering. MS Windows users may either use OpenGL or Direct3D. Native code use is restricted to only the final rendering steps while most of the core features are written in Java.

8.6.2 Future of Java3D

SUN finalized version 1.3 of Java3D in the summer of 2002. Version 1.3.1 was released 14 may 2003 which where bug fixes and performance enhancement.

According to Doug Twilleager a Java3D SUN developer, parallel with Java3D 1.3.1, Java3D 1.4 will be developed. Some of the new features that are currently being considered are:

- Programmable Shading (Vertex/Pixel Shaders)
- Stencil buffer support
- Limited Extensibility
- Render to texture

The JSR for Java3D 1.4 will be filed by Siggraph (July 21). According to Doug Twilleager they will work to provide as a compatible shader language with OpenGL2 and DirectX9 as they can.

8.6 Java3D

With version 1.4 Java3D will take a leap forward when dealing with the new graphics hardware possibilities that are only usable with OpenGl and DirectX at the moment.

The OpenGL2 specs also include native Java bindings.

Chapter 9

Conclusion

We have presented techniques for constructing a rigid body simulation system. We have described an implementation of a system that can be used for interactive, real-time simulation. We presented different methods of collision detection and an implementation of a flexible and powerful collision detection routine. The flexibility of the collision detection routine make it usable in many different fields. A library is provided that make it easy to create applications for simulating rigid body dynamics.

9.1 Progression, Difficult Subjects

When the first attempts at creating a object specific collision detection routine failed because of poor performance. It was decided that a more sophisticated collision detection routine was needed.
The implementation of JAOC was planned to take no more than 6-7 months. During development it became clear that the routine was more complex than expected and there were several technical problems that took much time to implement and debug. Some of the challenges were:

- Computation of the OBBTree.
- Calculate the rotation and translation of the OBB nodes
- Overlap check between OBB objects
- Finding exact collision point and normal

The OBB routine was implemented first, and by then we knew that it would be impossible to finish JAOC within the time frame we had

planned. The OBB routine worked and would be sufficient for our needs, but we felt that it was a half-done product and it would be more usable with the sweep-and-prune routine implemented. It was decided to finish JAOC as initially planned.

When JAOC was completed we had a collision detection routine that was fast, flexible and it could be usable in much more that rigid body dynamics. It also has an integrated collision handling routine which make it very easy to perform contact computation on the objects.

As a result of the time spent on JAOC there were less time spent on the physics part than planned. Resting contact and contact with friction are the two main subjects that were put aside.

9.2 Further Work

There are several directions in which this work could be built upon:

- Implementation of resting contact and contact modeling with friction.
  The theory of resting contact is provided in section 7.2 the main task would be to create a LCP solver. The implementation with the existing library would not be complicated.

- Implementation of contact with friction.
  David Baraff has written a paper[Bar94] where he introduces a method for solving the problem.

- Further development of the collision detection routine.
  Some methods of improving JAOC where provided in section 5.4.2. The exact collision detection routine should be rewritten. The algorithm for finding exact overlap is slow and it find many duplicate contact points.

Bibliography

[ABJN85] D. Ayalla, P. Brunet, R. Juan, and I. Navazo. Object Representation by means of Nonminimal Division Quadtrees and Octrees. *ACM Transactions of Graphics*, 4:41–59, 1985.

[AM01] Tomas Akenine-Möller. Fast 3D Triangle-Box Overlap Testing. *Journal of Graphics Tools: JGT*, 6(1):29–33, 2001.

[Bar94] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. *Computer Graphics*, 28(Annual Conference Series):23–34, 1994.

[BW97] David Baraff and Andrew Witkin. An Introduction to Physically Based Modeling: Rigid Body Simulation I - Unconstrained Rigid Body Dynamics. *Computer Graphics*, 1997.

[FKN80] H. Fuchs, Z.M. Kedem, and B.F. Naylor. On Visible Surface Generation by A Priori Tree Structures. *Computer Graphics (Proceedings of SIGGRAPH 80)*, pages 124–133, 1980.

[Gal01] P. Galli. Study - Java to overtake C/C++ in 2002. 2001.

[GJK88] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A Fast Procedure for Computing the Distance between Complex Objects in Three-Dimensional Space. *IEEE Journal of Robotics and Automation*, pages 193–203, 1988.

[GJS00] J. Gosling, B. Joy, and G. Steele. *The Java Languange Specification*. Addison-Wesley, second edition, 2000.

[GLM96] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A Hierarchical Structure for Rapid Interference Detection. *Computer Graphics*, 30(Annual Conference Series):171–180, 1996.

[Goe01] Sven Goethel. OpenGL for Java, GL4Java., 2001.

[Gos96] J. Gosling. The Java Languange - an overview. 1996.

[Gos01] J. Gosling. The Hotspot Technology. 2001.

[Got96] S. Gottschalk. Separating axis theorem. *Technical Report TR96-024, Dept. of Computer Science, UNC Chapel Hill*, 1996.

[Hub96] Phillip M. Hubbard. Approximating Polyhedra with Spheres for Time-critical Collision Detection. *ACM Transactions of Graphics*, 15, 1996.

[JMA] Java Media API.

[KHM+98] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient Collision Detection using Bounding Volume Hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4:21–36, 1998.

[Lin94] Ming Chieh Lin. Efficient Collision Detection for Animation and Robotics. Technical Report ERL-94-13, 1994.

[Mar02] Jacob Marner. Evaluating Java for Game Deveolpment. 2002.

[MC95] Brian Mirtich and John F. Canny. Impulse-Based Simulation of Rigid Bodies. In *Symposium on Interactive 3D Graphics*, pages 181–188, 217, 1995.

[Mic97] SUN Microsystems. Java 3D API Collateral, Technical White Paper, 1997.

[Mir96] Brian Mirtich. Fast and Accurate Computation of Polyhedral Mass Properties. *Journal of Graphics Tools: JGT*, 1(2):31–50, 1996.

[Mir98] B. Mirtich. Rigid Body Contact: Collision Detection to Force Computation, 1998.

[PFTV92] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipies in C: The Art of Scientific Computing*. Cambridge University Press, 1992.

[SH76] M. Shamos and D. Hoey. Geometric Intersection Problems. *Proceedings of the 17th Annual Conference on Foundations of Computer Science*, pages 208–215, 1976.

[van97] Gino van den Bergen. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools: JGT*, 2(4):1–14, 1997.

[VRM97] VRML Spesification, 1997.