Craft Physics Interface

Examensarbete utfört i bildkodning

av

Henrik Hansson

LiTH-ISY-EX--07/3887--SE
Linköping 2007
Craft Physics Interface

Examensarbete utfört i bildkodning
vid Linköpings tekniska högskola
av

Henrik Hansson

LiTH-ISY-EX--07/3887--SE

Handledare: Ingemar Ragnemalm, ISY
Micael Belin, Craft Animations

Examinator: Ingemar Ragnemalm
Linköping 15 februari 2007
Abstract

This is a masters thesis (20p) in computer science at the University of Linköping. This thesis will give an introduction to what a physics engine is and what it consist of. It will put some engines under the magnifying glass and test them in a couple of runtime tests. Two cutting edge commercial physics engines have been examined, trying to predict the future of physics engines. From the research and test results, an interface for physics engine independency has been implemented for a company called Craft Animations in Gothenburg, Sweden.

Nyckelord

Physics Engine, Havok, PhysX, ODE, Newton Game Dynamics, Physics Interface, Craft Animations
Abstract

This is a masters thesis (20p) in computer science at the University of Linköping. This thesis will give an introduction to what a physics engine is and what it consist of. It will put some engines under the magnifying glass and test them in a couple of runtime tests. Two cutting edge commercial physics engines have been examined, trying to predict the future of physics engines. From the research and testresults, an interface for physics engine independency has been implemented for a company called Craft Animations in Gothenburg, Sweden.
Table of contents

1 Introduction................................................................................................................. 1
  1.1 Background............................................................................................................. 1
  1.2 Purpose................................................................................................................... 1
  1.3 Problem formulation............................................................................................. 1
  1.4 Delimitations......................................................................................................... 2
  1.5 Method.................................................................................................................. 2
2 Theory....................................................................................................................... 3
  2.1 Basic terms used by a physics engine................................................................. 3
  2.2 What is a physics engine?..................................................................................... 5
3 Engines today.............................................................................................................. 15
  3.1 Overview............................................................................................................... 15
  3.2 A closer look.......................................................................................................... 18
  3.3 Runtime tests....................................................................................................... 24
  3.4 Two commercial physics engines......................................................................... 34
4 Engines of the future................................................................................................. 43
5 Design & Implementation............................................................................................ 45
  5.1 Introduction.......................................................................................................... 45
  5.2 Specification of requirements............................................................................. 46
  5.3 Implementation.................................................................................................... 46
  5.4 Requirement fulfilment....................................................................................... 47
  5.5 Conclusion............................................................................................................ 48
6 Conclusions................................................................................................................. 49
7 References................................................................................................................ 51
  7.1 Figures.................................................................................................................. 51
  7.2 Books................................................................................................................... 52
  7.3 Internet................................................................................................................ 53
  7.4 Papers & articles.................................................................................................. 55
8 Appendix A............................................................................................................... 57
  8.1 Design................................................................................................................... 57
9 Appendix B............................................................................................................... 67
1 Introduction

For several years, as discussed by David Bourg in [01], the computer gaming industry has primarily concentrated on developing stunning graphics and faster graphic cards to attract more customers. In games like Doom or Quake all monsters fell according to a specific pattern no matter where the player shot them. Game developers today are trying more and more to create games that simulate correct physics. Physics will add more realism to a game, making objects bounce, roll along a road or fall down a cliff or monsters fall to the ground in a way depending on where they are hit. The physics engine is the part of the program that calculates how the objects should move and interact with other objects. This thesis will investigate different physics engines, both commercial and non-commercial products and put some of them through several tests. Finally, I have designed and implemented an interface that can be used independently of physics engine.

Physics engines can simulate not only stiff objects but also things like fluids, smoke and cloth. This report will primarily look at physics engines for the stiff objects called rigid bodies, but other systems will be glanced at.

1.1 Background

Craft Animations AB [I01] is a software company, located in Gothenburg in Sweden, which is producing and selling real-time software for 3D animation, Virtual Reality (VR) and gaming industries. Their “Craft Director Tools” is a series of plug-ins for controlling and recording movements with realistic physics. The user can choose different plug-ins depending on which type of movement wanted to achieve giving smooth and realistic movements with correct physics.

Craft Animations suggested this project because they needed a good evaluation of physics engines on the market today and wanted a physics interface for future implementations. They also needed the interface to be able help other companies getting started using physics engines.

1.2 Purpose

I wanted to study physics engines because I want to learn more about the physics engines functionality and importance for gaming and animation. The purpose with this thesis was to evaluate some physics engines on the marked today and to create an interface making it easy for Craft Animations to change between physics engines.

1.3 Problem formulation

- What is a physics engine and what does it consist of?
- Which physics engines are on the marked today? What are their functionalities?
1.4 Delimitations

This thesis will focus on presenting the basic functionalities of a physics engine. Algorithms will not be examined or evaluated. There are more physics engines than presented in this thesis and a few have been chosen for evaluation. It would be interesting to test all physics engine in runtime tests, but due to limited time, only two engines were evaluated more closely.

1.5 Method

Here are the methods used in this thesis presented.

1.5.1 Collection of information

When beginning the work with this thesis the goal was to get an good overview of the concept of physics engines. From books, articles and internet resources the most relevant information was acquired. The books and internet resources gave a good introduction to the subject and the articles gave more detailed information. To get information about the physics engines the primary source of information is the engines websites and forums at the sites. For information about how to write a report I used the book about how to write reports by Booth[07].

1.5.2 Reference critics

Many references in this report are taken from the physics engines websites and forums at those websites. That way I got the latest information about the engines and to be able to communicate with the creators of the systems. This method might not give an objective view over the engines, but it had to be done to really get the latest functionality information about them. The positive side of this method is that the probability of correct information, when the information from forums comes from the creators of the engine themselves, is quite high. The basic information about how physics engines work was mostly acquired from books and articles because references like that are more scientifically accepted, which should raise the scientific correctness of the report.

1.5.3 Runtime tests

To test the engines at runtime a test program has been implemented. The program has been made in C++ using Microsoft Visual Studio 2005, the standard tool at Craft Animations. The test program no other functionalities than to show how the engines perform. The textures of the objects and the ground were taken from the Newton Dynamics Tutorial 2[I02] just to give the program a nicer look. The values from the programs were inserted into a Microsoft Excel document and graphs representing the results were created to give a better overview.

1.5.4 Implementation of interface

The interface was written in C++ using Microsoft Visual Studio 2005. The design of the interface was chosen in discussion with Craft Animations and by examining physics engines. UML-diagrams were created to get an overview of the interface and the diagrams were created with Visual Paradigm Free Edition, version 5.3 (www.visualparadigm.com).
2 Theory

The following chapter will give an introduction to what a physics engine consists of and how it works. First of all let us explain some basic terms used later in the report.

2.1 Basic terms used by a physics engine

To simulate physics in a computer game or in an animation you have to know quite a lot about physics in real life. This section will explain some physics terms, terms from linear algebra and other terms used by a physics engine. How several of these terms are calculated and implemented in a physics engine is well explained by David H. Eberly in [02].

2.1.1 Force

According to Eberly in [02], a force is a mechanism for changing the mechanical state of an object. A force is a vector quantity with direction and magnitude. Forces are what change the velocities of the objects and cause them to interact with each other. The standard unit for force is newton (N).

2.1.2 Torque

Torque is also referred to as moment of force. Torque \((T)\) is defined in the Swedish National Encyclopedia [I03] as the product of the force \((F)\) and the orthogonal distance \((d)\) measured between a point \((p)\) located in an origin and the force’s direction (see figure 2.1). You can say it’s the force that make an object rotate. The standard unit is newtonmeters (Nm).

\[
T = F \times d
\]

![Fig. 2.1: Torque](image)

2.1.3 Moment of Inertia

Moment of inertia is described by Eberly [02] as a measure of the rotational inertia of an object about any axis. The more difficult it is to make an object rotate about an axis, the larger moment of inertia about that axis.

2.1.4 Center of mass

Bourg [01] defines the center of mass as a point on the object where the mass of it is evenly distributed. You could say it’s a point on an object where a hit doesn’t make it rotate.

2.1.5 Friction

Friction is, as described by Bourg in [01], a force that resists motion and occurs between two contacting surfaces. It is a contact force. Always parallel to the contacting surface at the point of contact. Eberly in [02] describes two types of friction, static and kinetic friction.
An example of static friction is when you try to move a not already moving object. If you use a too small force, the object will not start moving due to the static friction. The force that you push with is smaller than the static force of friction, \( F_{\text{static}} \geq F_{\text{push}} \) (see figure 2.2a). When the object has started moving a new kind of friction starts to apply, namely kinetic friction. When the \( F_{\text{push}} \) is getting larger than \( F_{\text{static}} \), \( F_{\text{static}} \) becomes \( F_{\text{kinetic}} \) (see figure 2.2b). One approximation of the force of friction is the Coulomb friction model where the coefficient of friction is a scalar value which describes the ratio between two objects and the force pressing them together \([A12][I04]\).

![Diagram of static and kinetic friction](image)

**2.1.6 Linear velocity**

The linear velocity is the rate of displacement of an object with time. The standard unit is meters per second. It is a vector quantity with direction and magnitude. [02]

**2.1.7 Angular velocity**

Angular velocity is the speed at which an object rotates. It is a vector quantity with direction and magnitude. The standard unit is radians per second. [02]

**2.1.8 Transformations**

As defined in the book “Computer Graphics with OpenGL, third edition” by Donald Hearn and M. Pauline Baker [03], operations that are applied to objects to change position, orientation or size are called geometric transformations. They are represented with matrices. For more information about how the transformations are performed, the reader is referred to the book mentioned above [03].

**2.1.9 Quaternion**

A more compact way to rotate an object around an axis is to use quaternions for the rotation transformation. The quaternion require less space than an 4x4 transformation matrix and its simpler to write quaternion procedures for transformation sequences. It is well described in the book by Donald Hearn and M. Pauline Baker [03] how quaternions are characterized and how they are used.
2.2 What is a physics engine?

To make a game realistic, the Newtonian laws of real life must apply also in the computer world. You can describe a physics engine as a computer program that simulates these laws of physics on a computer [02]. As discussed in the Havok primer [I05] and [I06], a physics engine has three tasks to perform:

1. Detect collisions
2. Solve constraints
3. Update the system

The following chapter will explain each of the three tasks more closely and discuss different kinds of objects (bodies).

2.2.1 Collision detection

Collisions are a big part of computer games, real-time simulations and similar applications. In a computer game, collisions stop the characters from going through walls or falling through the floor and it can tell whether a monster in the game can see the character and e.g. attack. To detect whether a collision has occurred, a collision detection system is used.

The collision detection system is the most crucial part of the physics engine. It is the part of the engine that takes most CPU time (about 90%). Collision detection is matter of detecting if two or more objects in a simulated world are intersecting. For a world with \( n \) objects, then in worst case we need to do \( n(n-1)/2 \) checks to be sure that all pairs are checked for intersection. So if for 100 objects we need to do 4950 checks, it quickly gets expensive. To speed up the process there are a few methods that can be applied. First of all the complexity of the objects to be tested can be reduced. Secondly the number of objects can be reduced. Thirdly, simple collision tests can be done first to reduce the number of detailed collision tests. This is discussed in the Havok primer [I05].

Collision testing

Collision testing can be divided into a series of collision tests. Each test is more complex than the other, but after each test as many objects as possible are eliminated from the test process [I05]. The first test is for determining possible collisions between pairs of objects (often called broad phase). This can be done with different methods.

One method is Space Partitioning, which is discussed in the book “Collision detection in 3d environments” by Gino van den Bergen in [04]. It is a subdivision of the simulated room (collision space) into convex regions, called cells. Each cell maintains a list of references to objects that are (partially) contained in the cell. Only pairs of objects that share the same cell needs to be tested for intersection. This rejects a lot of object pairs from intersection testing. There are different structures of space partitioning and examples of it are Voxel grids, Octrees, k-d Trees, and Binary Space Partitioning (BSP) Trees. Gino van den Bergen continues saying that there is a drawback with space partitioning. Since cells maintain a reference to the objects in the cell, objects overlapping between cells are maintained in multiple cells. This can lead to either lots of intersection tests being repeated for the same pairs of objects or additional overheads for keeping records of pairs that have been tested.
According to Gino van den Bergen [04], another method called Model Partitioning is better than space partitioning since it does not have multiple references of objects. In this method you subdivide a set of objects into coherent subsets and compute a bounding volume for each subset of objects. In this way, subsets of objects can be quickly excluded from intersections testing depending on whether the bounding volumes overlap. This can be done with an algorithm called Sweep and Prune. Examples of bounding volumes are spheres, AABB (Axis Aligned Bounding Box) and OBB (Oriented Bounding Box), see figure 2.3.

The intersection test is often called the Narrow Phase. This is where the actual object polygons are tested for overlapping. This is a very costly process and should be avoided as far as possible. There are several methods for calculating this, e.g. Closest Feature. [A01]

Collision Primitives
Gino van den Bergen states in his book in [04] that the shapes of the objects being tested for collision has a major impact on the speed of the collision test. If the shapes are simplified for collision testing a lot of CPU time can be saved. If we assume that the objects are rigid the shapes do not vary from one time step to another. There are several shape primitives used for simplifying the objects. The most commonly used in 3D applications are spheres, boxes, line segments (rays), triangles and general polygons.

Ray casting
Ray casting is a very useful feature in a physics engine. For example, it can be used when modeling a car, where the wheels of the car shoot out rays towards the ground to get the distance to it. The idea with raycasting is to cast a ray from a point in space in a certain direction and get the distance to the closest body in the scene [I07].

Gino Van den Bergen proposes in his paper in [A02] an algorithm for calculating the hit point and normal of a ray and a general convex object. The algorithm is based on the Gilbert-Johnson-Keerthi [04] algorithm which is used for computing distances between convex objects.

Fast moving objects
Since time is discrete in the simulation (see section 2.4) there is a risk that collision for fast moving objects is detected too late or not at all. A bullet for example might pass through an object without resulting in a collision for any of the sampled time intervals (see figure 2.4). This problem can be reduced by increasing the sampling rate but
according to Gino van den Bergen [04] there is always a chance that a collision is not detected and the computational load of the simulation increases.

One popular method for solving the problem is using Continuous Collision Detection (CCD). The method is used by several physics engines in this thesis. In the paper by Redon and Kim among others[A03], CCD for rigid bodies are discussed and it is said that there are a few different algorithms for continuous collision detection between rigid bodies. All algorithms model a trajectory of the object between successive discrete time steps and they check the resulting path for collisions. More specifically, the paper [A03] continues, there are a few different approaches for the algorithms. Among them are the swept volume- and adaptive subdivision approach. The swept volume approach is based on calculating the swept volume (the geometric space occupied by an object during a time interval, see figure 2.5) of the object and checking for collisions between the swept volume and the rest of the environment.

The paper mentioned above, [A03], describes the adaptive subdividing approach and states that it does a separation test which ensures separation between objects between some time intervals. If the test fails on a time interval it subdivides that interval selectively until the interval becomes smaller than a tolerance set along the time dimension.

2.2.2 Constraints

When a collision has occurred, you usually want some sort of response to the collision. For example, you might want the objects to bounce off each other and continue to move in other directions or maybe you want objects to move along a specific curve or keep objects at a distance apart [A04]. This is where constraints come into play. The problem with constraints in dynamic simulations, discussed in the paper by Witkin [A04], is to make objects obey the Newton’s laws and at the same time obey the geometric constraints. In this chapter, collision response and different kinds of joints will be discussed and the collisions will be restricted to rigid bodies.
Collision response
When the contact points are known from the collision detection, the constraint solver takes over to adjust the physical parameters of the objects based on the type of contact. Contact points can, according to Eberly in [02], be divided into two categories based on how the objects collide at a point, either colliding contact or resting contact. The adjustments of the physical parameters for a colliding contact requires a force called impulse force[02] and for a resting contact requires computing the contact force. This approach described here is, according to Eberly in [02], to enforce non penetration constraints and is quite popular but not the only way to handle the physics. Another way, not described here, is based on Lagrangian dynamics and is described in the book by Eberly in [02].

Impulse force
A colliding contact is defined by Eberly [02] as a contact point where the velocities of the objects colliding cause them to penetrate into each other. To stop the objects from interpenetrating the obvious way would be to apply forces to both objects, as written in the article by Hecker [A05]. However, this way will not do because force acting on an object cannot instantaneously change a velocity. It takes some time to change a velocity and it can only be done via integration over time. The objects are already touching so there is no extra time to allow the force to apply a counterforce. The velocity must be changed immediately. That is what an impulsive force can do; it can change velocities directly, without waiting.

There are many ways to calculate the impulse magnitude and direction; I will present one model, made in 2D, described in the article by Hecker [A05]. The model is called “Newton’s Law of Restitution for Instantaneous Collisions with no Friction” and it assumes that collisions take no time, all non collision forces (such as gravity) go away during the collision and only impulses are calculated. This model is relatively simple, but it can still give interesting behaviour. The model uses a coefficient called restitution (denoted $e$), which tells how much of the incoming energy is dissipated during collision. A totally elastic collision has $e = 1$, and a totally plastic collision has $e = 0.0$. Knowing the incoming velocities, the collision normal and the masses of the objects the impulse can be calculated. Equations for calculating the impulse for two colliding objects can be found in Hecker’s article [A05].

Contact force
A resting contact is a contact point where the velocities of the objects neither cause the objects to penetrate nor separate. In this case the contact force has to be calculated. The force must satisfy three conditions:

1. The force must prevent the objects from interpenetrating.
2. It must be a repulsive force; it cannot act as glue between the objects.
3. The force must become zero when the objects separate.

The calculation of this force is a linear complementarity problem (LCP), see the book by Eberly [02] for more equations.
Variations
An alternative to using impulse forces is to use penalty-based methods to solve the interpenetration constraints. The idea with this method is to track the distances between two objects and instead of using impulsive forces, the system modifies the simulation internally by adding springs and when the distance between the objects is a certain value the spring exert repulsive forces on them.[02]

Joints
Joints in real life are things used to connect objects together, like a door hinge or a cartowing hook. Joints are used by the physics engine to put constraints to the objects so that they only have certain positions and orientation relative each other [I08]. At each integration step (see section 2.4), the joints apply constraint forces to the objects that they affect.

![Image](a) (b) (c) (d)

Fig. 2.6: (a) Ball-Socket joint (b) Hinge joint (c) Slider joint (d) Universal joint [I08][I09]

The most common types of joints, used in many physics engines are the Ball-Socket-, Hinge-, Slider- and the Universal-joint. Each type is shown in figure 2.6. There are more joint types specific for each physics engine and not all of them can be shown here.
2.2.3 Update system

The update system does three things: acquires results from the constraint solver, determines the new state of the objects and advances the time.

The real world is continuously evolving; objects are moving, colliding and reacting to things all the time. To simulate this with a physics engine, the update system uses time steps. At a given point in time we can get the state of an object (e.g. position) and if we know the forces acting on the object at that time we can make an approximation of the change in state for the object after a period of time (the time step size) has elapsed. The smaller step size the more accurate approximation. The approximation is done by integrators that implements numerical integration of a series of differential equations describing the motions of objects. There are different integrators available, each vary in CPU load and accuracy (see table 2.1). [I05]

<table>
<thead>
<tr>
<th>Integrator</th>
<th>CPU load</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Euler’s method</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Midpoint</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Runge Kutta (RK45)</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 2.1 A few integrators and their properties [I05]

Euler’s method

According to Witkin and Baraff in [A06], the Euler method is the simplest numerical method. Let $x$ be a state of the system, $x_0$ the initial state at time $t_0$ (denoted $x_0 = x(t_0)$) and denote the estimate of $x$ at a later time $t_0 + h$ by $x(t_0 + h)$ where $h$ is the step size. Euler’s method computes $x(t_0 + h)$ by taking a step in the derivative direction,

$$x(t_0 + h) = x_0 + h \dot{x}(t_0)$$

Fig 2.7 Euler Method formula [A06]

where $\dot{x}$ is the derivative of $x$. A point in the system now follows a polygonal path instead of the real integral curve. Euler’s method is neither accurate nor effective and shrinking the step size will not eliminate the error, only make it smaller (see. Figure 2.8).
**Midpoint method**

The midpoint method is a refinement of the Euler method. This method first evaluates an Euler step, then performs a second derivative equation at the midpoint of the step and use this to update $x$. The formula looks as follows (see figure 2.9):

$$x(t_0 + h) = x(t_0) + h\dot{x}(t_0) + \frac{h^2}{2}\ddot{x}(t_0)$$

Fig 2.9 Midpoint method [A06]

Where $\dot{x}$ is given by the function $f(x(t), t)$. [A06]

**Runge Kutta**

According to Witkin and Baraff in [A06], the midpoint method can be refined even more by splitting the Euler method into more substeps. The most popular method for this is the Runge Kutta of order 4. The midpoint method can be called the Runge Kutta of order 2 (Euler is first order). The formula for calculating $x(t_0 + h)$ with Runge Kutta of order 4 is:

$$
\begin{align*}
  k_1 &= hf(x_0, t_0) \\
  k_2 &= hf(x_0 + \frac{k_1}{2}, t_0 + \frac{h}{2}) \\
  k_3 &= hf(x_0 + \frac{k_2}{2}, t_0 + \frac{h}{2}) \\
  k_4 &= hf(x_0 + k_3, t_0 + h) \\
  x(t_0 + h) &= x_0 + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4.
\end{align*}
$$

Fig 2.10: Runge Kutta formula [A06]

Fig 2.11 Fourth-order Runge Kutta example. In each time step the derivative is evaluated four times. Once at the initial point (1), twice at midpoints (2 and 3) and once at an end point (4). From these points the function curve is calculated. [F05]
2.2.4 Bodies

Physics engines can simulate different kinds of bodies (objects) and in this chapter three types of bodies will be explained. Most of the open source physics engines I have examined simulate only rigid bodies, but here other types of bodies will be discussed.

Rigid bodies

Rigid body means that we put constraints on the object. The shape on the object does not change during simulation. In reality, no body is rigid, but in many simulations an assumption of rigidity is a close enough approximation to actual physical conditions [02]. This means that it is not possible to make flexing objects like jello, but it’s still possible to simulate human bodies, objects made of metal or wood etc. This makes the equations much simpler and can still give some good results according to Hecker in the article [A07].

Deformable objects

In some simulations, rigid bodies are not sufficient and we want to be able to simulate deformable objects. There are several methods for modeling deformable objects, but some are very computationally heavy. I will outline one method that is not so heavy.

The method is called Mass-spring systems [02], and is according to Andrew Nealen among others [A08], the far most simple and intuitive way to model a deformable object. The body can be modeled as a system of point masses connected by springs. The bodies can be curve masses (e.g. hair or rope), surface masses (e.g. cloth) or volume masses (e.g. jello), as described by Eberly [02]. The complexity of a system is dependent of how many masses an object has and how they are interconnected. A curve mass is thought of as a polyline open with two end points or closed with no end points (see figure 2.12a). Each vertex of the polyline represents a mass and each edge represents a spring connecting two masses. A curve mass is modeled as a one-dimensional array. A surface mass is represented by a collection of particles arranged as a two dimensional array (see figure 2.12b) and a volume mass is represented with the masses organized as a three-dimensional array.

Fig. 2.12 (a) a curve mass object and (b) a surface mass object represented with a mass-spring system. [02]
This method is expensive since it requires differential equation solving but the result looks realistic. Other methods for modeling deformable objects are for example Control-Point Deformation, Free-Form deformation, or Implicit Surface deformation, all which are described by Eberly[02].

**Particle systems**

Particle systems is a method to simulate things like smoke, fire clouds, trees etc. They are dynamic, have no well-defined surface and are non-rigid bodies (see figure 2.13).[A09] Objects consisting of particles are not deterministic, their shapes are not completely specified. New particles are created and old are destroyed. When created, a particle can be given a lifetime in frames and when the lifetime is over the particle is destroyed. There are other ways to destroy a particle, for example destroying it when a particle leaves a region of interest. A particle can be exposed to forces and have a velocity. In the article by Yi Wu and Daniel Thalmann[A10] a technique for creating and animating deformable surfaces using particle systems is presented. Particle systems are close related to the mass-spring method mentioned above. The point masses in the mass-spring system are particles held in certain positions relative each other. When simulating a cloud or fire, no certain positions relative particles are being held.

![Fig. 2.13. A particle system modeling fire.][F06]

### 2.2.5 Conclusion

A physics engine consists of many parts and there are many ways to create them. Most physics engines are built in similar way consisting of a collision detection module, a constraint solver module and a time stepping module that solves the objects motions, all described above. Collision detection is a critical performance part and there are several algorithms and methods for implementing it. In this chapter the basics of collision detection was presented to give an overview of how it works.

Constraints is a very important part of the physics engine for simulating realistic scenes. There are several kinds of constraint solvers and in this chapter one approach was discussed.

As discussed in the paper by Andrew Nealen and Matthias Müller among others [A08], deformable objects are very demanding computationally and models and algorithms for deformable objects are quite limited in todays applications and video games. Thanks to new physics processing units (such as Ageia [I10], see section 3.4.1) it is possible to create more and more complex deformable objects. The most physics engines in this thesis does not support deformable objects.
A physics engine doesn’t know or care anything about how the objects are displayed. It simulates the motion and interaction of the objects based on a physical description. To display the scene the user has to report the objects’s states to the display system.[I05]
3 Engines today

In this chapter, I will look at what kind of engines there are on the market today, what they consist of and what they can offer. I will look at free and open source engines, but I will conclude the chapter by discussing cutting-edge commercial software and hardware engines. I made an overview of a few engines and then, according to their functionality, two engines were chosen to be examined more closely and put through a series of tests. The engines to examine were chosen in discussion with Craft Animations and by searching the internet to find the most popular engines.

3.1 Overview

Six areas will be presented here for each engine. The areas were chosen in discussion with Craft Animations and after a quick glance of the engines. This was done to find the common features in the engines. The areas are:

1. Joint types supported
2. Collision primitives supported
3. Data types supported
4. Types of integrators supported
5. Platform support
6. Other functionality

3.1.1 Open Dynamics Engine (ODE)

Open Dynamics Engine (ODE) is an open source dynamics engine, developed by Russell Smith, started in 2001. It is made for simulating articulated rigid body dynamics. Typically good for simulating ground vehicles, legged creatures and moving objects in a VR environment [108]
Written in C, but with an interface for C++. [109]

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Supports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joint Types</td>
<td>Ball-and-Socket, Hinge, Slider, Contact, Universal, Hinge-2, Fixed, Angular Motor.</td>
</tr>
<tr>
<td>Collision primitives</td>
<td>Sphere, Box, Cylinder, Infinite plane, Geometry transform, Ray, Triangle mesh.</td>
</tr>
<tr>
<td>Datatypes</td>
<td>3x3, 4x3 and 4x4 matrices (row-major), 3- and 4-vectors, Quaternion (W-X-Y-Z). Single precision or double precision.</td>
</tr>
<tr>
<td>Integrator</td>
<td>Euler</td>
</tr>
<tr>
<td>Platforms</td>
<td>Windows, Linux, MacOS</td>
</tr>
<tr>
<td>Others</td>
<td>Uses collision spaces (Quad tree, hash space, and simple) which can divide collisions into hierarchies. External collision detection can be used.</td>
</tr>
</tbody>
</table>
3.1.2 Tokamak Game Physics
Tokamak is a free physics engine (not open source) created by David Lam in 2003. It is a real-time physics library designed specially for games. Tokamak is written in C++. [I11]

Table 3.2 Functionality of Tokamak [I11]

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Supports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joint Types</td>
<td>Ball-and-Socket, Hinge, Joint Limits.</td>
</tr>
<tr>
<td>Collision primitives</td>
<td>Sphere, Box, Capsule, Convex mesh, Static triangle mesh.</td>
</tr>
<tr>
<td>Datatypes</td>
<td>3- and 4-Vector, 3x3- and 4x4-matrices Quaternion(X-Y-Z-W). Single precision.</td>
</tr>
<tr>
<td>Integrators</td>
<td>No information</td>
</tr>
<tr>
<td>Platforms</td>
<td>Windows</td>
</tr>
<tr>
<td>Others</td>
<td>Breakable Objects, convex to convex collision detection</td>
</tr>
</tbody>
</table>

3.1.3 True Axis
The True Axis Physics SDK is free for non-commercial use created 2004 in Australia. It is designed for speed and sacrifices accuracy. It is written in C++. [I12]

Table 3.3 Functionality of True Axis [I12]

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Supports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joint Types</td>
<td>Ball-and-Socket, Hinge, EulerConstraint, SquareSocket, Slider, RotationConstraint, VectorConstraint.</td>
</tr>
<tr>
<td>Collision primitives</td>
<td>Box, Sphere, Capped Cylinder, Ray, Convex.</td>
</tr>
<tr>
<td>Datatypes</td>
<td>2-, 3- and 4-Vector, 3x3-matrices. Single precision.</td>
</tr>
<tr>
<td>Integrators</td>
<td>No information</td>
</tr>
<tr>
<td>Platforms</td>
<td>Windows</td>
</tr>
<tr>
<td>Others</td>
<td>Swept collision detection (method for handling fast moving objects). Vehicle simulation library.</td>
</tr>
</tbody>
</table>

3.1.4 Newton Game Dynamics
Newton Game Dynamics is a physics engine mainly for real time simulation of rigid bodies. It is not open source but the interface is free, created in 2003 by Julio Jerez. Written in C. [I13]

Table 3.4 Functionality of Newton Game Dynamics [I13]

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Supports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joint Types</td>
<td>Ball-and-Socket, Hinge, Slider, Corkscrew, Universal, Upvector, User defined.</td>
</tr>
<tr>
<td>Collision primitives</td>
<td>Box, Sphere, Cone, Capsule, Cylinder, Convex Hull, Ray, Usermesh.</td>
</tr>
<tr>
<td>Datatypes</td>
<td>4x4-matrices (row-major), 3-vector. Single precision.</td>
</tr>
<tr>
<td>Integrators</td>
<td>Euler</td>
</tr>
<tr>
<td>Platforms</td>
<td>Windows, Linux, MacOS</td>
</tr>
</tbody>
</table>
3.1.5 Bullet
Bullet is a 3D Collision Detection and Rigid Body Dynamics Library for games and animation. It is a toy project created by Erwin Coumans in 2005. It is free for commercial use, written in C++. The iterative constraint solver is taken from ODE.

Table 3.5 Functionality of Bullet [I14]

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Supports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joint Types</td>
<td>Hinge, Point to Point, 6DOF, Motors.</td>
</tr>
<tr>
<td>Collision primitives</td>
<td>Convex Polyhedron, Box, Sphere, Cone, Cylinder, Capsule, Compound, Static Triangle Mesh.</td>
</tr>
<tr>
<td>Datatypes</td>
<td>3-vector, 4-vector, 3x3 and 4x4 matrices, Quaternion. Single precision.</td>
</tr>
<tr>
<td>Integrators</td>
<td>No information</td>
</tr>
<tr>
<td>Platforms</td>
<td>Windows</td>
</tr>
<tr>
<td>Others</td>
<td>Continuous Collision Detection. Bullet is using the Sweep and Prune algorithm for convex collision detection.</td>
</tr>
</tbody>
</table>

3.1.6 Conclusion
All engines that I have looked at in this chapter have similar functionality when it comes to collision primitives, joint types and data types.

ODE supports the most joint types of all the engines and has good collision primitives like ray and triangle mesh. Another good thing is that it has support for both single and double precision. It has also good documentation and several example projects that make the engine easy to work with.

Tokamak on the other hand doesn’t support many joint types and has no support for primitives like rays. The documentation for Tokamak is good and another good feature is the support for breakable objects.

True Axis is designed for speed and not for accuracy and it has no support for quaternions or double precision. The vehicle library is a good feature with True Axis.

Newton Dynamics supports many joint types and user defined joints. Another good thing is the possibility to create your own collision shapes, also the vehicle and rag doll containers. The documentation is good with several tutorials on how to get started.

The Bullet engine uses several functionality from ODE but one thing that differs from ODE is the collision detection system which is using CCD. The engine is under development and it’s interesting to see what the future might hold for this engine. Bullet does not support a lot of collision primitives or joint types, but instead there is functionality for creating user defined primitives and types.

From this overview of the engines I chose ODE and Newton for more examination and for runtime testing. They were also chosen because of their other functionality, such as user defined collision primitives, vehicle containers and ray. This is functionality that can be interesting for Craft Animations to use in their products.
3.2 A closer look

This section will take a closer look on the two engines, ODE and Newton, chosen in the previous section (3.1). The engines were be put under the magnifying glass within four areas. These are world-, rigid body-, joint-, collision detection-functionality.

3.2.1 Open Dynamics Engine (ODE)

The following information about ODE is an excerpt from the ODE website [I08][I09].

1. The World

   The world contains rigid bodies and joints. Several worlds can be created but bodies in different worlds cannot interact with each other. When destroying a world, all bodies in it will be destroyed as well. The world data structure consist of e.g. two linked lists (for bodies and joints in the world), a gravity vector, and ERP- and CFM-parameters (explained later).

   There are two ways to step through the world, by Step or QuickStep. These functions include both stepping of the world and solve the constraints. Step uses a “big matrix”-method that takes time $O(m^3)$ and memory $O(m^2)$, where $m$ is the number of constraint rows. The QuickStep function uses an iterative-method that takes time $O(m^4N)$ and memory $O(m)$, where $m$ is the number of constraint rows and $N$ is the number of iterations.

2. Rigid bodies

   Creating and destroying rigid bodies can be performed dynamically. The position of the body is in ODE represented with a 3 x 1 vector, which is the position of the body’s centre of mass. The orientation of the body is represented with a 3 x 3 rotation matrix or with quaternions.

   The mass of each body is described with a mass structure without units. This structure consists of:
   - The total mass of the body
   - The centre-of-mass-position
   - A 3x3 inertia matrix.

   Forces are accumulated to each body and the accumulator is set to zero after each time step. It is possible to specify points on the body where forces should be applied; otherwise they will be applied to the centre of the body.

   Every body in the simulation can be enabled or disabled. When a body is created, it’s automatically put in an enabled state. A disabled body does not consume any CPU-time. Thus to make the simulation go faster, one can disable bodies that come to a rest. Automatic disabling can be done in two different ways with functions in ODE. A body can be disabled if it has been idle for a given number of simulation steps or it can be disabled after a given amount of simulation time.
3. Joints
You can connect a joint to either one or two bodies. There are eight types of joints supported:

- Ball-and-Socket
- Hinge
- Slider
- Contact: prevents two bodies from interpenetrating at the contact point.
- Universal
- Hinge-2: same as two hinges connected in series, with different hinge axis.
- Fixed: maintains a fixed relative position and orientation between two bodies.
- Angular Motor: allows the relative angular velocities of two bodies to be controlled.

When joints are connected to bodies they are supposed to hold the bodies in positions and orientations relative to each other. It can happen that these constraints are not met. To reduce this joint error, ODE has a mechanism called Error Reduction Parameter (ERP). The ERP can take values between 0 and 1 and specifies the proportion of the joint error that will be fixed during the next simulation step. ERP adds a special force to the joints to force the bodies back in alignment.

It is sometimes needed to violate the joint constraints, e.g. when you want objects to intersect each other (like soft objects). Constraint force mixing (CFM) is used for this purpose. CFM allows the constraint to be violated with an amount proportional to a CFM-parameter. Using ERP together with CFM can give various effects, for example springy constraints or spongy constraints.

In ODE, the constraint equations look as follows:

\[
J_1 v_1 + \Omega_1 \omega_1 + J_2 v_2 + \Omega_2 \omega_2 = c + C \lambda
\]

\[
\lambda \geq l,
\]

\[
\lambda \leq h.
\]

Equation 3.1: ODE constraint equation

Where \( J \) and \( \Omega \) are Jacobian matrices and \( v \) and \( \omega \) are the linear and angular velocity for the first and the second body. \( \lambda \) is a constraint force that is applied to the bodies to ensure that the first equation is satisfied. The second and third equations are for restricting the \( \lambda \) value to a lower (\( l \)) and a higher (\( h \)) bound. \( C \) is a CFM-matrix. Manipulating this can give special effects to the constraint, as mentioned above.

4. Collision detection and response
When a collision has occurred, the contact is represented by a contact point (gives the contact position, normal vector, penetration depth and which objects are colliding). The collision detection returns an array of contact points for which the user must create joint contacts before the stepping function is invoked, which is stated by Erleben[106]. Erleben states also that the stepping function starts by detecting contact graphs, in ODE called “islands”.

ODE’s collision system provides a fast identification of potentially intersecting objects with its concept of “spaces”.
Geometry objects
Geometry objects (geoms) are objects in the collision system. A geom can represent a rigid shape or several rigid shapes. The geoms can collide and create one or more contact points. A geom does not need to be connected to a rigid body; it can have its own transformation matrix. Every geom is an instance of a collision primitive. In ODE, collision primitives are sphere, box, plane, capped cylinder, ray, triangle mesh, user defined primitives and geometry transform. The geometry transform primitive encapsulates other primitives. This allows the encapsulated primitive to be positioned and rotated arbitrarily with respect to its point of reference.

Spaces
A space is a geometry object (geom) that can contain other geoms. A space does a check to see which geoms in it that are potentially intersecting. After that you can do a collision check to get the contact-points for those geoms.

There are three different space types. They all use different algorithms for the collision culling and have different internal data structures:

- Simple space: This method does no collision culling; it checks every pair of geoms to see if any of them overlap. The method should not be used with large amount of objects but it is useful when debugging.
- Multi-Resolution hash table space: uses a data structure that records how geoms overlap cells in several 3D grids.
- Quad tree space: uses a hierarchical grid-based axis aligned bounding box tree for the collision culling. It is fast for large environments.

For every time step in the simulation we want to know which objects that are colliding and create a list of contact points for those. ODE has three functions for this:

- dCollide(): intersects two geoms and creates contact points.
- dSpaceCollide(): determines which pairs that are potentially intersecting. Does not create contact points directly. User can choose to call dCollide() for every pair.
- dSpaceCollide2(): determines which geoms in a space can collide with geoms from another space. Useful when there are spaces inside other spaces.

Russel Smith, the creator of the ODE engine, states in his forum [I15] that ODE uses impulse-based collision response and corrects positional errors by applying a velocity to objects and allowing that velocity to take effect over one time step.

Materials
ODE uses the Coulomb friction model. You can specify a lambda-coefficient to simulate different materials. The ERP- and CFM-parameters can also be used for simulating softness and bounce.
3.2.2 Newton Game Dynamics

The following information is an excerpt from the Newton Game dynamics website [I13].

1. The World
The Newton world consists of rigid bodies and joints. No gravity vector can be set, as opposed to ODE. It has to be done in a callback-function where the forces and torques are applied to the objects.

Newton is using an Euler integrator method for integration which is fast but not so accurate, as discussed in section 2.2.3.

In the Newton world you can set how accurate the physics solver should be. There are three different modes for this:
- Exact mode: used in realistic simulation (more precision, less speed).
- Adaptive mode: A little less precision than exact mode, but still a high degree of accuracy
- Linear mode: The fastest mode, good when speed is most important (e.g. games)

The world has an active simulation list where all bodies that are active are listed. The user can freeze specific bodies and exclude them from the simulation (Compare with ODE’s enabling/disabling).

2. Rigid bodies
When creating a body with Newton, a collision object is assigned to the object via a collision pointer. The creation increments the reference counts of the collision geometry. When creating a new rigid body it gets a unique id. If a body is deleted inside a simulation step, it will not be removed until the end of the time step. All joints connected to the body are also deleted. The transformation matrix for a rigid body is in row-major-order. Another body feature is the ability to add buoyancy to a body.

The mass has no SI-units, it can be any unit needed. To make an object static the user either pass mass = 0, or doesn’t call the set mass function. The center of mass of the body can be set, as well as moment of inertia.

3. Joints
Newton’s constraint solver is not revealed but the creator of the engine, Julio Jerez, says it is mathematical correct and that it is using a method that is not using any micro collision technique and not using a Linear Complementarity Problem (LCP) solver.

Seven joint types are supported:
- Ball-and-Socket
- Hinge
- Slider
- Corkscrew is an enhanced version of a slider joint. It allows one body to rotate around the other.
- Universal
- Upvector: a specialized vector to limit rotation to one axis.
- User defined
**Vehicle and Ragdoll containers**

The ragdoll is a joint container specialized for animating death sequences or humanoid models and is made of ball-socket-joints.

The vehicle container implements a full rigid body vehicle. On the unofficial Newton Dynamics website [107] it is stated that it is difficult to tweak the different parameters for the vehicle joint to make it act accurate and that no gearbox is implemented.

**Friction**

There are two ways to set the accuracy for modelling friction using the Coulomb friction model:

- **Exact model:** Friction is calculated for every frame. Used when high precision is needed.
- **Adaptive model:** Friction values from a previous frame are used to calculate friction in current frame. 10 % faster than the exact model, but may give strange behaviour.

**Materials**

Newton uses a material graph for solving the physics behaviour of rigid bodies. The material graph is an undirected graph where the nodes are Material IDs and the edges are material interactions. The IDs are unique and represent different materials like wood, metal, concrete etc. The material interactions are objects that store information about the behaviour of two material IDs that come in contact. Directly after the world is created it is preferable to create the material graph for the whole scene, this is because it’s expensive to make materials and they can not be destroyed individually. So it’s important to decide early in the project how many materials are to be used in the scene.

4. **Collision detection**

Collisions in Newton use collision geometry the same way as ODE. The main differences are that in Newton the collision geometry cannot exist without a body and the space concept doesn’t exist. It is possible to offset the collision geometry from the body using an offset matrix. This matrix can only be used for rotation and translation, scale is not allowed [107].

Newton supports eight collision primitives including box, sphere, cone, capsule, cylinder, convex hull, ray and user mesh.

In an official Newton dynamics tutorial [107] it is stated that modelling the world with collision primitives is neither practical nor efficient. For that reason, Newton provides two special collision primitives, *Collision Trees* and the *user defined collision*. These primitives are best suited for static objects like background geometry with arbitrary shape or form and with unlimited number of polygons. The Collision trees in Newton works as described in the Collision tree tutorial [107]. It can be described in a simplified way as follows. You build the collision tree by calling the begin-build-function and after that you add faces to the tree by using an add function. When all faces you want in the tree are added you call the end-build-function, which tells Newton to compile the collision geometry. The tutorial finish off by saying it is a good idea to simplify the collision geometry as much as possible since complex shapes will give performance and accuracy problems with Newton. The user defined collision primitive is a method very similar to the collision tree, but is more efficient when modelling moveable objects.
3.2.3 Conclusion

In this chapter, the physics engines ODE and Newton were examined more closely. The two engines are very alike when it comes to functionalities. They both use a first order integrator (Euler) and the worlds consist of almost the same things. In Newton, the world consists of lists with bodies and joints just as in ODE, but Newton have no gravity vector in the world. The gravity has to be set in a callback-function as discussed in the chapter. The bodies in the engines have almost the same properties; one thing that differs is that in Newton, a buoyancy can be set to a body. Both bodies consist of a mass, a centre of mass and inertia.

The one main thing that differs in the engines is the collision detection system. ODE uses the “Space”-concept, where a space where objects collide is created. This is not done in Newton where the objects are connected to the collision primitives directly. In ODE the collision primitives can be created and used without a body, they have their own transform matrix, which is not possible in Newton. That is a positive thing with ODE, because the user may not want a body. Both engines use almost the same kind of collision primitives. Since there where no documentation about the constraint solver in Newton, it is hard to compare the solvers in the engines, but both engines use about the same kind of joint types. Newton has containers for rag doll and vehicles, which is a plus. Newton also has a material class where materials can be set how to interact with each other, which ODE does not have. Another feature with Newton that ODE does not have is the ability to set friction accuracy.
3.3 Runtime tests

ODE and Newton were tested in a test environment created for this purpose. The tests have been chosen to evaluate different aspects of the engine, that is gravity, friction, collisions and joints. The following chapter will describe the tests and discuss the results. The tests will be measured in standard units.

3.3.1 The test program

The test program is implemented for the sole purpose of these tests of the engines. I used a fixed timestep in the test cases since a variable amount of time in the time step can give bad results as discussed by Fiedler in [116]. Both engines are set for running on the most precise calculations and with default error correcting values.

3.3.2 Gravity test

The gravity test consists of dropping a box (size is 1.0 x 1.0 x 1.0 meters) from different heights onto a static ground. No other forces than gravity will be applied to the box (i.e. no air resistance). The time taken for the objects to touch the ground will be measured and compared between the different physics engines. These times will then be compared with the following formula (3.1) derived from Newton’s Second Law of Motion ($F = m \times a$). [05]

$$t = \sqrt{\frac{2 \times d}{g}}$$

Where $t$ is the time taken for the box to touch ground, $d$ is the distance to the ground and $g$ is the gravitation constant 9.81 m/s². Each height will be tested five times and a mean value will be calculated to ensure no faults.

---

Fig 3.1 (a) The box at starting position. (b) Box falling towards the ground
Results
This test was performed to show how close reality the engines could simulate a box falling to the ground.

Both Engines did well in the Gravity test, as can be seen in graph 3.1. As mentioned before, default settings in the engines were used. It is surely possible to tune the engines individually to get them even closer to the thin dotted line, which is the line for the formula. As can be seen below the lines all look the same (only displaced above and below). This tells us that the gravity approximations of the engines are good.

Since both engines had default settings and no other forces affected the falling box, the best engine in the gravity test is the engine that came closest to the thin dotted line. Newton was clearly the best engine at the lower heights, where it was very close to the thin dotted line. At higher heights, the ODE engine came a little bit closer to the thin dotted line and Newton drifts off a bit. By this observation, I concluded that ODE is the best engine in this test.

Graph 3.1 Results for falling box test.
3.3.3 Collision

The Collision test is for testing collision detection and detection response. This test is similar to test number seven in a test by Lander and Hecker in [A11]. Two different boxes will be dropped from 40 meters into a vertical chute formed by two large static narrow planes. The idea is that the boxes will become wedged between the planes and not move. One large box of size 10 x 10 x 10 meters with mass 5 000 kg and one small box of size 1 x 1 x 1 meters with mass five kilograms will be dropped to examine their behaviour with each of the physics engines. No other forces than gravity will be applied to the box (i.e. no air resistance). The test will be performed with three different values on the bounce coefficient. The tested values will be 0.0, 0.5, and 1.0 (0.0 is like clay, 1.0 is like a superball). All other values will be set to default. In the tests the engines will be graded from 0.0 to 1.0, where 0.0 being the worst and 1.0 the best score. The engine will get the score 1.0 if it produces a totally faultless result, no penetration whatsoever and 0.0 if the engine does not detect a collision at all. These scores are my own subjective grades.

Newton Results

**Bounce value 0.0, large box**
The box slides down on one side until it get wedged slightly penetrating the planes. When looking more closely on the box, one can notice that it is still moving very slowly down the chute. Finally it comes to a rest intersecting the planes far more than before. No wobbling or unrealistic behaviour except for the penetration.
Score: 0.6.

**Bounce value 0.0, small box**
The box falls down and hits one side, penetrating it quite much, and when it is stuck, the penetration is as for the large box.
Score: 0.5

**Bounce value 0.5, large box**
The box wobbles a bit before getting stuck, nothing strange about this since the bounce value is quite high, but then the box moves very slowly down the chute as with the previous test value.
Score: 0.5.
Bounce value 0.5, small box
The falling part is quite realistic, no visible penetration with the walls. When the box is wedged in, there is obvious penetration.
Score: 0.5.

Bounce value 1.0, large box
The box wobbles very much back and forth between the planes until it very unrealistically wedges itself between the planes, this is not so strange considering the large bounce value. The same situation appears in this case just as before, the box sliding down even more into the planes after being stuck.
Score: 0.4

Bounce value 1.0, small box
The simulation gives different behaviour each time the test runs with this bounce value. Overall the box bounces around in the chute for quite a long time until finally it get stuck, penetrating the planes a lot.
Score: 0.4

Overall performance
The problem with the box sliding very slowly down the chute into the plane, has nothing to do with the friction settings on the plane and the box or to do with the auto freeze function in Newton. It has been tested. My guess is that it is because Newton’s Collision detection system misses a few contact points. When it comes to the wobbling when the bounce has a value of 0.5 – 1.0, not very surprising since those high values on the bounce coefficient may give strange results.
Total score: 2.9
Average score: 0.483

ODE results

Bounce value 0.0, large box
Falls down the chute very realistic, not penetrating the walls at all, when getting stuck, the box never comes to a rest or gets disabled even if the box stops moving and the auto disable function is set to true. ODE prints an internal LCP error, saying that a variable s is too small, when using the most accurate stepping method. According to Russel Smith [I17] this is a symptom of things that has nothing to do with the constraint solver, but instead because the solver gets passed a nearly singular matrix (caused by coincident contact points). By changing the number of contact points the error messages can be evaded, at a cost of accuracy in collision.
Score: 0.8

Bounce value 0.0, small box
Falls down the chute quite unrealistic, starts to spin very much and penetrates the wall on the way down in the chute. When finally getting stuck, it looks good, no penetration between the box and the planes.
Score: 0.5
**Bounce value 0.5, large box**
The box bounces quite a lot before coming to a rest in the chute. All bouncing looks good, no apparent penetration of the walls. When the box is resting, also here the LCP error message appears, but no penetration.
Score: 0.8

**Bounce value 0.5, small box**
No penetration visible in this case either. About the same thing as with large box.
Score: 0.8

**Bounce value 1.0, large box**
The box never stops bouncing and eventually bounces out of the chute. Very good collision handling, no penetration visible.
Score: 1.0

**Bounce value 1.0, small box**
The box acts just like with the large box, no apparent difference.
Score: 1.0

**Overall performance**
ODE did very well in this test, box looked good when falling down the chute and when getting stuck, no penetration is visible. The test was performed with both four and eight contact points per geom, setting less than four contact points gave penetration in the planes. I concluded that four were enough for this test. On some occasions it happened that the box flew away and started spinning very much. This was because a collision vector for some reason got size 0, and therefore generated a divide by zero, setting some values to infinity, making the simulation unstable.
Total score: 4.7
Average score: 0.817
3.3.4 Friction test

This test will evaluate the physics engines friction approximations. It will be done with a box of size 1 x 1 x 1 meters sliding down an inclined plane. The friction coefficient \( \mu \) will be varied and the angle \( \alpha \) will be measured when the box starts to slide. The results will be compared with the Coulomb friction model (see formula 3.2).

Frictional force at rest [05]:
\[
F_f \leq \mu_s F_n
\]

Where:
- \( F_n \) = normal force
- \( F_f \) = friction
- \( \mu_s \) = static frictional coefficient

From figure (3.4) the following can be set up:
\[
F_n = mg \times \cos \alpha \\
F_f = mg \times \sin \alpha
\]

Maximum \( \alpha \) is when
\[
F_f = F_n
\]

This gives
\[
mg \times \sin \alpha = \mu \times mg \times \cos \alpha \Rightarrow \alpha = \tan^{-1} \mu \quad (3.2)
\]

Results

Both engines did well in this test. As can be seen in the graph (3.2) below; they both were close to the Coulomb friction model. The Newton Engine had a little bit higher threshold before letting the box slide down the plane. The ODE engine simulation was harder to set up due to ERP and CFM values (see section 3.2.1) that made the simulation act in an unpleasant way, i.e. box sliding down the plane even if the friction was set to infinity.

As can be seen in formula (3.2) above, the Coulomb friction coefficient is independent of the mass of the object sliding. When testing with a mass of 100 000 kg in ODE, it gave a significant different result from the one in the diagram below (where a mass of 680 kg is used). This was not the case with Newton. Even if ODE was closest to the formula in this test, the Newton Engine seems to simulate the Coulomb friction model more correctly and therefore Newton is the best engine in this test.
Graph 3.2. Results from the friction test.
3.3.5 Constraints

This test will put engines ability to hold constraints to the limit. The test is similar to the test in the Seugling/Rölin Thesis [A12]. A sphere (radius 0.5 m) will be connected to a static plane by a hinge joint like in figure 3.5a. The sphere will be dropped from a horizontal position ten meters from the centre of the hinge and go into a pendulum motion. The test program will measure the distance between the plane and the sphere at different masses. The mass will be changed from 500 kg to 1 000 000 kg in ten steps. The idea with the constraint is that the distance should be constant during the whole test.

![Figure 3.5: (a) Sketch of the Constraints test (b) Newton Engine during test run.](image)

Results

In this test, the results varied a lot between the engines. A graphic illustration of the results can be seen in graph 3.3a and 3.3b on the next page. A negative distance error means that the sphere was closer than ten meters to centre of the plane.

The Newton engine holds the distance error almost constant (negative values around -0.01 to -0.02 and positive values in the region of 10^-6 m). Even when the highest mass was tested, the Newton Engine performed well. The pendulum looked realistic with all masses and the sphere decreased in period time physically correct.

With the ODE engine, the distance error increased with the mass. This made the pendulum period time quite long and it did not look very realistic. The results from this engine are with ERP- and CFM-values not specified. The test was performed with different ERP and CFM values to see the differences. Only by specifying the ERP and CFM to zero the distance error was cut in half and the period time for the pendulum looked more realistic. With ERP value set to 1.0, the distance error was almost nonexistent (in the region 0.001 m) but the period time increased considerably. However, setting the ERP to 1.0 is not recommended due to internal approximations according to the user guide for ODE at [I09]. Increasing the CFM value only increased the distance error towards infinity due to that the CFM value is used for violating the constraints.

The outcome of this test was that Newton showed a better result than ODE, making Newton the best engine to hold constraints.
Graph 3.3 (a) Results with Newton Engine (b) Results with ODE
3.3.6 Tests Conclusion

These tests have evaluated only a small portion of what a physics engine can do. Both engines consist of lot more than just these features tested here. The tests were developed in discussion with Craft Animations. The first test, the gravity test, went well for both engines. Here, ODE performed closer to the formula and was chosen as the best engine. In the collision test was ODE significantly better than Newton in finding the contact points and it was possible to set how many contact points each object should have. In the friction test, Newton was chosen as the best engine. This was because ODE friction model was dependent of the mass, which according to formula (3.2) should not be the case. The constraint test confirmed results from [A12]. Newton was significantly better in this test.

After four tests, Newton and ODE are in a draw. But this will not end in a draw. Finally, Newton was chosen as the best engine and that is because of good on documentation and other functionalities besides the ones tested here. Newton has also several good tutorials on the website [I13]. Other functionalities such as vehicle and rag doll containers made Newton the best engine of the two.

After seeing these results, I draw the conclusion that the engines are similar in performance and that they are better in some areas and not so good in other. These tests where not performed only to choose the best engine, but more to show how close to reality the engines could be and how accurate they are according to constraints. The tests have shown that the simulations are far from full physics, and that these engines cannot really be used in other areas than games where speed is more important than accuracy. In the next chapter, two engines are presented, where more accurate simulations are possible to simulate.
3.4 Two commercial physics engines

There are two big commercial physics engines on the market today. Havok and AGEIAs PhysX. Both are put under the magnifying glass in this chapter.

3.4.1 AGEIA PhysX

AGEIA Technologies Inc (started 2002) is dedicated to deliver dynamic interactive realism to the next generation games. Their flagship solution is the AGEIA PhysX, the first dedicated physics engine and physics processor. Currently there are six titles supporting the PhysX physics engine, e.g. “CellFactor: Combat Training” and “Tom Clancy's Ghost Recon Advanced War fighter”. For complete list and upcoming games visit the AGEIA website at [I10].

PPU – Physics processing unit

According to the article by Nicholas Blachford [I18] the PhysX PPU chip is made up of three engines with its own memory controller, PCI interface and various I/O ports. The three engines are:

- DME – Data Movement Engine. Is responsible of moving data in and out of the memory.
- FPE – Floating Point Engine. Performs all actual physics calculations. All the data processing is done on 32 bit values stored in 16 floating point registers or 8 integer registers.
- PCE – PPU Control Engine. Is a conventional RISC processor, its job is to manage the DME and FPE.

PhysX SDK

The current version, at time of writing, of PhysX SDK is 2.5, but since it is only available to established developers who have a license with AGEIA, this chapter will give an overview of version 2.4 of the SDK.

The following information about the PhysX SDK is fetched from the AGEIA PhysX SDK documentation [I10].

The PhysX interface is based on C++ and is effectively an abstract base class. The SDK is implemented as a hierarchy of classes. It is possible to define your own memory management, which can be useful when special memory management is required. Current simulation state can be saved and loaded into a portable binary format (ascii is supported for write only, used for debugging) using a serialization class library.

Datatypes

PhysX SDK uses size specific type definitions that provide a certain degree of portability. It has representations for

- 3-vector: float or double precision.
- 3x3-matrix: row-major or column-major.
- 3x4-matrix: a complete affine transformation composed of a combination of rotation and translation matrix.
- Quaternion.
There are many type and format conversion methods in the math classes for easy use with user-defined classes. The representation is currently configured to use (32 bits) floating point scalars. No particular real-world units are used in the SDK. The user can define quantities (like mass, length, time) to be in any units wanted.

**Rigid bodies**
A rigid body can be static or dynamic and it’s possible to put a body to sleep (disabling it). Other properties of a rigid body see table 3.6.

<table>
<thead>
<tr>
<th>Linear</th>
<th>Angular</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mass (scalar)</td>
<td>Inertia(3-vector)</td>
</tr>
<tr>
<td>Position (3-vector)</td>
<td>Orientation(quaternion or 3x3 matrix)</td>
</tr>
<tr>
<td>Linear Velocity (3-vector)</td>
<td>Angular Velocity(3-vector)</td>
</tr>
<tr>
<td>Force (3-vector)</td>
<td>Torque (3-vector)</td>
</tr>
</tbody>
</table>

**Joints**
PhysX SDK supports only pair-wise constraints. When working with the PPU, only the D6-joint (see below) is supported and there is a limitation on 2048 joints per scene. The types of joints supported are listed in table 3.7.

<table>
<thead>
<tr>
<th>Joint type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spherical joint</td>
<td>Similar to a Ball-Socket-joint.</td>
</tr>
<tr>
<td>Revolute joint</td>
<td>Similar to a Hinge-joint</td>
</tr>
<tr>
<td>Cylindrical joint</td>
<td>Similar to a slider-joint.</td>
</tr>
<tr>
<td>Prismatic joint</td>
<td>Similar to a slider-joint, but with no relative rotation.</td>
</tr>
<tr>
<td>Fixed joint</td>
<td>No relative movement between connected bodies.</td>
</tr>
<tr>
<td>Distance joint</td>
<td>Maintains a distance between two points on two bodies.</td>
</tr>
<tr>
<td>PointInPlane joint</td>
<td>Constrains a point on one body to move along a plane attached to another body.</td>
</tr>
<tr>
<td>PointOnLine joint</td>
<td>Constrains a point on one body to move along a line attached to another body.</td>
</tr>
<tr>
<td>Pulley joint</td>
<td>Simulates a rope over two pulleys.</td>
</tr>
<tr>
<td>D6 joint</td>
<td>6 degrees of freedom. A general constraint, user can define linear and rotational degrees of freedom.</td>
</tr>
</tbody>
</table>

**Breakable joints**
The joints can be set to break when they are exposed to a certain amount of force.
Joint projections
Joint projections are for correcting large joint errors. When small joint errors appears, small correcting forces can be applied to try to reduce the errors. When the errors are too big the projections come into play. If the error distance is larger than a certain value the position of the object will be projected (or changed) to directly fix the joint error. This is only supported by the D6-joint, Revolute-joint and Spherical-joint.

Constraint solver
The PhysX constraint solver is an iterative solver and it iterates over all constraints. The more times it iterates at each time step the more accuracy is achieved.

Collision detection
The PhysX SDK has several primitive shapes used for collision detection.
- Box
- Capsule
- Convex (with a maximum of 256 polygons)
- Heightfield (works in a similar way as triangle mesh but is regular, rectangular grids)
- Plane
- Sphere
- TriangleMesh
- Wheel (for simulating car wheels, see below)

Not all shapes can interact, for a complete list see the SDK documentation on the Ageia website [I10].
The SDK supports:
- Broad phase collision detection (three checks to see if two objects are potentially colliding and then a contact determination is performed).
- Continuous Collision Detection (CCD).
- Triggers, which mean that a user-defined event is triggered when other shapes pass through a shape, compare with Phantom objects in the Havok SDK.
- Ray casting

Ray casting
Several functions supported:
- raycastAnyBounds, rayCastAnyShape: The fastest function, returns true if hits any AABB.
- raycastClosestBounds, raycastClosestShape: Returns the closest stabbed shape.
- raycastAllBounds, raycastAllShapes: Is the most complex function, returns all stabbed shapes, distances and intersection points.
Ray cast wheel shape
There are two ways to create vehicles with the PhysX SDK.
1. Create spherical wheel bodies and use joints to attach them to a rigid body representing the chassis and add further rigid bodies to represent steering and suspension.
   Advantage: You can create complex steering and suspension functionality.
   Disadvantage: Lack of sophisticated tire friction model and using several rigid bodies and joints is costly and inaccurate (joint errors).
2. A raycast car: Vehicles are composed of a single rigid body with special ground contacts to simulate wheels. The disadvantage is that the mass and inertia of the wheels do not influence the motion of the car automatically; it has to be added as a post process.

Other PhysX SDK features
The SDK supports:

- Fluids (allows the simulation of liquids and gases using a particle system)
- Cloth (stretching and bending constraints)
- Advanced Character Controller (Character Control and Character Interaction)

Pure hardware supports among other things (more details in the API references):
- Maximum three concurrent scenes, 2000 rigid bodies and 2048 D6-joints in each scene
- Simple rigid bodies
- Convex, box, sphere and capsule dynamic shapes
- Convex, box, sphere, capsule, plane and non-convex triangle meshes static shapes.
- Forces and impulses applied to dynamic shapes
- D6-joints (no other joints)
A Test of Ageia PhysX

In Darren E. Polkowskis article from July 2006 [I19], a test of the AGEIA’s physics engine is performed giving not so impressive results. In the article they want to know if there is a difference in 3D graphics experience with and without a physics accelerator card. To do this they took a whole spectrum of graphics cards (see [I19] for complete list and system hardware). The results can be seen in figure 3.6, showing the average frames per second. This is not so good results for PhysX engine.

![Figure 3.6 Test of the Cell Factor game with and without the PhysX Card [I19]](image)

Ageia commented on the test in the article and stated that several functions (such as cloth, cloth tearing and fluid simulations) were disabled when starting the game without the PhysX PPU, in the version of the game that was tested.

The test was run again with the new version of the game. This gave different results. This time cloths were simulated much better with PhysX enabled and without the card enabled the frame rate came to a crawl. The article states as a conclusion that it’s still difficult to determine what the PPU is doing. On one hand, hardware acceleration is needed for cloth simulation and it makes sense to offload this from the CPU to a PPU or a graphics card calculating effects physics. On the other hand, the game and PhysX did not give results convincing the article author.

Another test of the PhysX engine is performed in Tino Kreiss article from June 19 2006 [I20]. Here the results are not very impressive either, according to the author there simply are to many irregularities among the physics effects when testing the PhysX supported game Ghost Recon Advanced Warfighter. The author also says “As things stand now, we would venture to say that the Havok physics engine has a better chance of becoming established than Ageia's PhysX”.
3.4.2 Havok

Havok offers a collection of middleware products for game realism and interactivity. This chapter will look at the Havok Game Dynamics SDK. There are several games today that use this Havok physics engine, to mention a few popular games: “HalfLife 2”, “F.E.A.R”, “Halo 2”. [I21]

Havok SDK
Since the SDK only is available for registered developers, no access to the SDK was given. Therefore all the following information about the SDK is taken from the Havok Game Dynamics SDK product overview at [I22]. The SDK supports several platforms and all commonly used compilers.

Simulation
The SDK provides a set of functions to allow an easy management of the scene and allowing a large world with a large number of dynamic objects. All objects can be created and deleted at runtime and all physical objects may be cloned giving a very efficient memory usage. Several ODE solvers are provided including Euler, Midpoint, Runge Kutta 4 and 5, and Back Euler. The SDK has a fully featured math library with support for

- Vectors
- Matrices
- Quaternions

Rigid bodies
All bodies have properties like mass, elasticity, friction and restitution that can be changed at runtime. Bodies that are not moving are removed automatically from the simulation (tagged as “inactive”).
Joints
All joint properties are editable in real-time so it’s possible to make and break the joints at will in run-time. For most joints you can specify the limits of the relation motions and orientation of the constrained objects. Joint friction is also supported. The joint types are listed in table 3.8.

<table>
<thead>
<tr>
<th>Joint types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point to Point</td>
<td>One point on one object is constrained to one point on another object.</td>
</tr>
<tr>
<td>Point to Nail</td>
<td>One point on an object is constrained to a point in space.</td>
</tr>
<tr>
<td>Hinge</td>
<td>Allows one object to spin around the axis of another object. (As described in 2.2.2).</td>
</tr>
<tr>
<td>Rag Doll</td>
<td>A special constraint designed to match the sort of connections that exist between limbs in humans.</td>
</tr>
<tr>
<td>Car wheel</td>
<td>A special constraint to make it easy to construct vehicles.</td>
</tr>
<tr>
<td>Stiff Springs</td>
<td>A faster alternative to point-to-point constraints</td>
</tr>
<tr>
<td>Springs</td>
<td>Classic spring with control over rest length and damping.</td>
</tr>
<tr>
<td>Dashpots</td>
<td>Similar to springs but act on velocities of objects, not forces.</td>
</tr>
</tbody>
</table>

Collision detection
Havok has developed a mid-phase collision detection system based on Memory Optimized Partial Polotype (MOPP) Technology (used for compact representation and large collision meshes).

Collision primitive types supported by Havok SDK are:
- Spheres
- Planes
- Convex
- Concave objects
- Fixed polygon soups, used for geometry that never moves (e.g. landscapes)
- Optimized or not optimized heightfields.
- User defined collision primitives.

Objects which are included in the collision detection but do not cause collisions, great for making sensors, is in the Havok SDK called Phantom objects.
Vehicle dynamics
Havok has a vehicle dynamics module made up of a core vehicle physics system with ray casting for wheels and a wheel friction model. On top of this core a series of game specific modules (available with source, i.e. user replaceable). These modules are:

- Transmission
- Engine
- Suspension
- Gameplay
- Aerodynamics
- Steering

With each of these modules, several parameters are provided which gives the user access to over 100 parameters to tweak the vehicle.

Deformable bodies
Havok provides fast cloth, soft body and rope dynamics with control over several parameters based on deflectors (a specialized collision detection system for very fast and accurate collision detection with specialized objects, focus on cloth/soft body and rope).

Character Controller
Havok supports a basic character controller with walk, jump, run and climb actions.

Havok 4.0
Havok released version 4.0 in July 2006 which added two new products to their software solution – Havok Behaviour and Havok FX. Havok Behaviour allows physics based interactive character behaviour. Havok FX enables physics processing to be done on NVIDIA and ATI GPUs (Graphics Processing Unit). The new SDK is compatible with the upcoming Sony Playstation 3 and Microsoft’s 64 bit Windows operating system. [I21] No test of Havok 4.0 has been found at the time of writing.

3.4.3 Conclusion
In this chapter, two commercial engines, PhysX and Havok, where looked at more closely. The two engines have similar features, such as deformable bodies, vehicle- and character support. Collision primitives are also quite alike but they name the primitives differently. It is the same with the joint types. At the time of writing, there are no tests found between the two engines, so it is difficult to say how good they are in a comparison test. Havok was tested against some other engines in the article by Hecker and Lander in [A11], where Havok performed generally strong, but the constraint system did not impress Hecker and Lander.
4 Engines of the future

Even though the test of Ageias physics engine PhysX (in section 3.4.1) did not give such good results, I personally believe that we are only in the beginning of physics processors and that some sort of hardware for simulating full physics in simulations and games are inevitable. As discussed by Justice in [I23], the CPU is good for some things and not so good for other things and when it comes to physics calculations, the CPU is fairly slow when compared to other more focused solutions. This is why we see limited worlds in terms of physics in our games today.

As said in [I24], one of the most convincing arguments for using a discrete physics engine is the ability to create new physics effects, that will increase realism in games. According to Hachman and Cross [I25], Microsoft has stated that

"physics is a key part of the next generation gaming experience, bringing increased realism, greater immersion and more interesting experiences."

The article continues and says that Microsoft is currently working on adding physics support inside their DirectX application interface, called Direct Physics, and although Microsoft hasn’t called out to use Ageia’s technology, the company has licensed the Ageia SDK. Havok, on the other hand, has optimized its solution to run on graphics chips, which according to [I25] is the road Direct Physics will take. The development of the Direct Physics will be interesting to follow.

Most physics engines in this thesis are quite limited to what they can perform. The only physics engine that really can stand up to Ageia’s PhysX is the Havok engine. As mentioned in section 3.4.2, Havok and Nvidia has a engineering partnership to develop physics simulations that runs on NVIDIA’s GPUs and it is amplified with their Scalable Link Interface (SLI, NVIDIA’s method for connecting 2 or more video cards together to produce a single output [I21]). Maybe this new partnership can compete with PhysX. Only the future can tell.
5 Design & Implementation

5.1 Introduction

As seen earlier in this thesis, there is a long line of physics engines that can be used for physics implementations. Craft Animations wanted to get an overview of a few of them and an implementation example of one of them using this interface. My task was to write the specification for this interface. It should handle a portion of the functionality that a complete physics engine has and I should implement it to show how the interface could work with a physics engine. After the overview of the engines in chapter 3, I chose to integrate the engine Newton with the interface. This was because Newton uses many functions common among other engines and the engine has quite good documentation for easy implementation.

The idea with the interface was not to make a wrapper where several engines could be used at a time. Therefore it is not possible to change physics engine in real-time. Instead, the idea was to make an interface where one can choose an engine to use, and then later be able to change to another engine without needing to change the implementation considerably. The design diagrams and an interaction diagram can be found in Appendix A and B respectively.
5.2 Specification of requirements

Craft Animations had a list of requests for functionality in the interface. This is what was specified. The interface should have the ability to

- create and destroy objects dynamically.
- save and restore a specific state as XML.
- give all objects a unique identification and a name. The id and name should be saved if the object is saved. If an object is loaded into a scene where the id already exists, the new object should replace the old one.
- represent orientation with transformations.
- be easy to extend. It should be an open interface, where it is easy to add features to support more collision primitives and effects like water etc.
- specify some type of configuration for how the system is to be run. For example with larger stepping and faster system or a more demanding but more accurate stepping.
- focus the state retrieval on performance.
- express units in SI standards.
- let objects have parents and children.

When it came to other functionality, I chose to implement functions from the most common basic functionality in ODE and Newton and the other examined engines. Here is my specification for the interface. The interface should consist of

- a world in which bodies, joints, materials etc are created.
- functions to set time steps, gravity and accuracy in the world.
- an physical object-class that consist of a body part that can be exerted to forces etc, and a collision part consisting of a collision primitive that can be used for the collision detection.
- functions for transformations, forces, torque, angular- and linear velocity and mass in the object.
- functions for enabling and disabling both the body part and the collision part of the object.
- functions for offsetting the collision-part transformation from the body part in the object.
- materials as fields in the objects that can interact with other materials.
- collision primitives of type box and sphere.
- a joint of type Hinge.

5.3 Implementation

All code was written in C++ using the Visual Studio 2005. The implementation phase was done during a period of 8 weeks. During this time there were some issues with the implementation where some were solved in discussion with Craft Animations.

One problem was noticed when saving a state. The floating numbers that were saved in XML were not interpreted the same way when loaded, a rounding off error occurred.
This was solved by saving the numbers as hexadecimal numbers instead of floating point numbers. Craft Animations provided a hex-class, which converted floating point numbers to hexadecimals and vice versa.

One requirement was changed during implementation. That was the requirement that an object that is added to a world where the same id already exists, should replace the old object. The new object should now only change to another id but still have the same properties.

5.4 Requirement fulfilment

As discussed (in section 5.2) there were some requirements set up by Craft Animations and some requirements set up by myself. Here is how they are fulfilled.

**Ability to create and destroy objects dynamically:** Depends on the physics engine used as a base, but the interface created fully supports it.

**Ability to save and restore a specific state in XML:** Fully supported by the interface using the XML class provided by Craft Animations.

**All objects should have a unique identification and a name:** A unique identification number is given each object created in a world, and a name can be given to it by the user. Both are saved with the object. When loading, if an object exists in the scene with the same id, the new object will be given a new id.

**Orientation should be based on transformations:** Fully supported by the interface.

**It should be an open interface:** The interface is easy to extend with more features such as fluid support. It is also easy to add more collision primitives and joint types etc.

**It should contain some type of configuration for how the system is to be run:** The interface has support for this with the `setSolverAccuracy`- and `setFrictionAccuracy` methods (see Appendix A).

**The state retrieval should focus on performance:** The states saving functionalities is based on XML and there may be some more optimal structure for saving the states, but in discussion with Craft Animations the XML structure is sufficient.

**All units are in SI-units:** The units in the interface can be any kind of unit standard. It has to be implemented specifically for the engine used as base.

**Objects should have parents and children:** Fully supported.

All requirements set up by myself (see section 5.2) were implemented successfully as well.
5.5 Conclusion

All functionality in this interface depends on the engine used. All engines may not support all functionality. The interface is easy to extend to support more functionality than just these in the interface. As mentioned before this implementation supports only a small portion of what a complete physics engine might support, but it shows what a interface might look like, kind of a “proof of concept” and it is a basis for future work. Due to limited time the interface could not be tested in a test program other than the one created for the runtime tests. The result was just as when using the Newton engine directly.
6 Conclusions

This report gives an explanation to what a physics engine consist of and how it works. I have not concentrated so much on algorithms used in a physics engine or how the algorithms work. For the interested reader, algorithms and implementation code can be found in some references [01][02][06], to name a few. Instead, I have given an explanation to what the physics engine do that may be easy to understand for most readers. There are many ways to implement a physics engine and I present an overlook of how a typical physics engine work. In the beginning of the thesis, there were some explanations to some basic terms used in the thesis and important terms in a physics engine. The thesis continued with what the physics engine do, giving an overview how collision detection works with names of some algorithms for the interested reader. An important part in the physics engine is the constraints, used for collision response and joints. This part of the thesis was for getting a good basic knowledge of physics engines and to give a better understanding of the rest of the thesis. Since a physics engine can simulate more than just rigid bodies, I described in short how deformable objects can be modelled and how particle systems work; this was to describe what a physics engine is capable of.

I chose a few physics engines and made a short overview of them, showing what their functionalities consisted of and chose two of them for a closer look. The two chosen engines where ODE and Newton Game Dynamics. These engines was put through some tests, to see how they performed and the results where interesting.

The biggest physics engines on the marked today, PhysX and Havok, were examined, also with interesting test results. The future for physics engines looks bright.

When it comes to the other engines in this thesis, it differs quite much in how they are implemented, how they for example handle collision objects and body objects. This is one reason to why integration between the engines was hard. An interaction between engines using for example a solver from one engine and joints from another engine and collision detection system from yet another engine was not possible. With the ODE engine there was a possibility to use an external collision detection system, so in that case it would be possible to use ODE as base engine and use for example Newton for the collision detection system.
7 References

7.1 Figures

[F01] Torque (Moment) Glenn Research Center. NASA Official: Tom Benson Last
   (visited 2006-11-21)........................................................................................................... 3

[F02] Spark Notes, SAT Physics: Types of Forces,
   http://www.sparknotes.com/testprep/books/sat2/physics/chapter6section4.rhtml
   (visited 2006-07-07)........................................................................................................... 4

[F03] Figure inspired by the website "GJK vs Raycasting, GameDev.Net Discussion Forum",
   http://www.gamedev.net/community/forums/topic.asp?topic_id=333503&cforum_id=20&gforum_id=0
   (visited 2006-11-16) ................................................................................................. 7

[F04] Robust Computation of Swept Volumes, Stefan Schirra (responsible of page).
   (visited 2006-11-17)........................................................................................................... 7

   (visited 2006-11-09).............................................................................................................11

   (visited 2006-11-21).............................................................................................................13

[F07] Matt Mason, Mechanics of manipulation, Lecture 16 , page 9,
   (visited 2006-11-22).............................................................................................................29
7.2 Books


   ISBN 1-55860-740-4 ................................................................................. 3, 4, 5, 8, 12, 13, 49


   ISBN 91-44-03227-7 .................................................................................. 2
7.3 Internet

(visited 2006-09-20)........................................................................................................... 1

http://www.newtondynamics.com/downloads/NewtonWin-1.53.zip  
(visited 2006-08-15)........................................................................................................... 2


(visited 2006-11-08)........................................................................................................... 4

[I05] Havok Physics Primer version 1.0 (2000/2001), Havok.com Inc: http://director-  
online.com/havok/downloads/docs/HavokPrimer.zip  
(visited 2006-07-18)........................................................................................................... 5, 6, 10, 13

(visited 2006-11-14)........................................................................................................... 5, 19

(visited 2006-11-16)........................................................................................................... 6, 22

[I08] Russell Smith (February, 2002) "How to make new joints in ODE",  

(visited 2006-11-22)........................................................................................................... 9, 15, 18, 31


(visited 2006-11-21)........................................................................................................... 16

(visited 2006-11-21)........................................................................................................... 16

[I13] Newton Game Dynamics Physics engine, website,  
http://www.newtondynamics.com (visited 2006-11-21) ............................................ 16, 21, 33

(visited 2006-11-21)........................................................................................................... 17

[I16] Glenn "Gaffer" Fiedler, Game Physics: Fix Your Timestep!


http://www.blachford.info/computer/articles/PhysX2.html (visited 2006-09-05) ................................................................. 34

[I19] Darren E. Polkowski (July 19, 2006), Is Ageia's PhysX Failing?
http://www.tomshardware.com/2006/07/19/is_ageias_physx_failing/page9.html (visited 2006-09-05) ................................................................. 38

[I20] Tino Kreiss (June 19, 2006), Can Ageia's PhysX Card Bring Real-World Physics to Games?
http://www.tomshardware.com/2006/06/19/can_ageia/index.html (visited 2006-09-05) ................................................................. 38


[I22] Havok Game Dynamics SDK, Telekinesys Research Ltd (2002),

[I23] Brent Justice (March 20, 2006), NVIDIA SLI Physics Tech Preview,
http://enthusiast.hardocp.com/article.html?art=MTAwNSwxLCxoZW50aHVzaWFz (visited 2006-11-18) ................................................................. 43

[I24] Sean Pelletier (March 22, 2006) AGEIA PhysX Launch - GDC Event Coverage, ................................................................. 43

[I25] Mark Hachman and Jason Cross (June 20, 2006), Microsoft Planning To Add Physics Support To DirectX, ................................................................. 43

[I26] John Simmons (2001), Creating a Unique String using a UUID,
7.4 Papers & articles


[A02] Gino Van Den Bergen (June 2004), "Ray Casting against General Convex Objects with Application to Continuous Collision Detection" Playlogic game factory. .............. 6


[A04] Andrew Witkin (1997) "Physically Based Modeling: Principles and Practice,
(visited 2006-11-21) ................................................................................................................. 8


and Practice, Differential Equation Basics”,
(visited 2006-11-22) ..................................................................................................................10, 11


The eurographics association (2005) ..................................................................................12, 13


http://www.gamasutra.com/features/20000913/lander_03.htm
(visited 2006-08-15) .................................................................................................................. 26, 41
8 Appendix A

8.1 Design

The design of the interface will be presented in detail here. The UML-diagrams show most of the methods and the derived classes show examples of how it could look. All derived classes have to be implemented specifically for that engine. Here, only the Newton-derived classes are implemented, consisting of Newton specific code. The derived classes with no names (shown as …), are to show that there can be infinitely many engines integrated with the interface. In some methods a Vector3-class, a Matrix-class, a XML-class or a Quaternion class has been used, these classes are all provided by Craft Animations.

8.1.1 PhysicsInterface

The PhysicsInterface (fig 9.1) is the part of the interface that creates all objects in a world and the worlds themselves. The user starts using the interface by choosing which engine to use. For example by simply writing:

```cpp
PhysicsInterface* myInterface = new NewtonInterface();
```
Here, the user tells the interface to use the Newton Dynamics Engine and its functionality. The interface creates worlds (see below) and saves a pointer to them in a reference list. The interface class is mostly used internally by the worlds to create all sorts of objects, materials and joints using the `create`-method.

### 8.1.2 Entity

The Entity class (fig. 9.2) is the base class for all objects created. This class holds a unique identity string, a name of the object and what type of object it is. The unique id string is created by creating uuid-strings with a method from [I26] by John Simmons. The Entity class is also the holder of the `setState`- and `getState`-methods, which are inherited by all objects and used for the ability to set and get a state during runtime, without considerably slowing down the simulation. It is also possible to save states to an XML-file, and restore the state from that file again using the `saveState` and `restoreState` methods. An XML class was provided by Craft Animations for easy XML-handling. All objects inherit the `destroyMe`-method, which destroys an object (deleting all pointers, references etc.).

<table>
<thead>
<tr>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>- id : string</td>
</tr>
<tr>
<td>- name : string</td>
</tr>
<tr>
<td>- type : int</td>
</tr>
<tr>
<td>- subtype : int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>+setState(element : XML::Element &amp;) : void</td>
</tr>
<tr>
<td>+getState() : XML::Element</td>
</tr>
<tr>
<td>+saveState(filename : string) : void</td>
</tr>
<tr>
<td>+restoreState(filename : string) : void</td>
</tr>
<tr>
<td>+setName(name : string) : void</td>
</tr>
<tr>
<td>+getName() : string</td>
</tr>
<tr>
<td>+setId(id : string) : void</td>
</tr>
<tr>
<td>+getId() : string</td>
</tr>
<tr>
<td>+setSubtype(subtype : int) : void</td>
</tr>
<tr>
<td>+getType() : int</td>
</tr>
<tr>
<td>+getSubtype() : int</td>
</tr>
<tr>
<td>+destroyMe() : void</td>
</tr>
</tbody>
</table>

Fig. 9.2 The Entity class.
8.1.3 World

Create a world, write:

```
WorldObject* world = myInterface->createWorld();
```

It is also in the world where the time is stepped, gravity is set here and the accuracy of the engine as well. With the `setSolverAccuracy`-method, the user can set how accurate the solver should be. The integer-value can be used differently for each engine for defining accuracy. The `setFrictionAccuracy`-method is used for setting models for friction. These methods can be used in different ways for each engine.
getWorld() returns a world object specific for each engine, for example Newton’s NewtonWorld*. The createMaterial-method creates a material with a name; it can be used as follows:

```cpp
Material* wood_material = world->createMaterial("wood");
Material* steel_material = world->createMaterial("steel");
```

With these materials you can set an interaction between them, using the setMaterialInteraction-method.

```cpp
MaterialInteraction* wood_steel =
    world->setMaterialInteraction("wood_steel",
                              wood_material,
                              steel_material,
                              static_friction,
                              kinetic_friction,
                              bounce,
                              softness);
```

Methods like findMaterialPtr(), findObjectPtr() and findJointPtr() are used when you want to find a pointer to a object, material or joint, when all you have is the id string of it (used internally by the interface when setting a state of an object).

To create a collision object, that is to be used for collisions, write for example the following:

```cpp
CollisionObject_Box* collisionObject =
    world->createCollisionObject_Box(size);
```

As can be seen in the diagram above, there is only support for creating a collision box and sphere, but the interface is easily extendable to support other collision objects.

An object can now be created as follows:

```cpp
Object* myObject = world->createObject(parent, collisionObject, 1);
```

This creates an object called myObject and specifies a parent to the object, connects it to the collisionObject created before and the last parameter, the 1, tells the world to create a BodyObject and connect it to myObject.

### 8.1.4 Object

The Object (fig 5.4) is created by the world as seen above and this moves and collides with other objects. It contains a BodyObject, which moves and is exposed to forces and velocities, and it contains a CollisionObject, which interacts and collides with other objects. In this implementation, the object can be manipulated according to force, torque, transformation, linear and angular velocity.
Fig 5.4. The Object base class with derived example classes.
It’s easy to extend the object interface with other functionality. The body object and the collision object (geom) can be enabled and disabled independently from each other with e.g. enableBody- and enableGeom-methods. The collision object can be transformed independently from the body and it can be offset from the body using setGeomTransform- and setGeomOffsetTransform-methods.

### 8.1.5 BodyObject

**Diagram**

The BodyObject is the object that moves in the world and is exposed to forces etc. The derived classes only consist of a body object that is specific for the engine being used. The BodyObject is used internally by the Object-class.

### 8.1.6 CollisionObject

**Diagram**

The BodyObject with derived classes.

The CollisionObject base class with derived example classes.
The CollisionObject is the object that interacts and collides with other CollisionObjects. The derived classes contain the collision primitive specific for each engine. It must be connected to an Object to be used. With some engines, like Newton, the Object must have a BodyObject connected to it for the CollisionObject to work. The CollisionObject is mostly used internally by the Object. It is created as seen above in the world section.

8.1.7 Material

Fig 5.7. The Material base class.

The material is created by the world as seen before. It contains a reference list to all material interactions and objects using it. When a material is destroyed, both the interactions and objects using it are notified and their pointers are set to NULL.
8.1.8 MaterialInteraction

The material interactions specify what happens when two materials collide. As seen above the user can set bounce, friction and softness values for when materials meet. The MaterialInteraction class contains pointers to the two materials and also id strings with the names of the materials if the materials don’t exist when the material interaction is restored from a file.
8.1.9 Joint

Fig. 5.9 The Joint base class with derived example classes.
The hinge joint interface looks as above. In the current interface, it is only support for creating a hinge joint, but this is also easy to extend to support other joints. It contains pointers to the objects connected in the joint and vectors for the pin and anchor points. A joint is created by writing:

```c
Joint* hingeJoint = world->createHingeJoint(jointPin, jointPivot, parent, child);
```
9 Appendix B

The Interaction diagram
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Uphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Henrik Hansson