

Evaluation of Physics Engines and Implementation of a Physics Module in a 3d-Authoring Tool

Axel Seugling, Martin Rölin

March 6, 2006

Master's Thesis in Computing Science, 2*20 credits
Supervisor at CS-UmU: Anders Backman
External supervisor: Kalle Jalkanen, Markus Häggqvist
Examiner: Per Lindström

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
SWEDEN

Abstract

This master thesis will evaluate physics engines that are free for non-commercial use. The evaluation consists of a theoretical comparison between physics engines and nine run time tests. Important aspects of a physics engine are tested, e.g., friction, scalability and stability. The evaluation will be used to select a suitable physics engine to use in a plugin module for a 3d-authoring tool.

The results have shown that a free physics engine is just as good as a commercial one in many situations. However, looking at all tests that have been made it is clear that the commercial physics engine performed best.

Contents

1	Introduction	1
1.1	Background	1
1.2	Thesis description	2
1.2.1	Goal	2
1.2.2	Method	2
1.3	Report outline	2
1.4	Related work	3
2	Background	5
2.1	Virtools	5
2.1.1	Behavioral objects	6
2.2	Terminology	8
2.2.1	Physics engine	8
2.2.2	Object representation	9
2.2.3	Friction	9
2.2.4	Elasticity	9
2.2.5	Constraints	9
2.2.6	Energy	9
2.2.7	Simulation methods	10
3	Design and Implementation	11
3.1	Related Work	11
3.2	Requirements	11
3.2.1	Functional Requirements	11
3.2.2	Non-Functional Requirements	12
3.2.3	Requirements for testing	12
3.3	Design	12
3.3.1	Overview	13
3.3.2	OpenPhysics	13
3.3.3	Connection	16
3.3.4	Debug	16

3.4	Implementation	16
3.4.1	Overview	17
3.4.2	OpenPhysics	17
3.4.3	Connections	18
3.4.4	Debug	19
4	Test and evaluation of physics engines	21
4.1	Introduction	21
4.1.1	Description	21
4.1.2	Related Work	21
4.2	Product Research	23
4.2.1	Requirements	23
4.2.2	Available Physics engines	23
4.2.3	Evaluation of physics engines	24
4.2.4	Result	30
4.3	Runtime Tests	32
4.3.1	Testing dry friction	32
4.3.2	Gyroscopic Forces	35
4.3.3	Bounce	37
4.3.4	Stability of constraints	40
4.3.5	Basic accuracy	41
4.3.6	Scalability of constraints	44
4.3.7	Scalability of contacts	47
4.3.8	Stability of piling	48
4.3.9	Testing complex contacts	49
4.4	Final result	52
4.5	Discussion	53
5	Results	55
5.1	Accomplishment	55
5.1.1	Preliminaries	55
5.1.2	Software Development	55
5.1.3	Evaluation and Testing	55
5.2	Requirements fulfillment	56
5.2.1	Functional Requirements	56
5.2.2	Non-functional Requirements	56
5.2.3	Requirements for testing	57
6	Conclusions	59
6.1	Discussion	59
6.2	Limitations	59
6.3	Future work	60

7 Acknowledgments	61
References	63
A User's Guide	65
B UML diagrams	73

List of Figures

2.1	Virtools plugins structure	6
2.2	A simple building block in Virtools	7
2.3	An example behavior graph	8
2.4	A script	8
3.1	Overview of the plug-in software	13
3.2	Overview of OpenPhysics.	14
3.3	DLL design	15
3.4	UML over the NewtonConnection	16
3.5	Simulation loop	17
4.1	First friction test	33
4.2	Coulomb friction curves	34
4.3	Result from the anisotropy test	34
4.4	Error in Angular Momentum	36
4.5	Error in Rotational Energy	36
4.6	Screenshot of the bounce test part 1.	37
4.7	Bounce results with different values of restitution.	39
4.8	The pendulum setup	40
4.9	Error in distance	41
4.10	Period time error	42
4.11	Energy	43
4.12	The trajectory of the pendulum in the x-axis	44
4.13	Chain of ball socket constraints	44
4.14	Constraint error	45
4.15	Average time per constraint	46
4.16	Pile of boxes	47
4.17	Time to solve contacts	48
4.18	Novodex	49
4.19	ODE	50
4.20	Newton	50

4.21	Novodex	50
4.22	ODE	50
4.23	Newton	51
4.24	Novodex	51
4.25	ODE	51
B.1	UML-diagram of the manager.	73
B.2	UML-diagram of the building blocks.	74
B.3	UML-diagram of the Novodex connection.	75
B.4	UML-diagram of the ODE connection.	76
B.5	UML-diagram of the Newton connection.	77

List of Tables

4.1	Theoretical evaluation requirements.	23
4.2	Available physics engines and their websites.	24
4.3	Supported features in ODE	24
4.4	ODE grading	25
4.5	Supported features in Novodex	25
4.6	Novodex grading	25
4.7	Supported features in OpenTissue	26
4.8	OpenTissue grading	26
4.9	Supported features in Newton	27
4.10	Newton grading	27
4.11	Supported features in Tokamak	27
4.12	Tokamak grading	28
4.13	Supported features in Dynamechs	28
4.14	Dynamechs grading	28
4.15	Supported features in True Axis	29
4.16	True Axis grading	29
4.17	Supported features in Bullet	29
4.18	Bullet grading	29
4.19	Grades put together	30
4.20	Runtime grading	52

Chapter 1

Introduction

There are a number of fields that can benefit from using interactive 3d-applications, medicine, education, construction and entertainment. One example is relaxation applications, where virtual environments can be of aid for people suffering from stress.

One way to improve the realism and presence in a virtual environment is to include physics. Physics will make objects bounce, spheres roll down a hill, like objects would do in the real world. A physics engine is a part in a program that computes how physical objects should move and interact with each other. Physics engines are hard to implement since they involve many research areas and often require high level of knowledge[Ken02]. Due to this many developers license a physics engine[Mar]. In the last few years several physics engines that are open source and/or free for non-commercial use have been developed. In this thesis physics engines will be evaluated and tested.

There are also disadvantages using physics in a virtual environment. Physics simulation is computational demanding and can affect the performance of the application[Mar].

Physics engines are suitable for various purposes. The physics that is supported varies widely. Physics engines typically handle rigid bodies that can interact with each other, but there are more things that a physics engine can simulate, e.g., cloth and particles[LJ00a].

1.1 Background

This thesis project was proposed by Q-Life studio¹ which designs and creates virtual and interactive 3d-applications. Q-Life is part of the Interactive Institute², a research institute that uses art, design and technology in their work. Interactive Institute consists of different research groups, so called studios that are placed at different locations in Sweden. The different studios concentrate on different research topics[Ins]. The Q-Life studio is located in Umeå and focuses on developing applications that increase the quality of life[QL].

Q-Life develops virtual reality applications, where users can navigate through a world and perform different exercises. One example is the application *Relaxation Island: A*

¹Web site: <http://www.tii.se/qlife>

²Web site: <http://www.tii.se>

Virtual Tropical Paradise, described in[WJ04]. Relaxation Island is designed to aid people suffering from stress and enables a user to walk around on a tropical island and try different relaxing exercises.

Q-Life creates applications where it is important for users to feel present and immersed. One way to make applications more realistic is to add physics.

Q-Life uses a 3d-authoring software called Virtools (see section 2.1 for more information about Virtools) to develop applications. Q-Life wishes to add a physics module to this program. Since Virtools has an application programming interface (API) through which it is possible to implement extra functionality, it is suitable to create a physics plugin.

1.2 Thesis description

A description of the thesis is presented in the following sections.

1.2.1 Goal

The goal of this thesis is to create a functional physics plugin for the *Virtools* 3d authoring software. The following functionality should be supported by the plugin:

1. Ability to add different objects that can interact with each other. It should be possible to represent an object as a box, sphere or a mesh. A mesh is the exact object while box and spheres are approximations.
2. Possibility to create different joints.
3. Be able to set parameters in the physics engine, such as gravity, friction and elasticity. It should also be possible to interact with the objects by adding forces to the system.

To be able to create a physics module, a suitable physics engine has to be chosen. This should be done by performing an evaluation of the available engines. The evaluation consists of a theoretical analysis and runtime testing.

1.2.2 Method

The following software will be used to develop the physics module for the 3d-authoring software Virtools.

- *Microsoft Visual Studio .NET 2003*, where all the code will be written.
- *Virtools DEV 3.0*, where the testing and development of example scenes will be done.

1.3 Report outline

- *Chapter 2* Background
- *Chapter 3* Design and Implementation

- *Chapter 4* Test and evaluation of physics engines
- *Chapter 5* Results
- *Chapter 6* Conclusions
- *Chapter 7* Acknowledgments

1.4 Related work

A commercial physics module exists for Virtools. It is implemented with the physics engine Havok³, which has a license cost.

³Web site: <http://www.havok.com>

Chapter 2

Background

This chapter describes the basic theory and tools used in this project.

2.1 Virtools

Virtools is a 3d-authoring tool that has been around since 1999 and offers a way to create real-time 3d applications. Users can create advanced applications without any programming knowledge, using a simple drag and drop interface. The key is the use of behavioral objects that contain functionality that affects objects in the scene. These behavioral objects are organized to describe the program flow, also known as composition. *Virtools* is composed of modules, which makes it easy to add new functionality. The main module around which all other modules are built is called *Virtools Dev*[Vir]. Other available modules are:

- *VR Pack* - Provides functionality for connecting different peripherals to the computer, e.g. 3d trackers, stereo viewing, etc.
- *Physics Pack* - Makes it possible to integrate physics in the program, the existing physics pack is using the commercial Havok physics engine.
- *AI Pack* - Integrates artificial intelligence.
- *CAD Pack* - Tool for converting several 3d formats to a format *Virtools* can read.
- *Xbox Kit* - Makes it possible to create Xbox applications.
- *Server* - Makes it possible to add network support in *Virtools* applications.

Another feature offered by *Virtools* is that applications can be exported to a format readable by a web player. This makes it possible to view the created applications in a web browser. A limitation with *Virtools* is that it is only available for Microsoft Windows.

Virtools Dev processes behavioral objects. Behavioral objects in *Virtools* are implementations of actions that can affect objects in the compositions. The scripts created by the behavioral objects in the composition are processed once each frame creating the desired behavior of the scene.

Behavioral objects can be either building blocks, behavior graphs or scripts. These three will be described further below.

The core of Virtools is a behavioral engine which processes behavioral objects. The behavioral engine is often referred to as CK2. Several types of plugins can be added to this engine to add functionality. This is illustrated in figure 2.1.

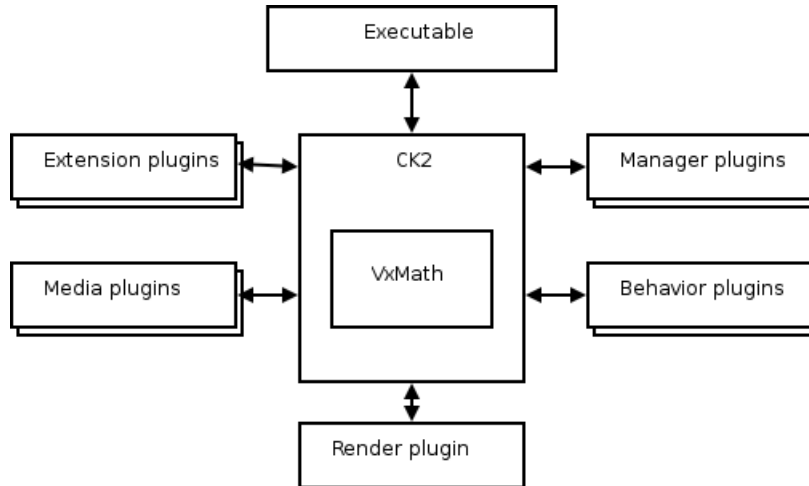


Figure 2.1: Virtools plugins structure

The differences between the plugins are described here.

- Manager plugins handle system wide algorithms. Example of such plugins can be a sound-manager that handles all sound input and output in a program.
- Behavior plugins add functionality that affect objects in a composition. It can be anything from rescaling an object to handling input from the keyboard.
- Media plugins can add functionality such as importing pictures, sounds or movies from a file format to a format that virtools can use.
- Extension plugins can be new parameter types or new parameter operations.
- Renderplugin handles the rendering of the scene.

At runtime the CK2 engine executes the behavioral objects each frame. The behaviors are executed in a sequential way following the definition of the composition.

2.1.1 Behavioral objects

The basic type of behavioral object is a building block which is a visual representation of a function that can be applied to objects in the composition. An example can be a translation which if applied to an object moves the object a distance defined in the input parameters to the building block.

A building block can have several input and output parameters graphically presented as small boxes and arrows, see image below. On the left side there is an activation-pin that activates the functionality of the building block. On the right side, the output-pin activates the next building block in the execution chain. On the top there are several input parameters which affect the functionality of the building block.

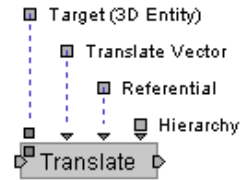


Figure 2.2: A simple building block in Virtools

For more complex behavioral objects several building blocks can be put together in a group defining a more advanced functionality. These groups are referred to as behavior graphs. Behavior graphs can have both in and out parameters just as building blocks. Graphs can also be nested.

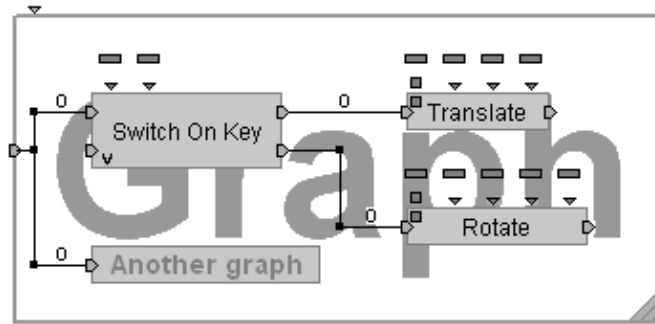


Figure 2.3: An example behavior graph

The most complex behavioral objects are called scripts. Scripts are a collection of building blocks and behavioral graphs, describing the program flow. Each script is attached to an entity in the scene, for example a camera. The following script allows the user to move around the camera in the 3d-environment using keyboard and mouse.

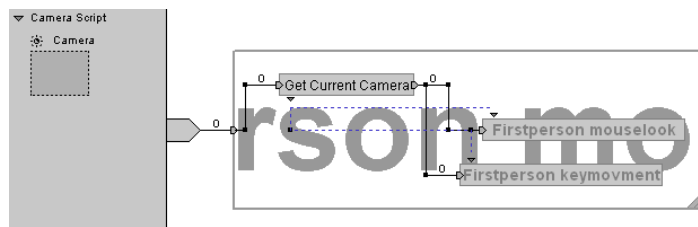


Figure 2.4: A script

2.2 Terminology

This section will describe some terminology used when discussing physics engines.

2.2.1 Physics engine

A physics engine is a part in a program that computes how objects should behave. To simulate objects, attributes such as mass, velocity and friction are used. Physics engines can simulate a variety of different physical entities like rigid bodies, cloth and liquid.

A physics simulation can be divided in two important phases, collision detection and

dynamic simulation. These two phases can then be divided further depending on different aspects, e.g., what objects that are supported, there can be different solving methods. The paper *Module Based Design for Rigid Body Simulators*[Ken02] gives a good description of how a physics engine is built.

2.2.2 Object representation

Objects in a physics engine can be represented in different ways. It is common that objects are represented by primitives such as boxes, spheres or cylinders. Objects can also be represented as a triangle mesh, which can represent any shape. It is common that only convex triangle meshes are supported. An object is convex if for any pair of points in the mesh a straight line can be created without it crossing the surface of the object.

2.2.3 Friction

Friction is a force that occurs when two objects are in contact and are moving relative to each other. The friction force counteracts their motion and works in a direction perpendicular to the normal force of the contact surfaces. The friction force causes heating and molecular deformations.

2.2.4 Elasticity

Elasticity also called restitution is a measurement on how much kinetic energy is going to be preserved in a collision.

2.2.5 Constraints

A rigid body has 6 degrees of freedom when not constrained. 3 translational and 3 rotational. Translational freedom mean that the objects can move in each of the three dimensions, rotational freedom allows the objects to change angle around three perpendicular axes. A joint constraint is a model that removes degrees of freedom. E.g. a hinge joint forces an object to rotate about one axis, so the hinge joint has only one degree of freedom. The constraints supported in this project are hinge, ball socket and slider.

2.2.6 Energy

While evaluating physics engines it is interesting to know how well the energy in the scene is preserved. It is common that physics engines damp out energy to make the simulations more stable. There are kinetic energy E_k and potential energy E_p . Kinetic energy can be divided in two parts, the rotational energy E_{kr} and the translational E_{kv} . These are calculated in the following way:

$$\mathbf{E} = \mathbf{E}_k + \mathbf{E}_p = \mathbf{E}_{kv} + \mathbf{E}_{kr} + \mathbf{E}_p = \frac{1}{2} \cdot \mathbf{m} \cdot \mathbf{v}^T \cdot \mathbf{v} + \frac{1}{2} \cdot \mathbf{m} \cdot \mathbf{w}^T \cdot \mathbf{I} \cdot \mathbf{w} + \mathbf{m} \cdot \mathbf{g}^T \cdot \mathbf{p} \quad (2.1)$$

Where m is the mass, v is the velocity, w is the angular velocity, g is the gravity and p is the position of the object.

2.2.7 Simulation methods

The heart of the physics engine is called a solver or a stepper. The solver, given a time step, takes a step in the simulation meaning that new positions, linear velocities, angular velocities, etc, are calculated. It has to take all collisions and constraints into consideration. Different approaches can be taken when solving for the new parameters, for example constraint, penalty and impulse based methods[Ken05].

Chapter 3

Design and Implementation

This chapter will describe the software plugin design and implementation. First the requirements of the plugin are described. From this the design is presented. Finally some implementation details will be discussed.

3.1 Related Work

To our knowledge there does not exist a similar solution for Virtools where a general interface for a physics engine is implemented. But there exists general physics interfaces that could have been used instead of the one we designed. The interfaces we looked at were.

- OPAL, Open Physics Abstraction Layer
http://ox.slug.louisville.edu/~o0lozi01/opali_wiki/index.php/Main_Page
- Gangsta Wrapper
<http://sourceforge.net/projects/gangsta>

The reason that they were not chosen was that they didn't include support for fetching data from the physics engine needed to do the testing.

3.2 Requirements

The requirements can be divided into functional and non-functional requirements. These requirements have been set by our commissioners. One more type of requirement has been added, called requirements for testing, which are functional requirements set by us so the tests can be performed. All these requirements will be evaluated later in the report.

3.2.1 Functional Requirements

Functional requirements are descriptions of features in the project, functions and the system behavior[Sha01].

The following functional requirements were considered during the design and implementation of the software:

- *Dynamic Objects, objects that are influenced by the physics.*
- *Static Objects, objects that can't move but is still influenced by physics.*
- *Composite objects, objects consisting of more than one geometry.*
- *Support for the following primitives: sphere, box and triangle mesh.*
- *Joints, at least the following joints should be supported:*
 - *Ball-Socket*
 - *Hinge*
 - *Slider*
- *Be able to set different physical aspects on objects such as friction and elasticity*
- *Deactivate/active objects.*
- *Be able to apply impulses and torque on objects.*
- *Set the position of an object, without changing the physical properties.*

3.2.2 Non-Functional Requirements

Non-functional requirements are restrictions (restrictions of speed, memory) of the attributes in the functional requirements [Sha01].

The non-functional requirements used in this project are:

- *The plugins should work well with Virtools.*
- *The software layer should not affect the overall performance of the applications.*
- *The software should use minimum amount of memory and computational power.*

3.2.3 Requirements for testing

The following list states additional requirements on the software that is needed during the testing:

- *Be able to fetch the energy from objects, both potential and kinetic.*
- *Be able to set the linear and angular velocity.*
- *Fetch other parameters, e.g. time-step, distance between positions.*
- *Print data to files.*
- *As far as it is possible, features are going to be implemented as building blocks.*
- *Be able to easily replace the physics engine.*

3.3 Design

The design is based on the previously stated requirements and discussions with the commissioners and the supervisor.

3.3.1 Overview

To be able to easily replace a physics engine, a modular design approach was considered. One main module will communicate with Virtools. Each physics engine used in the project must then have a wrapper which the main module can communicate with. Figure 3.1 shows an overview of the modules in the plug-in software.

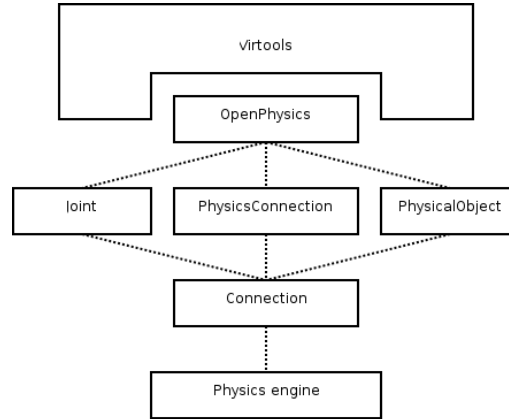


Figure 3.1: Overview of the plug-in software

The main module is called `OpenPhysics`¹, see 3.3.2 for a design description of this module.

The module where each physics engine is wrapped is called a `Connection`. The `Connection` is described further in section 3.3.3.

As figure 3.1 shows, the layer between `OpenPhysics` and the `Connection` consists of `PhysicsConnection`, `PhysicalObject` and `Joint`. This layer is abstract and each connection must implement this.

Each physical object is shared between the main module(`OpenPhysics`) and the `Connection`. The objects are created from `OpenPhysics` but initialized in the connection. This approach makes it easy for the main module to, for example update the position of the objects since the reference to the object is known. It is also a good idea to separate primitive objects and joints, since these have different characteristics.

3.3.2 OpenPhysics

`OpenPhysics` is the main module of the plug-in software, which is designed as a manager in Virtools. Through this module all communication with Virtools is done. Communication with the physics engine is done through the connection which also is illustrated in figure 3.1.

¹This has nothing to do with OPAL, which was described earlier.

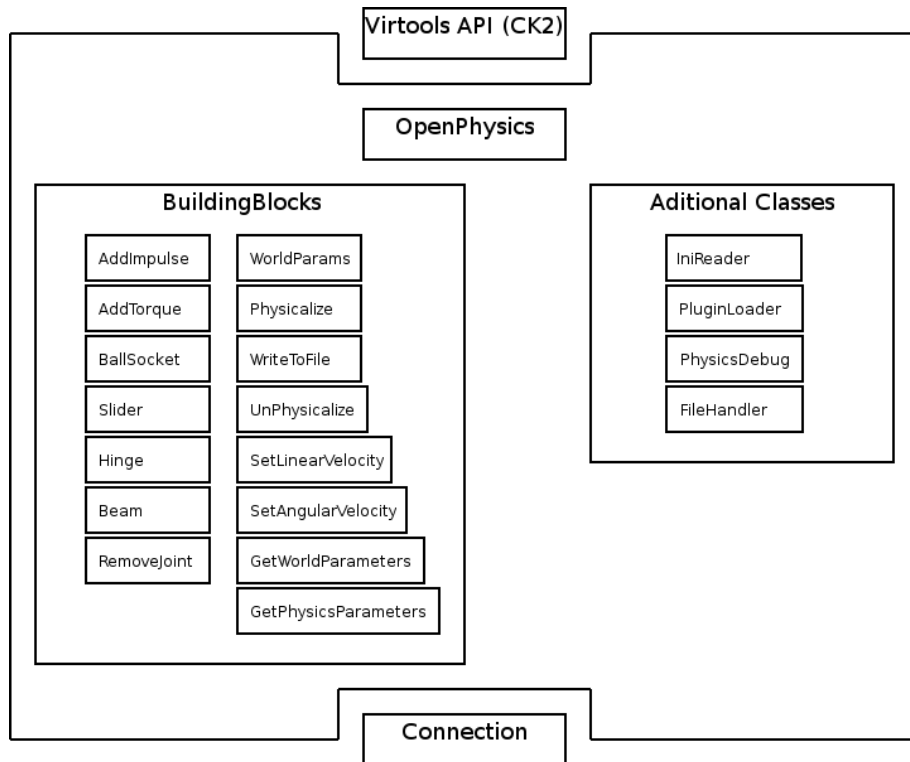


Figure 3.2: Overview of OpenPhysics.

Figure 3.2 shows an overview of the main module, OpenPhysics. OpenPhysics has some help classes to read initialization files and load a Connection module dynamically. OpenPhysics communicates with Virtools through the API CK2.

The different building blocks are defined in OpenPhysics. The following building blocks are implemented. For a full overview of the building blocks, see the user-guide in appendix A:

- **WorldParams**, *Set parameters in the physics engine.*
- **Physicalize**, *Make an object participate in the physics simulation. Parameters specific for the object can be set here.*
- **HingeJoint**, *Create a hinge joint between two objects.*
- **BallSocketJoint**, *Create a ball-socket joint between two objects.*
- **SliderJoint**, *Create a slider joint between two objects.*
- **GetWorldParameters**, *Fetches global attributes from the scene.*
- **GetPhysicsParameters**, *Fetches physical attributes from a physical object in the scene.*

- **Beam**, *Move an object.*
- **AddImpulse**, *Add an impulse to an object.*
- **AddTorque**, *Add torque to an object.*
- **UnPhysicalize**, *Remove an object from the physics simulation.*
- **RemoveJoint**, *Remove a joint.*
- **SetAngularVelocity**, *Set the angular velocity of an object.*
- **SetLinearVelocity**, *Set the linear velocity of an object.*
- **WriteToFile**, *Write data to a specified file.*

Virtools API

Virtools has an extensive API called CK2, to which plugins can be implemented. OpenPhysics is going to be run as a manager in Virtools. A manager is a plugin for Virtools and has methods which are executed on certain occasions during each frame.

This will be integrated into Virtools through a dynamically linked library (DLL), containing the physics manager and the related building blocks. Figure 3.3 illustrates the design.

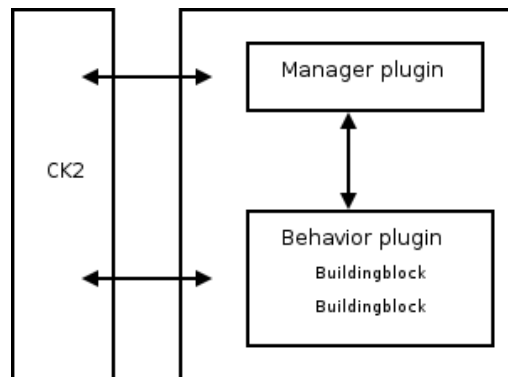


Figure 3.3: DLL design

OpenPhysics is unaware of the underlying physics engine. OpenPhysics reads from a initialization file which contains information about the physics engine to load. The file must be called OpenPhysics.ini and contain the following lines:

```
[Settings]
PhysicsConnection=NameConnection.dll
```

See the users guide in Appendix A for a more thorough explanation.

3.3.3 Connection

Each implemented connection must inherit the abstract classes: `PhysicsConnection`, `PhysicalObject` and `Joint` as shown in figure 3.1. Apart from these functions, other functionality specific for each physics engine has to be implemented. Figure 3.4 shows a sample of one Connection, in this case the physics engine Newton Game Dynamics. The complete UML-diagram can be viewed in Appedix B.

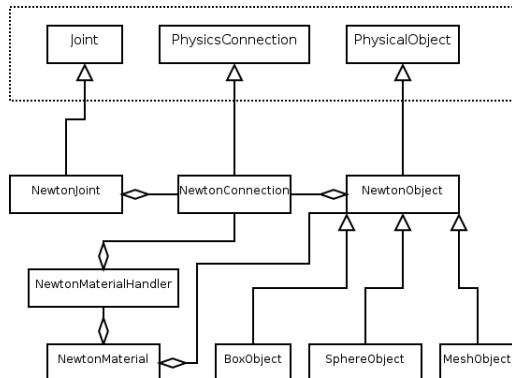


Figure 3.4: UML over the NewtonConnection

3.3.4 Debug

Since all external plugins to Virtools are implemented as DLL's² it is very hard to debug the plugin libraries. One way to print debug text in Virtools is to write to the built-in console. Another way to display debug information is to draw objects in the scene through the CK2 API.

To be able to test and debug the plugin, a *PhysicsDebugLib* was implemented. This library handles different kinds of output functions, such as printing text, numbers, vectors, drawing spheres and drawing boxes.

To perform the tests all the output had to be written to text files. To accomplish this a file-handler was created to handle output to several files simultaneously. The debug-library is used in the OpenPhysics manager as well as in the connections. When building the final version of the plugin, this library may be removed.

3.4 Implementation

The implementation follows the design described in the previous section. This section will have the same structure as the design section.

²Dynamic-link library

3.4.1 Overview

All the implementation have been done in C++, all functionality have been built in an object oriented way as far as possible. The project produces one DLL³-file for the OpenPhysics module and one DLL-file for each Connection that has been implemented.

3.4.2 OpenPhysics

OpenPhysics consists of different classes that handle specific things. The main class is the OpenPhysicsManager which contains the main simulation loop. This loop basically have four different phases, update time, update physics, update objects and optional time for testing. The main loop is illustrated in figure 3.5.

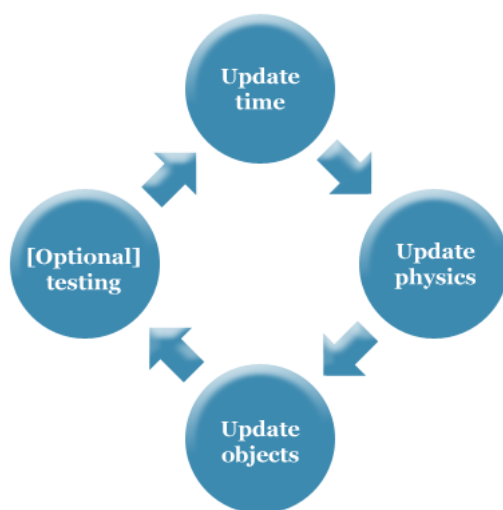


Figure 3.5: Simulation loop

The functionality for loading the Connections is implemented in a class that uses templates⁴ to make the connection as loose as possible.

The implemented building blocks are defined in a large structure which is registered in Virtools. The structure is defined as a *CKPluginInfo* where each instance contain information about the building block. This information defines which functions to call when the building block is initiated/exited, information about where in the building block hierarchy to place it and other information like version, author and a summary.

³Dynamic-link library

⁴Term in Generic programming, which means that a value can have different data types.

3.4.3 Connections

The different Connections must be treated and implemented in different ways. In the following subsections some of these differences for each physics engine will be described. The implemented physics engines are Newton Game Dynamics, ODE⁵ and Novodex⁶. These will be described more in chapter 4.

Newton Game Dynamics

One thing that distinguishes Newton from the other physics engines, is that for each new object with a new material the behavior between these objects have to be specified. The following functions were used to set the behavior between the objects:

```
NewtonMaterialSetDefaultCollidable
NewtonMaterialSetDefaultFriction
NewtonMaterialSetDefaultElasticity
```

This means that if there exists x different materials and these should all be able to interact with each other, then $x \cdot (x - 1)$ behaviors have to be specified. This was implemented with a Material-class where the behavior between different materials is set as soon as a new material is created. Both the friction and the elasticity/restitution was set to the mean value between the objects.

Newton has one big matrix for all positioning of an object (Transformation matrix). This means that if you want to extract more than the Euler angles you have to implement this.

Setting forces and torques has to be done in a callback-function called:

```
NewtonBodySetForceAndTorqueCallback
```

Since the callback-function isn't a class member of NewtonObject, the class attributes cannot be used in the callback-function. This was solved by setting the current NewtonObject as user data to the object. When the callback-function is invoked the user data was used to call a class-function:

```
static_cast<NewtonObject*>(NewtonBodyGetUserData(body))->applyForceAndTorque();
```

In Newton there is no way of setting another inertia than the diagonal inertia. This is no problem in this project but might be in the future if unsymmetrical objects are used.

ODE

ODE has a clean C interface and it is relatively simple to use. The collisions in ODE are handled with a callback-function. This is done in much the same way as with forces and torques in Newton. The callback-function is registered by calling the following function:

```
dSpaceCollide(_space, ((void*)this), &nearCallback);
```

User data is set to each object and when the callback-function gets called the user data is used to call a class function that handles the collision. In ODE this looks like:

⁵Open Dynamics Engine

⁶AGEIA NovodeX

```
static void nearCallback (void *data, dGeomID o1, dGeomID o2)
{
((ODEConnection*)data)->nearCallBack(o1, o2);
}
```

Novodex

Objects in Novodex are called NxActors. All objects/actors which are in the simulation are added to a NxScene.

In Novodex it is not necessary to define a callback-function for collisions. Friction and elasticity parameters in the contacts are calculated automatically.

3.4.4 Debug

The Debug library was built as a static library and can be removed before making a final version. This library contains functions for printing and drawing debug information. Drawing in Virtools must be done when Virtools enters the render loop. This loop is accessed by registering a callback-function through the render manager.

```
CKRenderManager *m = _myContext->GetRenderManager();
```

This gives a reference to the module which has a list of “render contexts” which is where the actual drawing occurs.

```
CKRenderContext *rc = m->GetRenderContext(0);
```

This call fetches the first render context. The callback-function can be registered to this context.

```
rc->AddPostRenderCallBack(renderCallback, ((void*)this) ,false, false);
```

Now the function renderCallBack will be called when Virtools enters the render loop.

Another frequently used approach to debug was to write data to a text file. For this purpose a file-handler class was developed, where it was possible to create and write to any number of files.

Chapter 4

Test and evaluation of physics engines

This chapter will evaluate and test physics engines. The results will be used to decide which physics engine that is going to be used in the software implementation.

4.1 Introduction

Physics engines support different features and are used in different ways. When running the same scene with different physics engines the result will vary. This is why it is interesting to evaluate physics engines to see how they behave in different situations.

4.1.1 Description

The purpose of this in-depth study is to evaluate and compare open source or free physics engines. The engines will be compared both by looking at their documentation and their performance to find their pros and cons. They will also be tested in several run-time tests.

It is important to consider more than rigid body physics which is the first thing that one often associates to the word physics engine. Modern physics engines can often handle much more, e.g., cloth, fluid and deformable objects. These features will not be tested in the runtime tests but will be considered during the more theoretical comparison.

The results of this test will form a base for deciding which physics engine is the most suitable to use in the physics module.

4.1.2 Related Work

There are not much work done testing different physics engines. There are two articles written 2000 by Jeff Lander and Chris Hecker called *Product Review of Physics Engines, Part One: The Stress Tests*[LJ00a] and *Product Review of Physics Engines, Part Two: The Rest of the Story*[LJ00b]. These articles present three different physics engines, MathEngine, Iqon and Havok. With these engines twelve different tests were done, which tested different aspects concerning collision detection, some constraints and

a hinge joint. These tests are presented in the first article where the different engines get grades on how well they perform. The second article is an evaluation of the different engines which considers: *Feature set*, *Documentation*, *Ease of use*, *Production*, *Integration*, *Input and feedback*, *Cost* and *Technical support*.

Some of the tests in the articles are interesting but there are many important things that were not tested e.g. friction and energy preservation. Another thing that was missing from the test was some data from the results. The results was based on observations by the authors.

4.2 Product Research

Choosing a physics engine is hard since there are many engines out there with a lot of different features and limitations. Requirements have been established to use as a base in the evaluation of physics engines. These requirements are presented in the following subsection. As a starting point, a number of physics engines will be evaluated to select three engines that will be tested further in nine runtime tests.

4.2.1 Requirements

Defining requirements for a test like this is difficult because it is hard to grade the different requirements. The grading will be based on our judgment. Each engine will be given three grades, one grade for each part in table 4.1. The grade will be a number between 1 and 5 where 5 is the best grade. The requirements are described in table 4.1.

<i>Features</i>	The features that the physics engine supports will be presented here. What kind of collision primitives it supports (box, sphere, convex mesh, general trimesh). Does the engine support particle systems? Does it have deformable objects? Does the engine have rag doll support or any other special functionality.
<i>Documentation</i>	The documentation will be graded on how good it is, how well documented the functions in the SDK are. Can example projects be found that show sample implementations of certain features? Also how good community support such as mailing lists or forums that exists.
<i>Usability</i>	This has to do with how easy it is to integrate the engine into another program. Is the engine written in an object-oriented manner? Also factors such as if the engine supports multiple platforms or if it is bound to a certain platform. Since we can't test implement all engines this requirement will be affected on how good the documentation is.

Table 4.1: Theoretical evaluation requirements.

4.2.2 Available Physics engines

The following physics engines have been found, table 4.2. These engines will be further analyzed using the requirements presented in the previous section with the purpose of narrowing the list to the most interesting.

<i>Open Dynamics Engine</i>	http://www.ode.org/
<i>AGEIA NovodeX</i>	http://www.ageia.com/
<i>OpenTissue</i>	http://www.opentissue.org/
<i>Newton Game Dynamics</i>	http://newtondynamics.com/
<i>Tokamak</i>	http://www.tokamakphysics.com/
<i>Dynamechs</i>	http://dynamechs.sourceforge.net/
<i>True Axis</i>	http://www.trueaxis.com/
<i>Bullet</i>	http://www.continuousphysics.com/Bullet/
<i>CM-labs Vortex</i>	http://www.cm-labs.com/
<i>Havok</i>	http://www.havok.com/

Table 4.2: Available physics engines and their websites.

Since one of the original requirements was that the engine should be free for non commercial use Havok and CM-labs Vortex will not be evaluated.

4.2.3 Evaluation of physics engines

In this section the remaining physics engines will be evaluated using the requirements previously specified. The purpose is to select three engines that will be further tested in the runtime tests.

The presented information below is collected from each physics engines web site.

Open Dynamics Engine

Open Dynamics Engine or ODE is a popular open source physics engine that has been around since 2001 and is used in many projects. ODE has a *GNU Lesser General Public License*¹. The main author of ODE is *Russel Smith* and the current version of ODE is 0.5.

Types	Support
Collision Primitives	Box, Sphere, Cylinder, Plane, Ray and Trimesh
Joint Types	Ball-Socket, Hinge, Slider (prismatic), hinge-2, Fixed, Angular motor, Universal
Other	Offers collision spaces where the collision handling can be organized in hierarchies. Composite objects, more complex objects can be created and be treated as one body.
Platforms	Windows, Linux and MacOS.

Table 4.3: Supported features in ODE

The online documentation is good with many describing sections. There is also a mailing list where one can get answers on different questions about ODE.

¹License: <http://ode.org/ode-license.html>

Together with ODE follows some demos that are a good starting point when trying to use the engine. ODE is written in C but there exists a C++ interface[ODE]. ODE works on Windows, Linux and MacOS.

Features	Documentation	Usability	Average
3	4	4	3.6

Table 4.4: ODE grading

AGEIA Novodex

AGEIA was founded in 2002 and their physics toolkit is called Novodex. The latest version of Novodex is 2.2. The Novodex physics SDK is a famous physics library which is used by a number of prominent companies. Novodex is free for non-commercial use, which makes it possible to use in these tests.

Types	Support
Collision Primitives	Box, Sphere, Capsule, Plane, Convex and non-convex mesh
Joint Types	Ball-Socket(Spherical), Hinge(Revolute), Slider (prismatic), Fixed, 6 degree of freedom joints with motors, Springs, Projection and joint with limits. Joints can also be breakable
Other	Support for rag doll, vehicle, character controllers Collision groups Particle system
Platforms	Windows and MacOS.

Table 4.5: Supported features in Novodex

As seen in table 4.5 Novodex has many features. The most interesting are presented in the table.

The documentation is extensive, with documents, tutorials and sample implementation. There is also a forum on AGEIA's webpage where questions can be asked.

The SDK also includes many help functions used for math calculations, saving data etc. This collection of help functions increases the usability and results in a more structured program.

Another interesting thing about AGEIA is the PhysX chip they have developed. This chip has a PPU (Physics Processor Unit) where physics can be simulated on the hardware. This is the first physics processor in the world. This new processor will make physics calculations faster[AGI].

Features	Documentation	Usability	Average
4	4	5	4.6

Table 4.6: Novodex grading

OpenTissue

OpenTissue is a physics library launched in 2001 by teachers at *Department of Computer Science, University of Copenhagen*. OpenTissue is a collection of code written by these people to facilitate their teaching. 2003 OpenTissue was released under the *GNU Lesser General Public License*².

Types	Support
Collision Primitives	Box, Sphere, Cylinder, Plane, Ray, Tetrahedron, Meshes
Joint Types	Ball-Socket, Hinge, Slider (prismatic), Universal, Wheel, Linear limits and angular limits. Joints can also be motorized.
Other	Splines, Particle systems (cloth), SPH, Deformable objects
Platforms	Windows, Linux and MacOS.

Table 4.7: Supported features in OpenTissue

As stated above, OpenTissue is a collection of several peoples work, assembled into a physics toolkit. As seen in table 4.7 OpenTissue offers a lot of features, even more features exists. The big difference from other physics engines is that the toolkit has deformable objects.

It looks like OpenTissue doesn't have any documentation except the one that is written in the code. This makes it hard to use.

OpenTissue is written in an object oriented style with modular elements and templates, this makes the implementation hard in the beginning. But as soon as these structures are initialized the usability/readability of the engine is increased.

OpenTissue differs from other physics libraries in the way a scene is constructed. Usually there is a world to which you can add objects(box, sphere, mesh). In OpenTissue you initialize a "configuration" to which you add objects. The "world" in OpenTissue is called a simulator, to this simulator you can add a configuration, collision detection and a stepper. The library hierarchy makes it very modular, where it is easy to change one part. This modular design introduces some extra work. In other toolkits the gravity is set to the world and all objects in this world have the same gravity. In OpenTissue gravity is handled like any other force and has to be explicitly set (attached) to the objects. OpenTissue can be compiled on both Windows and Linux[Ope].

Features	Documentation	Usability	Average
5	1	3	3.0

Table 4.8: OpenTissue grading

²Terms and conditions: <http://www.gnu.org/licenses/lgpl.html>

Newton Game Dynamics

Newton or Newton Game Dynamics is a free physics engine that is made for several platforms, Windows, Mac and Linux. It is not an open source project but has a free and open interface (SDK). The latest version is Newton SDK 1.32. Newton has been around since 2003.

Types	Support
Collision Primitives	Box, Sphere, Cylinder, Cone, Capsule, Convex mesh, Heightmap
Joint Types	Ball-Socket, Hinge, Slider (prismatic), Corkscrew, Universal, Up-vector Cone, Linear and angular limits
Other	Containers for rag doll and vehicle
Platforms	Windows, Linux and MacOS.

Table 4.9: Supported features in Newton

Newton also have functionality for raycasting which works with all collision primitives.

The documentation is good with relevant texts about the functions that the SDK offers. There are also several tutorials and sample implementations that are good to learn from. An active forum exists. [Dynb]

Features	Documentation	Usability	Average
3	4	3	3.3

Table 4.10: Newton grading

Tokamak

This engine is free for both non-commercial and commercial use. The main author is David Lam and the engine has been available since 2003. The engine is created as a game engine with focus on simple and fast computations. Current version is 1.25.

Types	Support
Collision Primitives	Box, Sphere, Capsule, Convex mesh, Static triangular mesh.
Joint Types	Ball-Socket, Hinge joint limits
Other	Rigid Particle, a light weight rigid body Breakable Objects.
Platforms	Windows.

Table 4.11: Supported features in Tokamak

Tokamak offers a SDK documentation on the web which is not that detailed. Their webpage also has a forum page where questions can be asked.

The engine has an object oriented interface written in C++. The SDK is only available for Windows[gP].

Features	Documentation	Usability	Average
2(3)	2	2	2.0

Table 4.12: Tokamak grading

Dynamechs

Dynamechs or Dynamics of Mechanisms is a simulation software that has been under development since 1991. It is developed mainly at the Ohio State University, where it was started by Scott McMillan. There has not been any recent updates on the engine. The newest version is from 2001. Dynamechs was built to aid two graduate research projects. The toolkit is mainly aimed towards robot simulations.

Types	Support
Collision Primitives	Box, Sphere, Capsule.
Joint Types	Ball-socket, Hinge, Slider, Universal
Other	
Platforms	Windows, Linux, Solaris and IRIX-MIPSPPro.

Table 4.13: Supported features in Dynamechs

Dynamechs comes with documentation that describes the available functions/classes. There are a couple of small example programs.

The engine is written for several platform including windows and linux. The SDK interface is written in an object oriented manner[Dyna].

Features	Documentation	Usability	Average
3	2	2	2.3

Table 4.14: Dynamechs grading

True Axis

True Axis SDK is a physics engine developed in Australia. True Axis is a engine that focuses on speed and sacrifices realism. One special feature is the *Swept collision testing* which is a functionality that handles fast moving objects and prevents them from passing objects in their way. True Axis is free for non-commercial use.

The SDK is written in C++ and works on both Windows and Linux. The documentation that follows True Axis has some example source code to demonstrate how to use the physics engine[Axi]. There also exists demo projects. This is the best starting point when learning the toolkit.

Types	Support
Collision Primitives	Box, Sphere, Capped Cylinder, Ray, Convex
Joint Types	Ball-socket, Hinge, Slider , Fixed Different Limits.
Other	Swept collision detection Functions for creating vehicle
Platforms	Windows.

Table 4.15: Supported features in True Axis

Features	Documentation	Usability	Average
3	3	3	3.0

Table 4.16: True Axis grading

Bullet

Bullet or Bullet Collision Detection and Rigid Body Dynamics Library is a free and open source physics toolkit. Bullet was developed to share knowledge between physics developers. Bullet uses parts from the ODE library, e.g., the lcp solver[Bul].

Types	Support
Collision Primitives	Box, Sphere, Cylinder, Cone, Convex, Triangle Mesh
Joint Types	User-defined joints
Other	Discrete and Continuous Collision Detection.
Platforms	Windows.

Table 4.17: Supported features in Bullet

The only documentation that is available is auto generated from the source code.

The design of the physic engine is modular and it is easy to change a part of the physics engine, e.g., the solver.

Features	Documentation	Usability	Average
3	2	3	2.6

Table 4.18: Bullet grading

4.2.4 Result

Now it is time to summarize the results and present the engines that were chosen for the run-time tests. All engines support the standard collision primitives and the standard joints. Therefore the requirements that separate the engines from each other are how well they support mesh-collision and how well documented the API is. Another important factor for the commissioner is the fact that the engine is free and preferably not only free for non-commercial use. Table 4.19 summarizes the grades from the evaluation for each engine. The maximum grade is 5.

Physics Engine	Average grade
Novodex	4.6
ODE	3.6
Newton Game Dynamics	3.3
OpenTissue	3.0
True Axis	3.0
Bullet	2.6
Dynamechs	2.3
Tokamak	2.0

Table 4.19: Showing the grading of all evaluated engines sorted by their grade.

Since Dynamechs doesn't support mesh-collision at all, it has a much smaller chance of being chosen than the others. Tokamak, True Axis and Newton all support convex mesh collision and static general triangle meshes. ODE, Novodex and OpenTissue all support general trimesh collision even though Novodex and OpenTissue are more stable than ODE. This is because they use a pmap structure, which is a data structure that makes collisions between mesh objects more stable at the cost of higher memory utilization.

Some of the other features that are worth mentioning are that Tokamak has support for breakable objects. Novodex and OpenTissue have support for particle systems, OpenTissue also has deformable objects. Newton Game Dynamics has containers for both vehicles and ragdolls. True Axis also has support for vehicles in a library available from their webpage.

When it comes to documentation, OpenTissue and Dynamechs have very little information available on their webpage. ODE and Novodex have really good documentation with many example projects where users can find examples of the functionality in use. Newton, Tokamak, Bullet and True Axis also have good documentation but not as extensive as ODE and Novodex.

The engines that are free only for non-commercial use are Novodex and True Axis. The other engines are free to use in any project. Another factor to mention is that ODE, Bullet and OpenTissue are open source engines.

All the above information was used to select the engines for further testing. The result is based on how well the physics engines fulfill the requirements.

Novodex has the highest grade and is the first choice for further tests. One of the few negative things about this engine is that it is only free for non-commercial use.

ODE is also a very capable engine and completely free since it is open-source. Therefore ODE was also chosen for the runtime tests. The third engine is Newton Game Dynamics. The reason that it was chosen is mainly that it is free for commercial use and it also got a higher grade in the evaluation than True Axis.

The three engines chosen for further testing are:

1. Novodex
2. ODE
3. Newton Game Dynamics

4.3 Runtime Tests

The three engines chosen for further testing will undergo several performance tests in critical situations. These have been chosen to evaluate as many different aspects as possible, including friction models, scalability, and energy preservation. The following sections will explain the results and how they were planned to be executed.

All tests simulating used 0.01s as the time step.

4.3.1 Testing dry friction

Two friction tests will be performed. The first one is to see how the friction forces corresponds to the Coulomb friction law. This model distinguishes between the case when two objects are sliding on each other, dynamic friction, and when the objects don't move, static friction, [MB95, SJ98]. In the case of static friction the active friction force is only limited by the normal force of the contact:

$$|\mathbf{F}_t| \leq \mu_s |\mathbf{F}_n|. \quad (4.1)$$

In the other case with dynamic friction, when the objects have a relative velocity, the friction force is proportional to the normal force and pointing in the opposite direction of the velocity in the contact plane:

$$\mathbf{F}_t = -\mu_k |\mathbf{F}_n| \hat{\mathbf{v}}_t. \quad (4.2)$$

The engines will also be tested for anisotropy. Anisotropy means that the active friction coefficient can vary when the gravity is varied in 360 degrees around a inclined plane. This results in a friction pyramid instead of a friction cone. The test can be compared to a plane fixed with a ball and socket joint in the middle and then the plane is tilted in all possible angles.

Description

Both test cases will be performed using a box sliding on a inclined plane. In the first test where the coulomb friction model is tested, the scene will simulate an inclined plane by setting the gravity according to the following formula, where θ is the angle of inclination:

$$\mathbf{G} = -\mathbf{g} \begin{bmatrix} \sin(\theta) \\ \cos(\theta) \\ 0 \end{bmatrix}. \quad (4.3)$$

The angle will then be changed from 0 to $\frac{\pi}{2}$ with a constant value on the gravity, $\mathbf{g} = -10N$. The size of the box is set to [6, 1, 6] m. The friction coefficient is set to 0.5 for both the box and the plane. During the test run, the average ratio of tangential to normal force will be computed:

$$\frac{|\mathbf{F}_t|}{|\mathbf{F}_n|}. \quad (4.4)$$

To calculate this ratio the following assumption must hold, that the objects always are in contact with each other. That means that the following equation holds if the assumption is true:

$$\mathbf{F}_{\mathbf{gN}} + \mathbf{F}_N = \mathbf{0}. \quad (4.5)$$

F_{g_N} is then calculated using the following formula where m is the mass of the box:

$$\mathbf{F}_{g_N} = m \cdot \mathbf{G} \cdot \hat{\mathbf{y}}. \quad (4.6)$$

And F_{g_T} is calculated like this:

$$\mathbf{F}_{g_T} = \mathbf{F}_g - \mathbf{F}_{g_N}. \quad (4.7)$$

When this is known the friction force can be calculated from this formula:

$$\mathbf{F}_{\text{friction}} = m \cdot \mathbf{a} - \mathbf{F}_{g_T}. \quad (4.8)$$

Now it is possible to calculate the ratio between tangential and normal force.

Picture 4.1 shows the setup of the scene.

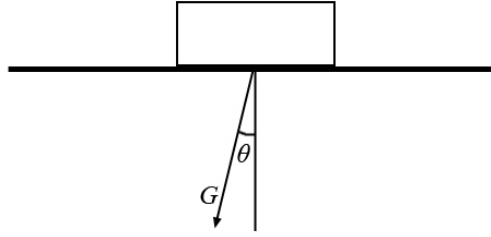


Figure 4.1: First friction test

The second test will find the minimum θ for which the box slides on the plane. It will also vary α from 0 to 2π . The test is designed to find if the friction approximations in the physics engines uses a isotropic or anisotropic friction model. The gravity parameter g was set to -10N and the active gravity (G) will be calculated by the following formula to get a gravity that simulates the inclination of the plane:

$$\mathbf{G} = -g \begin{bmatrix} \cos(\alpha) \sin(\theta) \\ \cos(\theta) \\ \sin(\alpha) \sin(\theta) \end{bmatrix}. \quad (4.9)$$

The same value, the average ratio of tangential to normal force will be calculated:

$$\frac{|\mathbf{F}_t|}{|\mathbf{F}_n|}. \quad (4.10)$$

The values will then be plotted with a polar curve.

Result

All three engines gave good results in the first test. The active friction coefficient increased linearly until $\tan(\theta) = 0.5$. At that point the block starts to slide along the plane. One difference here is that Novodex has a higher threshold and starts to slide first at $\tan(\theta) = 0.7$. When θ is increased further the block always slides and the active friction coefficient approaches 0.5. To conclude, all engines present nice Coulomb curves, which can be found in figure 4.2.

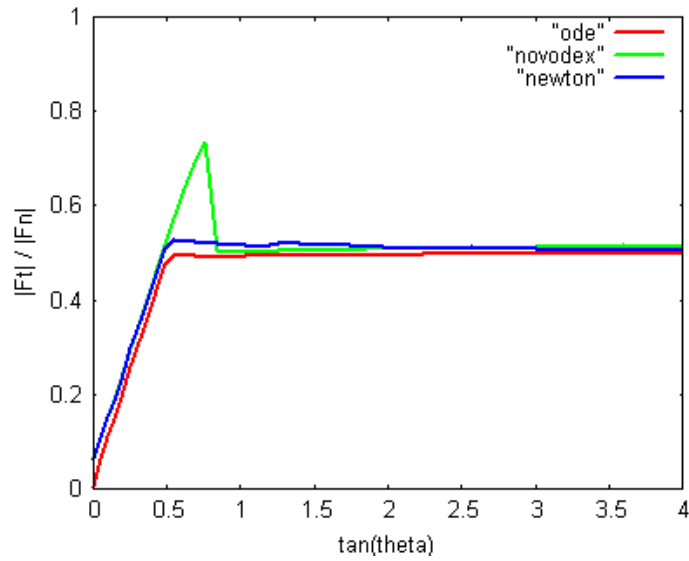


Figure 4.2: Coulomb friction curves

When it comes to the second test all engines produce friction pyramids instead of the friction cone that Coulomb friction model should generate. A friction pyramid is an approximation of the friction cone. A friction pyramid is anisotropic meaning that the active friction coefficient is larger when the box slides diagonally on the plane.

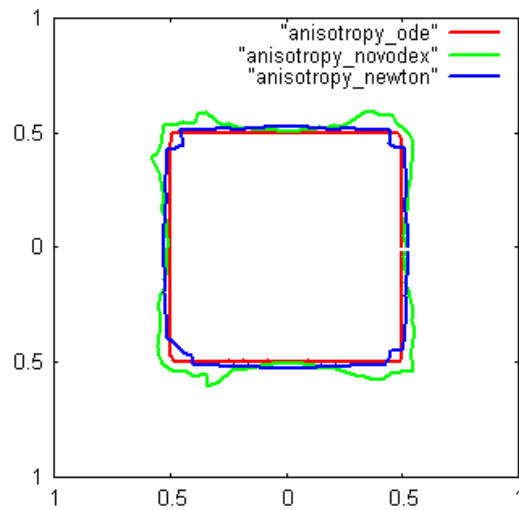


Figure 4.3: Result from the anisotropy test

4.3.2 Gyroscopic Forces

When objects are rotating around their own central point of mass, they are subject to a gyroscopic force which is hard to handle correctly.

These forces, if not handled correctly, will make the object rotate unnaturally or even explode. A common way of handling these forces is by damping the system. This method can also lead to different anomalies, e.g., a rotating body without any interference forces might stabilize and stop moving which is not the correct behavior. Handling the gyroscopic forces can be hard. In the paper “Stabilizing gyroscopic forces in rigid multi body simulations” [Cla19] different methods are evaluated and analyzed.

Test description

To test whether the engines can handle this instability or not, a box with the sides [10,4,2] m and mass $m = 100\text{kg}$ will be placed in a world with no gravity and a starting angular velocity of $\omega = [2, 3, 4]^T \text{m/s}$. The inertia tensor can then be computed to yield:

$$\mathbf{I}_0 = \frac{\mathbf{m}}{12} \begin{bmatrix} (y^2 + z^2) & 0 & 0 \\ 0 & (x^2 + z^2) & 0 \\ 0 & 0 & (x^2 + y^2) \end{bmatrix}. \quad (4.11)$$

Initial values of angular momentum will be calculated as follows:

$$\mathbf{L}_0 = \mathbf{I}_0 \omega, \quad (4.12)$$

$$\mathbf{E}_0 = \frac{\omega^T \mathbf{I}_0 \omega}{2}. \quad (4.13)$$

The simulation will run in a couple of seconds and after that the difference in angular momentum and energy will be compared with values calculated on the initial conditions to find the normalized error:

$$\Delta \mathbf{l}(\mathbf{t}) = \frac{\|\mathbf{L}(\mathbf{t}) - \mathbf{L}_0\|}{\|\mathbf{L}_0\|}, \quad (4.14)$$

$$\Delta \mathbf{e}(\mathbf{t}) = \frac{(\mathbf{E}(\mathbf{t}) - \mathbf{E}_0)}{\mathbf{E}_0}. \quad (4.15)$$

Result

Calculating the angular momentum and the energy was easy in Novodex. In this toolkit the global inertia can be fetched by a function call. In ODE the global inertia has to be calculated explicitly using the rotation matrix. Extracting inertia and the rotation matrix from Newton is not as easy. Newton has one matrix for all transformations. So extracting the rotation matrix has to be done from this “global matrix”.

The results of the test show that ODE doesn’t preserve the angular momentum. It increases exponentially and after 30 seconds the error in angular momentum is almost 700 percent of the initial value. Both Novodex and Newton have big oscillations, up to 100 percent of the initial value. One big difference between Newton and Novodex is that Newton eventually will oscillate with small intervals near 1 which means that either all angular momentum is lost or it has gained 100% in angular momentum. Figure 4.5

reveals that Newton loses energy and therefore the angular momentum is also lost. This is also visible in the simulation, the block will eventually stop moving.

Figure 4.5 shows that with ODE, the rotational energy increases in the same way as the angular momentum. With Newton the rotational energy oscillates and slowly decreases. Novodex keeps the rotational energy at the initial level. When observing that and the fact that the error in angular momentum is oscillating it is likely that Novodex only preserves the value of the angular momentum vector and not the correct values in the vector. This will ignore the gyroscopic effect.

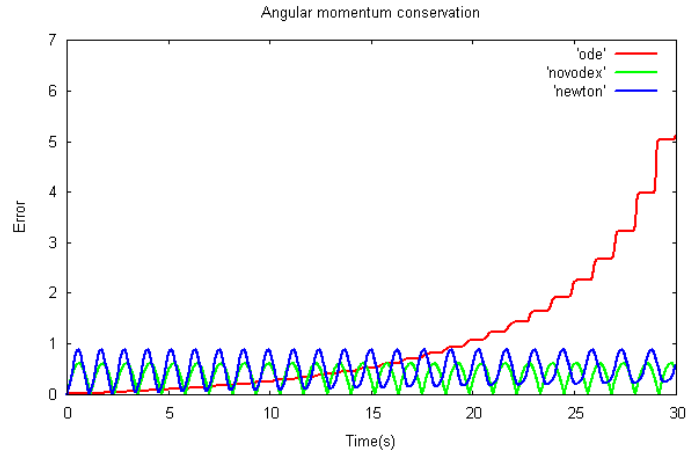


Figure 4.4: The normalized error in angular momentum.

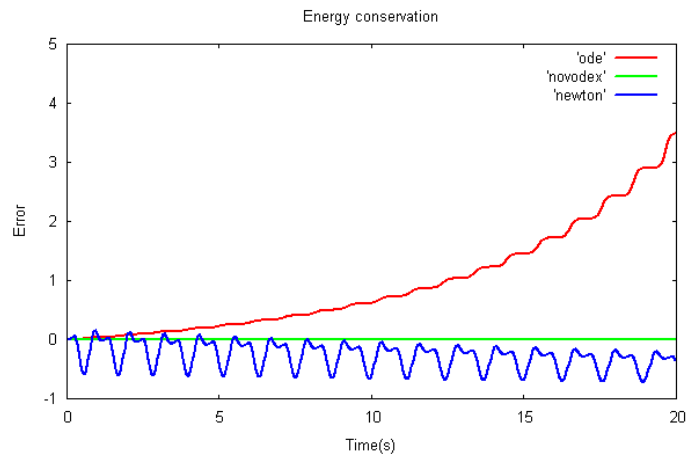


Figure 4.5: The normalized error in rotational energy.

As shown by the results this test is hard for ODE and Newton. When running the simulation a little longer, the Newton toolkit will make the object stop moving. The simulation with ODE will explode. The documentation for ODE(section 12.12) explains this problem and gives some suggestions on how to make a simulation more stable. The suggestions are not used in this test since then the physical aspects wont be fair to the other engines. Novodex handles this test best, but still does not conserve angular momentum.

4.3.3 Bounce

Objects that collide and bounce of each other is a common scenario. This test will look at how the different physic engines handle this.

Description

The purpose of this test is to find whether objects gain, lose or preserve the right amount of energy on collisions. The scene used in this test is very simple. A sphere will be placed above a plane in a world with no gravity. The kinetic energy will be calculated before and after the collision and these values will be compared.

This test have been divided into two parts.

- The first part will see what happens when two object start in collision where one object is fixed. The free moving object is a sphere and is placed with a penetration depth of 0.3 m into the plane. The sphere has a radius of 0.5 m and the plane is a box which is fixed with the following size [20, 1, 2] m. Figure 4.6 is a screenshot from the simulation. The simulation will be run with three different values on the restitution, 0.0, 0.5 and 1.0.
- In the second part the sphere will be placed so that the following equation holds $v \cdot h \geq d$ where v is the initial velocity, h is the time step and d is the distance. This approach will be tested with different values of the restitution coefficient to see if the constant describes the measured behavior. The restitution of the plane is set to 1.0 in each test.

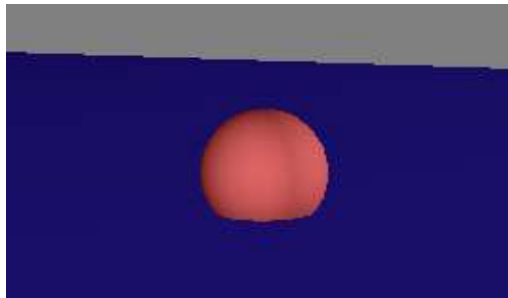


Figure 4.6: Screenshot of the bounce test part 1.

Result

Part1 - Discussion

The physics engines have different approaches to solve this problem, it is hard to say what the correct behavior is. This depends a lot on the approach of the solving method. Some methods will prevent penetration from ever happening. Even if there will never be a penetration in the simulation, initial conditions could have been set where objects are penetrating.

- ODE gives the objects a velocity in the opposite direction of the supposed collision. The energy and the resulting velocity is the same for all three tested restitution coefficients, 0.0, 0.5 and 1.0.
- Newton also gives the object velocity in the opposite direction. One thing that differ from the simulation with ODE is that the energy and velocity is smaller. The simulation also shows that the velocity is reduced in each time step which means it must be damped. When observing the velocity values for each time step it is found that the velocity is reduced to a factor of 0.9999 every time step. This behavior is independent of the restitution.
- Novodex will not give the object energy or any velocity. For every value of the restitution the object is repositioned right above the plane.

Part2 - Discussion

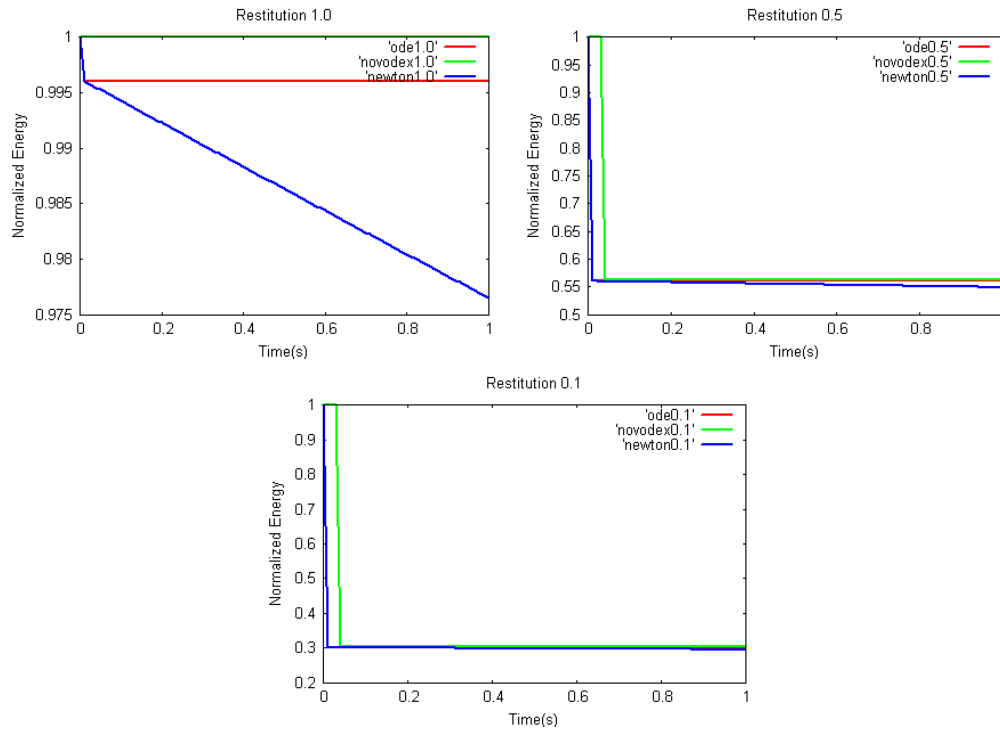


Figure 4.7: Bounce results with different values of restitution.

The difference in energy can be explained by the empirical law of frictionless collision:

$$v^+ = \epsilon \cdot v^- \quad (4.16)$$

Where v^+ and v^- is the velocity before and after the collision respectively. ϵ is the restitution coefficient. When the restitution is 1.0 the velocity is preserved but going in the opposite direction. This will preserve the energy as described in [Bar97]. In the first graph where the restitution is 1.0, it can be seen that all the engines preserves the energy. The following calculation will explain how the energy level will change in a collision when the the restitution is 0.5. When two objects are colliding the restitution coefficient is set to:

$$c = \frac{c_1 + c_2}{2} \quad (4.17)$$

where c_1 and c_2 is the restitution of object 1 and 2 respectively. There are other ways of setting the restitution but this is the most common way. When the restitution of object 1 is 1.0 and 0.5 for object 2 the restitution coefficient is 0.75 and the new velocity of the bouncing object will be set to 75% of the original velocity. The energy will then be 28% of the original, calculated in the following way:

$$E = \frac{m \cdot v^2}{2} = \frac{1 \cdot 0.75^2}{2} = 0.28125. \quad (4.18)$$

The same holds when the restitution is 0.1 which results in a factor of 0.15 of the original energy:

$$E = \frac{m \cdot v^2}{2} = \frac{1 \cdot 0.55^2}{2} = 0.15125. \quad (4.19)$$

The biggest difference in this test is that Newton loses velocity even though it should be constant, given this fact Newton has the worst results. Novodex has the correct energy values and is the best in the test. ODE also has good results but not as correct as Novodex.

4.3.4 Stability of constraints

Constraints are an important part of every physics engine when building complex scenes. For a complex scene to behave correctly all constraints must be fulfilled at all times.

Description

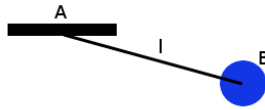


Figure 4.8: The pendulum setup

Having a pendulum setup like in figure 4.8. A is a fixed point, B is a spherical body which has a distance constraint to point A. The sphere falls from a horizontal position and the deviation in distance between the objects is measured during a period of 200 time steps. This is repeated with different masses of the sphere to see if the distance constraint is violated. The distance between the two spheres were set to 1 meter. During the simulation the mass of the moving sphere was changed from 1 to 10000 kg with an increase of 100 kg. Each sphere has a radius of 0.25 m.

Result

All the toolkits handle this test with little deviation. As seen in the plot, figure 4.9. Novodex oscillates a little which doesn't look that good. These oscillations are nothing that shows up in the simulation.

In the ODE simulation the distance deviation increases slightly for each mass. One might think that there exist a maximum weight according to the graph, but when testing ODE with very heavy objects (1000000kg!) the constraint still holds.

This test doesn't show the whole truth. There are many more things that are important when running a simulation with constraints so these results should be read with the other constraint tests in mind.

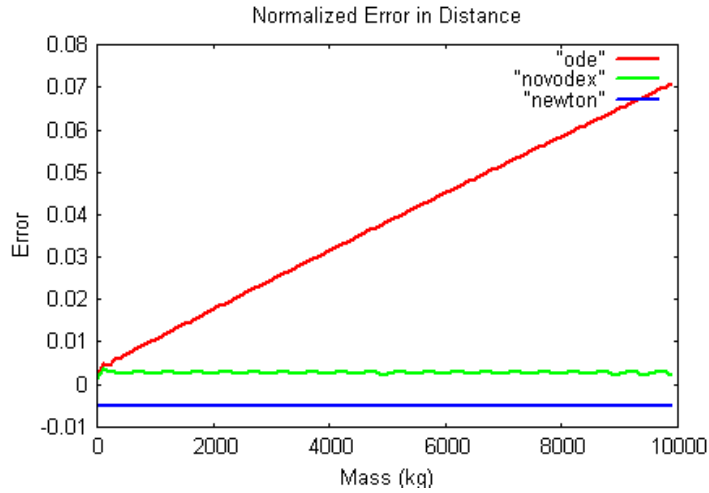


Figure 4.9: Error in distance

A factor to bear in mind is that there are ways to tune the parameters to make the simulation better, e.g., in ODE you can set an ERP parameter which might make the simulation more close to a horizontal line.

4.3.5 Basic accuracy

Testing the physics engines with some physical fact in the real world, might give a hint on how correct the engine is. This test will have a simple configuration of a pendulum then each engine's simulation will be compared to the "correct" physical behavior.

Description

Knowing the period of the pendulum:

$$\mathbf{T} = 2 \cdot \pi \sqrt{\frac{\mathbf{l}}{\mathbf{g}}}. \quad (4.20)$$

one can run the simulation to measure the deviation in time when the pendulum is vertical. Let the pendulum cross the vertical line 100 times when measuring the deviation.

Result

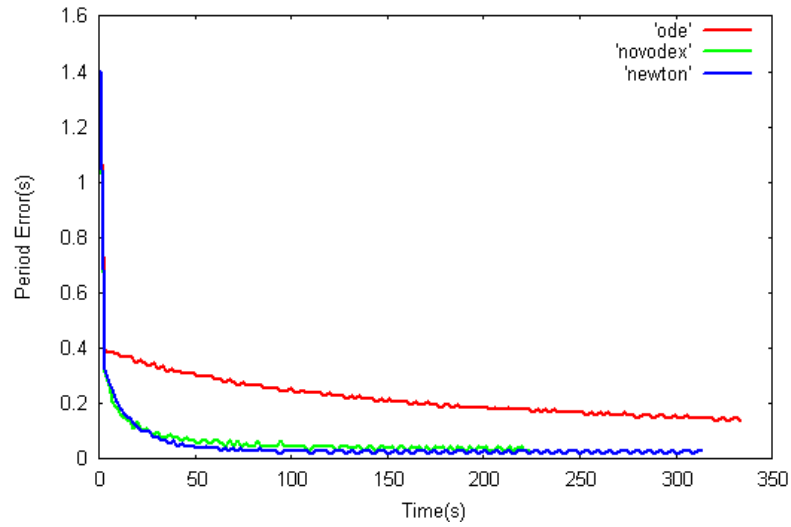


Figure 4.10: Period time error

It is hard to select the best engine by looking at the period time error in figure 4.10. Novodex and Newton have similar results. ODE has a little higher period time error.

An explanation of the peak in the beginning of the graph is that there are some initial time to set up objects.

To further analyze this test we have to look at more graphs. The following graph shows how well the physic engines conserves the energy.

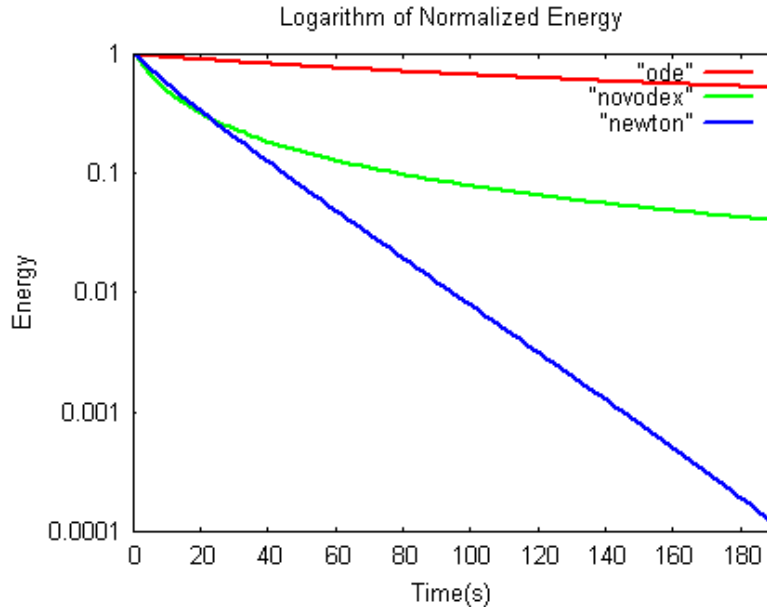


Figure 4.11: Energy

As can be seen in figure 4.11 Novodex and Newton are still similar in this graph, both loses almost all the energy in the simulation. Newton loses almost all energy and Novodex conserves a little. ODE conserves the energy best. This graph is interesting and raises the question if this loss in energy have any visual effects in the simulation.

In figure 4.12 the position of the pendulum is plotted. The graph shows the position in the x-dimension which is the interesting dimension since the pendulum is rotating around the y-axis. The graph shows what was suspected when looking at the energy in figure 4.11, both Novodex and Newton decrease the pendulum movement. Newton will eventually stop the pendulum from moving. Compared to the other ODE only loses little of it's movement.

Looking at the whole picture, ODE handles this test best. Novodex and Newton is way behind but Novodex has better results than Newton.

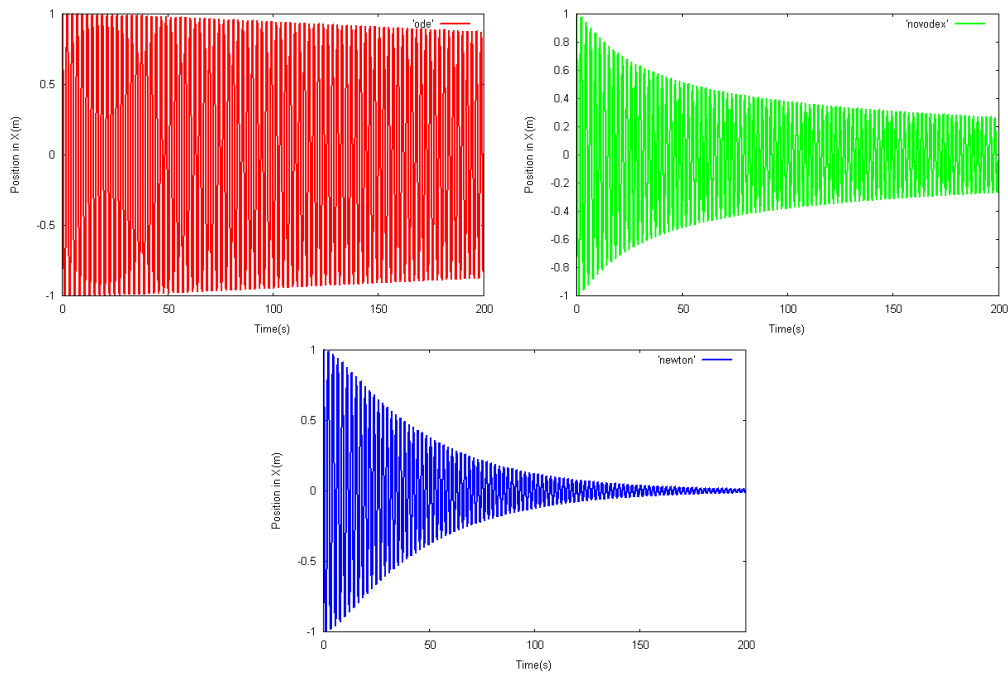


Figure 4.12: The trajectory of the pendulum in the x-axis

4.3.6 Scalability of constraints

The purpose of this test is to see how fast and accurately the physics engines can handle multiple joints. This is a very hard task to perform both fast and correctly.

Description

To test this a scene containing a chain of spheres connected with ball and socket constraints is created. The constraints are positioned in the middle between two spheres. The first sphere is then fixed so that the other spheres will swing under the fixed sphere as a pendulum.

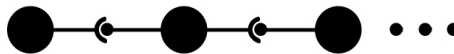


Figure 4.13: Chain of ball socket constraints

The execution time for a time step taken by each physics engine will be measured. Then an average will be calculated to present the time/constraint it takes for a simulation step. Also, the average error in constraint violation will be measured. This is done by calculating the error in distance between the spheres.

The spheres were positioned on the x-axis with 1 meter distance between them. The masses were set to 1 kg and the gravity to $[0 \ -10 \ 0]N$.

Result

The tests were performed with up to 20 joints in a chain. Novodex behaved well in the test. When the number of joints were close to 20 some elastic effects could be seen. Newton behaved very bad in this test, when there were more than three joints in the scene the simulation wasn't stable and the spheres moved around in a chaotic way. When ODE was first tested the ERP and CFM parameters were set to default. This resulted in spring effects where the joints looked elastic. The chain bounced and the constraint error was very large when the chain consisted of 20 joints. For this reason we did the test with ODE again but this time the ERP and CFM parameters were set to respectively 1 and 0. This time the error was much smaller but it was still much bigger than the error produced by Novodex. All these results are presented in figure 4.14.

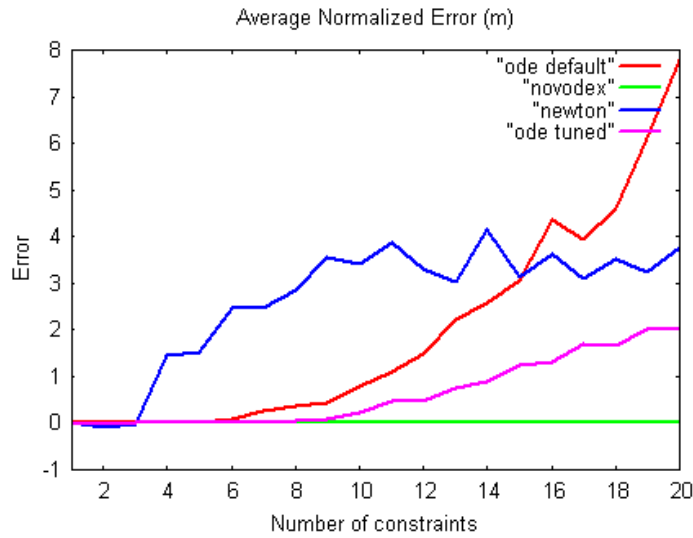


Figure 4.14: Constraint error

The next part of this test was to see how fast the engines solve each time step. This was done by calculating the total time for a step made by the physics engines and then divide this by the number of constraints in the scene. For scenes with a small number of constraints the overhead time for the step is visible but the time per constraint converges when the number of constraints get close to 20. ODE was the fastest followed by Novodex and Newton almost equally fast.

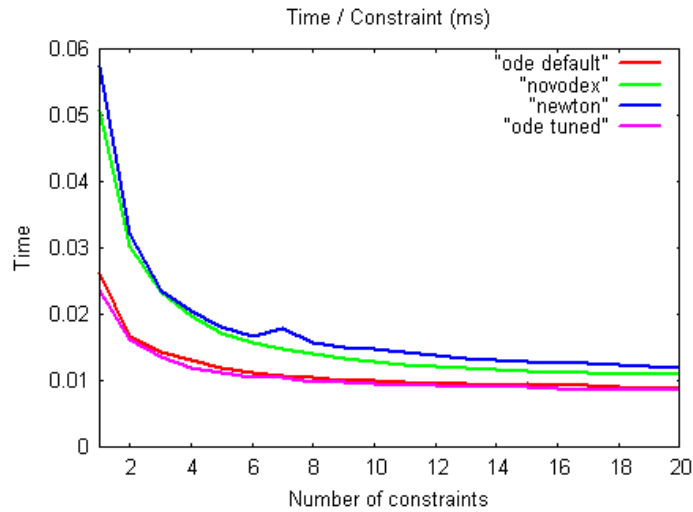


Figure 4.15: Average time per constraint

The best engine in this test is Novodex which could handle 20 constraints in a chain without large errors in the distance. ODE was a little bit faster but that isn't as important as computing the constraint forces correctly. ODE was second best because it was the fastest engine and kept the chain connected even though it was stretched with fairly large errors. Newton did not perform well, the spheres in the chain started to move unnaturally already at three constraints.

4.3.7 Scalability of contacts

A common way to demonstrate a physics toolkit is to have a scene with a big pile of boxes which is then destroyed. This scene is not always that common in real applications. Despite this, piling seems to be a way of measuring how good a physics toolkit is, the following test will look into how the different toolkits scale when boxes are piled on top of each other.

Description

In this test the average time of colliding objects is measured. 1 to 20 boxes with equal distance between them were dropped on a fixed plane. The computational time of a time step is measured and the average time per colliding pair is calculated.

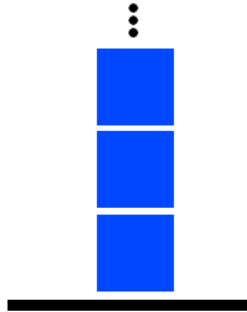


Figure 4.16: The pile will be initialized with some space between each box.

Result

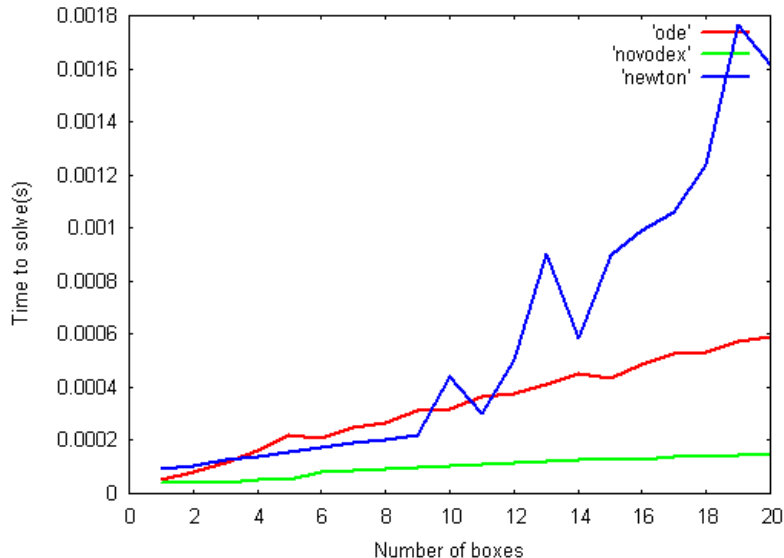


Figure 4.17: Time to solve contacts

Both Novodex and ODE increases linearly when boxes are added. Novodex is the fastest of them all, ODE is second best. Newton is slower than the other two and also has a steeper slope. This means that the computation time per colliding pair increases faster resulting in a more expensive simulation. The oscillations in Newtons graph could be explained by instability in the pile resulting in less contact points.

In a real simulation where a pile is used time is just one factor that is important. Many would argue that the most important thing is how it looks, if the pile can stand without falling for a long time. The next test will look into this.

4.3.8 Stability of piling

This test is evaluated based on the same simulation as the previous test, but is a bit more subjective since the result will be based on observations during the simulation. It can sometimes be hard to say what is a good behavior. Nonetheless this is an important test that will be a part of the evaluation of the toolkits.

Description

The test consists of a scene with a large amount of objects (boxes) stacked on each other (or falling on top of each other). The boxes will have different sizes and masses to make the solving more difficult, because this problem is a simple problem for most physics engines. The simulations will be run and graded depending on how stable they are. It is hard to measure energy in this test due to the restitution coefficient. This test is similar to the tests made in the Gamasutra article[LJ00a, LJ00b].

Result

All the simulations look good. There is no simulation that behaves like any other. The pile of boxes may fall in different directions depending on which engine is used.

There are a lot of tricks that can be used to make the pile be more stable, e.g., freeze different parts of the pile that is not affected by any external forces.

With Novodex the pile falls after 15 boxes, Newton after 18 boxes and ODE after 13 boxes.

4.3.9 Testing complex contacts

This will test collision with trimesh objects and convex hull objects. The test will be divided into three different scenes which will test different kind of collisions. The types of collisions that were tested were.

- Standard primitives with static nonconvex triangle mesh.
- Convex triangle meshes with each other.
- Non-convex triangle meshes with each other.

As in the previous test it is hard to measure anything in this test. The grading will be based on how visually convincing the collisions look.

All scenes used a time step of 0.01 s.

Result

The first thing that was tested were standard primitives colliding with a static triangle mesh. The scene consisted of 6 objects (boxes and spheres) falling into a funnel.

ODE and Novodex worked satisfactorily while Newton had some penetration. The following figures illustrate the course of events in the different physics engines.

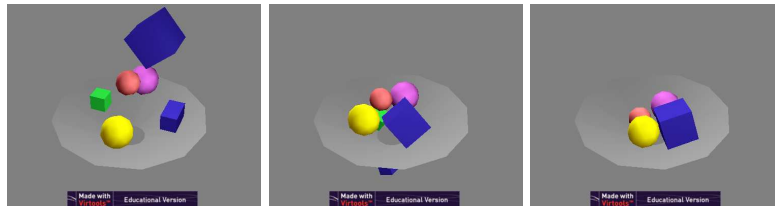


Figure 4.18: Novodex

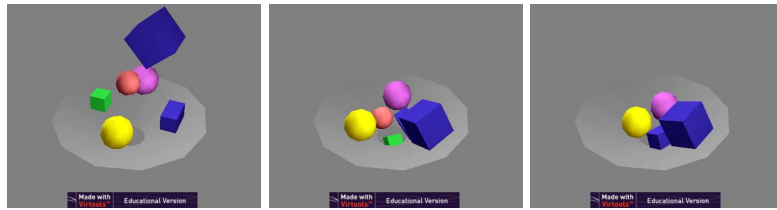


Figure 4.19: ODE

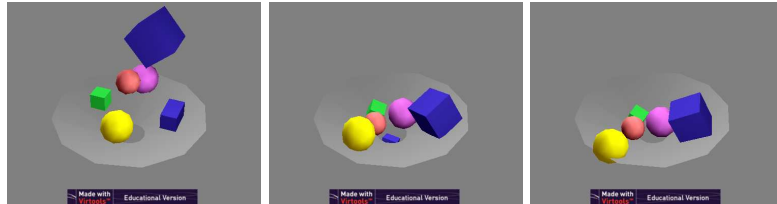


Figure 4.20: Newton

In the following scene collisions between convex meshes are tested against each other. Several convex meshes will be dropped onto a plane and the course of events will be observed and documented.

Novodex and Newton worked without any penetration or jittering. ODE have penetration resulting in large forces which makes some objects bounce of the plane.

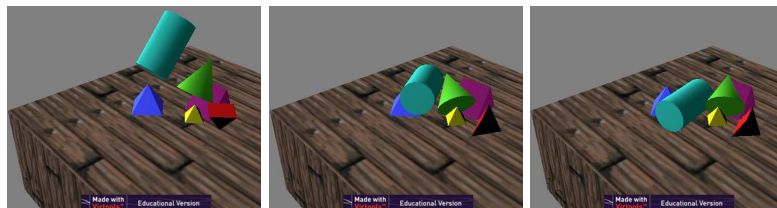


Figure 4.21: Novodex

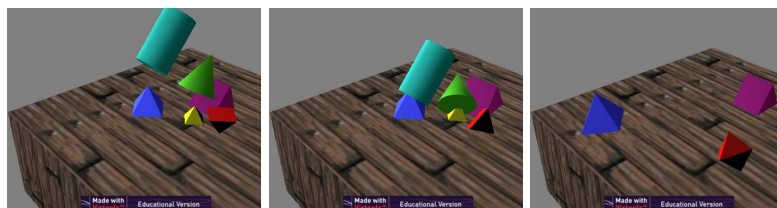


Figure 4.22: ODE

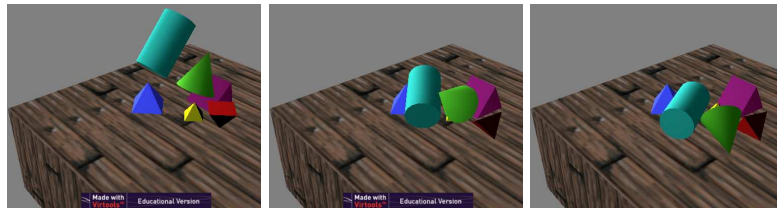


Figure 4.23: Newton

The third and final part of this test will test collision between nonconvex objects. A chain of five toruses is created, each with the weight of 0.2 kg. Since Newton doesn't support this kind of collision it is left out of this test.

Novodex handles this test good while ODE has some problems. The contact points are calculated correctly but the objects still penetrate each other and the toruses are separated.



Figure 4.24: Novodex



Figure 4.25: ODE

4.4 Final result

Testing physics engines is not an easy task. There does not exist a general test that would satisfy every developers needs. There are a lot more things that could be tested, which might be more important in one situation while not as important in other. The tests presented here are however important and a physics engine should be able to handle these.

Another important thing to remember is that different physic engines aims towards different areas. One big area is the game industry. In this area it is more important that the physics calculations are fast than correct.

Table 4.20 summarizes the results of the runtime tests. 1 means that this physics engine handles the test best. If there are a '-' instead of a number, this means that the engines handle the test equally good or bad. The grading is based upon our impression of how the physic engines handle the tests. This is mostly based upon how different attributes (energy conservation, angular momentum, trajectory) resembles the "correct solution".

#	Test	Novodex	ODE	Newton
1	Testing dry friction	-	-	-
2	Gyroscopic Forces	1	3	2
3	Bounce	1	2	3
4	Stability of constraints	-	-	-
5	Basic accuracy	2	1	3
6	Scalability of constraints	1	2	3
7	Scalability of contacts	1	2	3
8	Stability of piling	2	3	1
9	Testing complex contacts	1	3	2
Total		9	16	17

Table 4.20: Grading of the physic engines in the runtime tests.

Looking at table 4.20 Novodex is the physics engine that handles the runtime tests best. Recall that Novodex also had the best results in the theoretical evaluation. This will make Novodex the physics engine with best score in total. ODE and Newton also received good results in the tests. It is important to remember that both these engines are free for commercial use while Novodex isn't. ODE and Newton both have problems with stability in some situations, Newton doesn't work well with multiple constraints and ODE have problems with collision between mesh objects. Another drawback with Newton is that it doesn't handle general meshes, only convex meshes. ODE handles general meshes, but have problems with penetration of objects resulting in unnatural behavior. Novodex handles collisions between general meshes best, this might be because Novodex has a special structure for meshes. Newton and Novodex damps the simulations to keep things stable, which is visible in test 5, Basic accuracy, where the pendulum almost stops. Newton damps much more than Novodex, which is visible in both test 2 and 3 where the energy clearly decreases.

4.5 Discussion

It is now time to choose the physics engine that is going to be used in this project. The commissioners would prefer a physics engine that is free for commercial use and for that reason Novodex will not be chosen for further implementation. Novodex would otherwise be the obvious selection.

When using a physics engine in a project it can in many situations be just as good to use a free physics engine instead of an engine that has a license cost. It is important to investigate the situations that should be simulated by the physics engine before choosing so that the most suitable is selected.

In the theoretical test ODE did better than Newton because of the usability. This fact is important when implementing the physics engine.

The physics engine that will be used in the implementation is ODE. This physics engine has all the desired features the implementation will include. It is easy to use with good documentation and a lot of test implementations. ODE also has an active community and the development of the engine is ongoing. All factors have played a role in choosing this physics engine.

Chapter 5

Results

This chapter will present the progress of the work and look into how well the requirements have been fulfilled.

5.1 Accomplishment

Here the progress of the work will be presented. The progress can be divided into three phases, Preliminaries, Software Development and Evaluation and Testing.

5.1.1 Preliminaries

The work started with an introduction on how to write plugins for Virtools. After that the requirements on the physics module had to be established. These requirements then formed the base for the design phase. The design was reviewed and approved by the supervisors and commissioners. The final design is the one described in this report and used in the implementation.

5.1.2 Software Development

Once the design for the module was finished, the implementation started by testing simple communication functions between Virtools and the manager. When this worked the manager and the connections to the different physics engines were implemented. During the implementation phase a lot of testing and debugging was done. More about this can be read about in the design chapter.

5.1.3 Evaluation and Testing

The evaluation of the physics engines were done mostly by reading information from the different physics engines web pages. In addition to that documentation, sample programs were downloaded and studied and we looked at the source code from the open source engines. We also searched a lot for information on the available forums and mailing lists. Since the engines had documentation of varying quality it was sometimes hard to find the information that we were looking for.

The testing of the physics engines were done by creating scenes in Virtools and then

running them with the different physics engines. This made it very easy to verify that the basic conditions were the same for all engines.

5.2 Requirements fulfillment

A number of requirements was presented in chapter 3. These requirements are presented here with additional information about the status.

5.2.1 Functional Requirements

- *Dynamic Objects, objects that is influenced by the physics.*
Works.
- *Static Objects, objects that can't move but is still influenced by physics.*
Works.
- *Composite objects, Objects consisting of more than one geometry.*
Not Implemented.
- *Support for the following primitives: sphere, box and triangemesh.*
Support for box, sphere, mesh have been implemented.
- *Joints, at least the following joints should be supported: Ball-Socket, Hinge, Slider.*
All these joints have been implemented.
- *Be able to set different physical aspects on objects such as friction and elasticity*
Works.
- *Deactivate/active objects.*
Object can be unPhysicalized which will deactivate the object.
- *Be able to set impulses and torque to objects.*
Works.
- *Set the position of an object, without changing the physical properties.*
Works.

5.2.2 Non-functional Requirements

- *The plugins should work well with Virtools.*
The plugin works with Virtools according to the user guide in Appendix A. The functionality works well in the test programs that have been built.
- *The software layer should not add too much overhead.*
Some overhead exists which is inevitable. Because the physics module is a plugin to Virtools some special functions must be called to make the plugin work. This gives overhead to the program which is unavoidable.
- *The software should use minimum amount of memory and computational power.*
This has been followed to the extent that it is possible. Some extra memory and computational power is unavoidable because of the overhead to map between Virtools and the physics engines.

5.2.3 Requirements for testing

The last set of requirements where requirements set to be able to evaluate the physic engines in Virtools.

- *Be able to fetch the energy from objects, both potential and kinetic.*
Works and is implemented as a building block.
- *Be able to set the linear and angular velocity.*
Works and is implemented as a building block.
- *Fetch other parameters, e.g. time-step, distance between positions.*
Works and implemented as a building block.
- *Print fetched data to file.*
Works and implemented as a building block.
- *As far as it is possible, features are going to be implemented as building blocks.*
All functionality have been implemented as building blocks.
- *A physics engine should be easily changed to another.*
To change a physics engine, Virtools has to be restarted and the physics engine that is going to be used has to be specified in a file. See the user's guide in Appendix A for full description.

Most of the requirements has been implemented in the plugin. All the requirements have been implemented for each physics engine that is used in the run-time tests.

Chapter 6

Conclusions

This chapter contains a discussion about the master thesis, description of limitations in the plugin and some ideas about future work.

6.1 Discussion

This report have described the evaluation of physics engines and the implementation of a plugin for Virtools.

Our ambition has been to make tests that could be a foundation for testing physics engines. We believe that these tests form a good base for evaluating physics engines, since so many aspects are tested. The results from our tests show that there are differences between physics engines when it comes to scalability and stability. The tests also shows that physics engines handle certain areas better than others.

The design of the software plugin is modular where it is easy to use different physics engines. This design is a good foundation for further testing, either by more tests or by writing a new connection for a different physics engine. A possible extension of this plugin is the ability to run more than one physics engine at a time in the same simulation. Doing this would have made it easier to judge the more subjective tests where the result depend on the authors opinion.

The project resulted in a functional software plugin for Virtools. This plugin has already been used in a project. This is an implementation of the famous “Labyrinth Game”, where a ball is maneuvered through a maze while avoiding falling down holes. The game works well with the plugin and will be used as a part in a larger project.

6.2 Limitations

The biggest limitation in the physics module is that the collisions between mesh objects often results in unwanted behavior.

6.3 Future work

Composite objects was the only requirement that wasn't implemented in the module. The reason is that we had problems with representing the grouping of the composite objects in the Virtools interface.

A possible extension that was mentioned earlier is the ability to run more than one physics engine in the same simulation. This approach would not allow objects to interact with objects simulated by another physics engine. To add this functionality to this project would require some modifications and additions to the current design.

Since ODE still is being developed, the connection between the physics manager and the physics engine have to be updated as well. On the ODE web page it is stated that the triangle mesh class is not final and that API changes might be expected. If the changes are made in ODE the connection between ODE and OpenPhysics should be adjusted to receive better collision between mesh objects.

It is also possible to add more parametrisation for ODE into the manager. This would make it easier for the users to modify the behavior of the engine so it suits the developers needs. One example of this can be to change the softness of the surfaces on the objects.

Chapter 7

Acknowledgments

We would like to thank our supervisor at VRLab Anders Backman. We would also like to thank Kenneth Holmlund at VRLab for explaining much of the physics used in the testing.

Special thanks to Claude Lacoursieré at VRLab who designed test 1 to 8 and for explaining the theory and answering our questions.

Our supervisors at Q-Life, Kalle Jalkanen and Markus Häggqvist. John Waterworth and Eva Lindh-Waterworth for letting us do our master thesis at Q-Life.

Finally we would like to thank Annie Hansson and Johanna Lundström for their support and patience.

References

- [AGI] AGIEA. Novodex. <http://www.ageia.com/> (visited 2005-11-28).
- [Axi] True Axis. True axis. <http://www.trueaxis.com> (visited 2005-11-28).
- [Bar97] David Baraff. An introduction to physically based modeling: Rigid body simulation ii. nonpenetration constraints. Technical report, Robotics Institute, Carnegie Mellon University, <http://www.cs.cmu.edu/~baraff/pbm/rigid2.pdf>, 1997.
- [Bul] Bullet. Bullet continuous collision detection and physics library. <http://www.continuousphysics.com/Bullet/> (visited 2005-12-15).
- [Cla19] LaCoursière Claude. Stabilizing gyroscopic forces in rigid multibody simulations. 19.
- [Dyna] Dynamecs. Dynamecs. <http://dynamechs.sourceforge.net/> (visited 2005-11-28).
- [Dynb] Newton Game Dynamics. Newton game dynamics. <http://www.physicsengine.com> (visited 2005-10-11).
- [gP] Tokamak game Physics. Tokamak game physics. <http://www.tokamakphysics.com> (visited 2005-10-11).
- [Ins] The Interactive Institute. The interactive institute. <http://www.tii.se> (visited 2005-10-27).
- [Ken02] Erleben Kenny. Module based design for rigid body simulators. Technical report, University of Copenhagen, Denmark, <http://www.diku.dk/publikationer/tekniske.rapporter/2002/02-06.pdf>, 2002.
- [Ken05] Erleben Kenny. Stable, robust, and versatile multibody dynamics animation. Technical report, PhD thesis, Department of Computer Science, University of Copenhagen (DIKU), Universitetsparken 1, DK-2100 Copenhagen, Denmark, <http://www.diku.dk/~kenny/thesis.pdf>, 2005.
- [LJ00a] Hecker Chris Lander Jeff. Product Review of Physics Engines, Part One: The Stress Test. Technical report, http://www.gamasutra.com/features/20000913/lander_01.htm, 2000.
- [LJ00b] Hecker Chris Lander Jeff. Product Review of Physics Engines, Part Two: The Rest of the Story. Technical report, http://www.gamasutra.com/features/20000920/lander_01.htm, 2000.

- [Mar] Fred Marcus. What designers need to know about physics. http://www.gamasutra.com/resource_guide/20030121/marcus_01.shtml (visited 2005-10-27).
- [MB95] Canny John Mirtich Brian. Impuled-based simulation of rigid bodies. 1995.
- [ODE] ODE. Open dynamics engine. <http://www.ode.org> (visited 2005-09-14).
- [Ope] OpenTissue. Opensource Project, Physical based Animation and Surgery Simulation. <http://www.opentissue.org>.
- [QL] Q-Life. Q-life. <http://www.tii.se/qlife> (visited 2005-10-27).
- [Sha01] Lawrence Pfleeger Shari. Prentice Hall, 2001.
- [SJ98] Schömer Elmar Sauer Jörg. A constraint-based approach to rigid body dynamics. 1998.
- [Vir] Virtools. Product specification. <http://www.virttools.com/solutions/products/index.asp> (visited 2005-09-09).
- [WJ04] Waterworth Eva Lindh Waterworth John. Relaxation island: A virtual tropical paradise. *Proceedings of BCS HCI2004: Designing for Life*, a:a, 2004.

Appendix A

User's Guide

This physics pack loads a physics engine when Virtools is started. This is configured in an initialization file located in the windows folder. This file has to be called OpenPhysics.ini and must be in the following format:

```
[Settings]
PhysicsEngine=ODEConnection.dll
```

Where ODEConnection.dll can be changed to a one of the following physic connections.

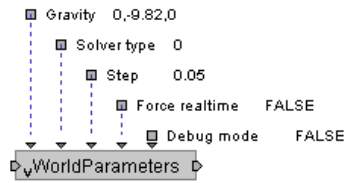
- ODEConnection.dll
- NovodexConnection.dll
- NewtonConnection.dll

All the DLL-files needed by a physics engine must be placed in the subdirectory Building blocks under Virtools installation path.

To start using the physics plugin developed in this project in Virtools, the first thing to do is to add the building block(BB) *WorldParameters* to a level script, see description of the BB below for more information.

This physics pack supports the following building blocks:

1. WorldParameters

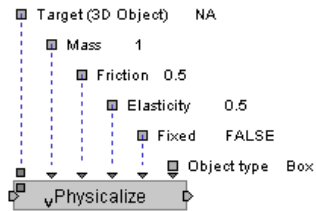


This BB initializes the physics toolkit. Different parameters can be set to change the behavior of the physics simulation.

The following parameters are supported:

- Gravity** (vector3) (0,-9.82,0), set the gravity.
- Solver type** (string) (first solver in the list), many physics toolkits offers different solver methods. If they do the solver can be changed to one that best simulates the current world.
- Step** (float) (0.05), The step size in milliseconds.
- Force realtime** (boolean) (FALSE), Can force the simulation to run in realtime.
- Debug mode** (boolean) (FALSE), Show debug information. For example contact points.

2. Physicalize



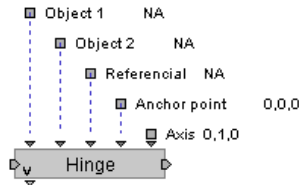
For an object to be part of the physics simulation in the world it has to be physicalized. To physicalize an object, simply drag the BB onto the object in mind, or drag the BB to the “level script” and set the object as target parameter.

The following parameters are supported:

- Object** (3dEntity) (NULL), the object to be physicalized

- (b) **Friction** (float) (0.5), the friction of the object. A higher value means higher friction.
- (c) **Elasticity** (float) (0.5), the restitution of the object. A higher value means that the object is more bouncy.
- (d) **Objecttype**, (choice) (BOX), How the object should be represented in the physicstoolkit. This can be Box, Sphere or mesh. If mesh is selected, the physics toolkit will represent the real object.

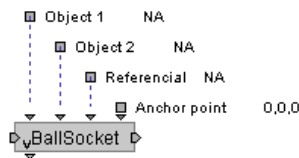
3. HingeJoint



This constraint will remove all but one degree of freedom, so that the objects attached is limited to move around one axis. This building block will output a joint id which can be used to remove the joint.

- (a) **Object 1** (3dEntity), First object to be constrained.
- (b) **Object 2** (3dEntity), Second object to be constrained.
- (c) **Reference object** (3dEntity), Position the joint relative this object.
- (d) **Position** (vector3), The position of the joint.
- (e) **Axis** (vector3), The axis around which the object should to move.

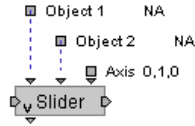
4. BallSocketJoint



This constraint will attach two objects at point a certain point. This building block will output a joint id which can be used to remove the joint.

- (a) **Object 1** (3dEntity), First object to be constrained.
- (b) **Object 2** (3dEntity), Second object to be constrained.
- (c) **Reference object** (3dEntity), Position the joint relative this object.
- (d) **Position** (vector3), The position of the joint.

5. SliderJoint



This constraint will attach two objects along an axis. This building block will output a joint id which can be used to remove the joint.

- (a) **Object 1** (3dEntity), First object to be constrained.
- (b) **Object 2** (3dEntity), Second object to be constrained.
- (c) **Reference object** (3dEntity), Position the joint relative this object.
- (d) **Position** (vector3), The position of the joint.

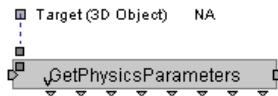
6. GetWorldParameters



This BB returns parameters that affect the world. The following output parameters are available:

- (a) **Elapsed time** (float), The elapsed time.
- (b) **Total Energy** (float), The total energy in the system.
- (c) **Potential Energy** (float), The total potential energy in the system.
- (d) **Transitional Energy** (float), The total transitional energy in the system.
- (e) **Rotational Energy** (float), The total rotational energy in the system.

7. GetPhysicsParameters



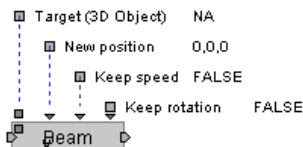
This BB will produce output for a given object. The following input must be set:

- (a) **Object** (3dEntity), the object to fetch output from.

This BB will produce the following output:

- (a) **Velocity** (vector3), The velocity vector of this object.
- (b) **Velocity value** (float), The length of the velocity vector.
- (c) **Angular velocity** (vector3), The angular velocity of the object.
- (d) **Total Energy** (float), The total energy of the object.
- (e) **Potential Energy** (float), The potential energy of the object.
- (f) **Transitional Energy** (float), The transitional energy of the object.
- (g) **Rotational Energy** (float), The rotational energy of the object.

8. Beam

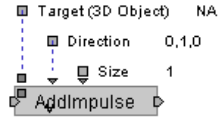


This BB will reposition an object to a specific location. This can also be done using Virtools builtin setPosition BB, but this building block will not have any effect if the object is physicalized, since the physics toolkit will reset the position.

- (a) **Object** (3dEntity), the object to be beamed.
- (b) **Position** (vector3), the new position
- (c) **Keep velocity** (Boolean), Check this if the object is going to preserve the velocity.

- (d) **Keep rotation** (Boolean), Check this if the object is going to preserve the rotation and angular velocity of the object.

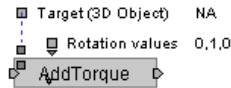
9. AddImpulse



Add a impulse to a specific object.

- (a) **Object** (3dEntity), the object to which the impulse is added.
- (b) **Direction** (vector3), the direction of the impulse.
- (c) **size** (float), the size of the impulse.

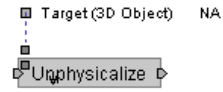
10. AddTorque



Add a torque to a specific object.

- (a) **Object** (3dEntity), The object to which the torque is added.
- (b) **Torque** (vector3), The value of the torque to be added.

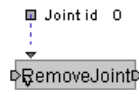
11. unPhysicalize



Remove the object from the physics simulation. The object will still be visible but will not be influenced by any physics.

- (a) `Object (3dEntity)`, The object to be unPhysicalized.

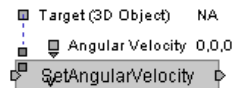
12. removeJoint



Remove a joint between two objects.

- (a) `Joint id (int)`, The id of the joint to be removed.

13. setAngularVelocity

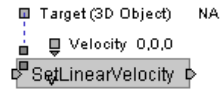


Set the angular velocity of an object.

- (a) `Object (3dEntity)`, The object to add angular velocity to.

(b) **Angular velocity** (vector3), The angular velocity.

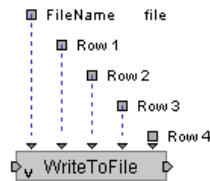
14. **setLinearVelocity**



Set the linear velocity of an object.

- (a) **Object** (3dEntity), The object.
- (b) **Velocity** (vector3), The velocity.

15. **writeToFile**



Write data to a file on disk. The file will be placed in the Virtools directory. If there already exists a file with the same name, the new information will be attached to the file. On each row in the file the different parameters Row1-4 will be written if they are defined.

- (a) **File** (string), The name of the file to write to.
- (b) **Row1** (string), First row in the file.
- (c) **Row2** (string), Second row in the file.
- (d) **Row3** (string), Third row in the file.
- (e) **Row4** (string), Fourth row in the file.

Appendix B

UML diagrams

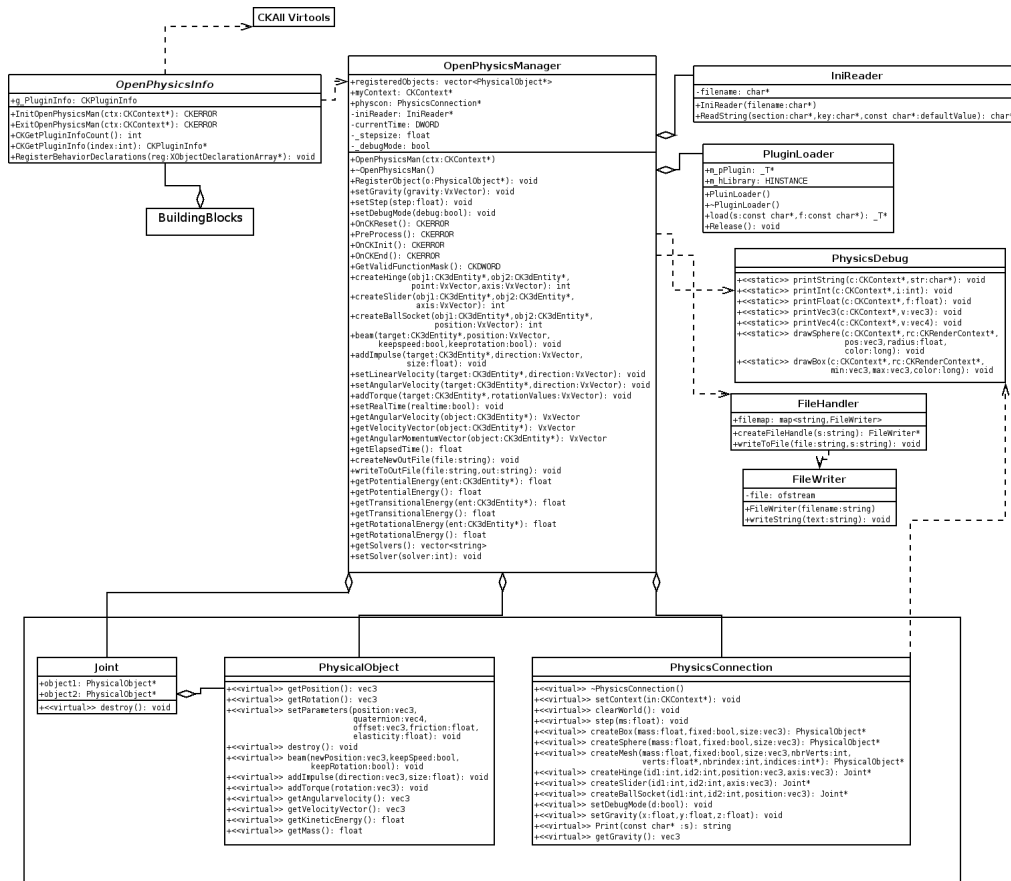


Figure B.1: UML-diagram of the manager.

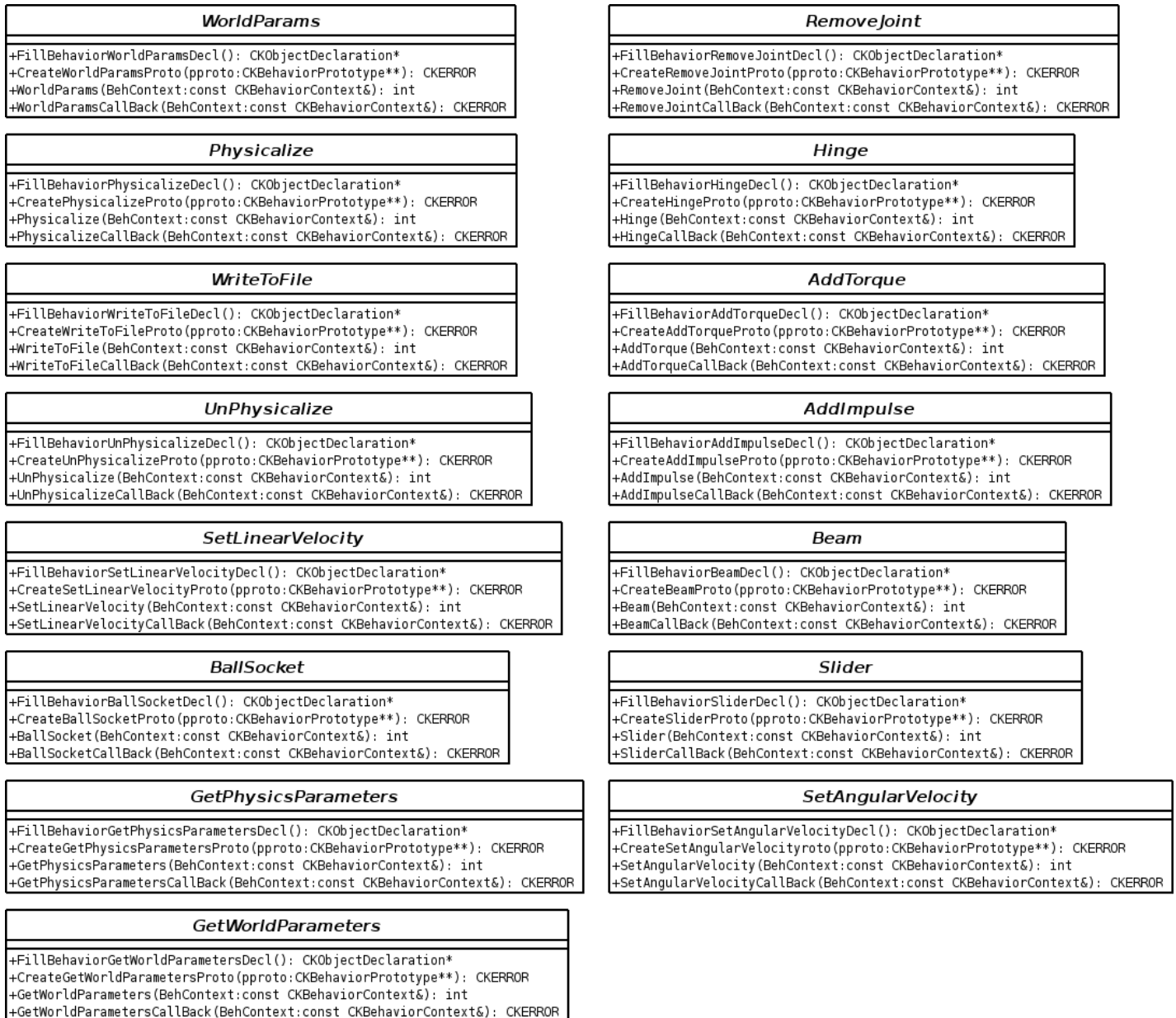


Figure B.2: UML-diagram of the building blocks.

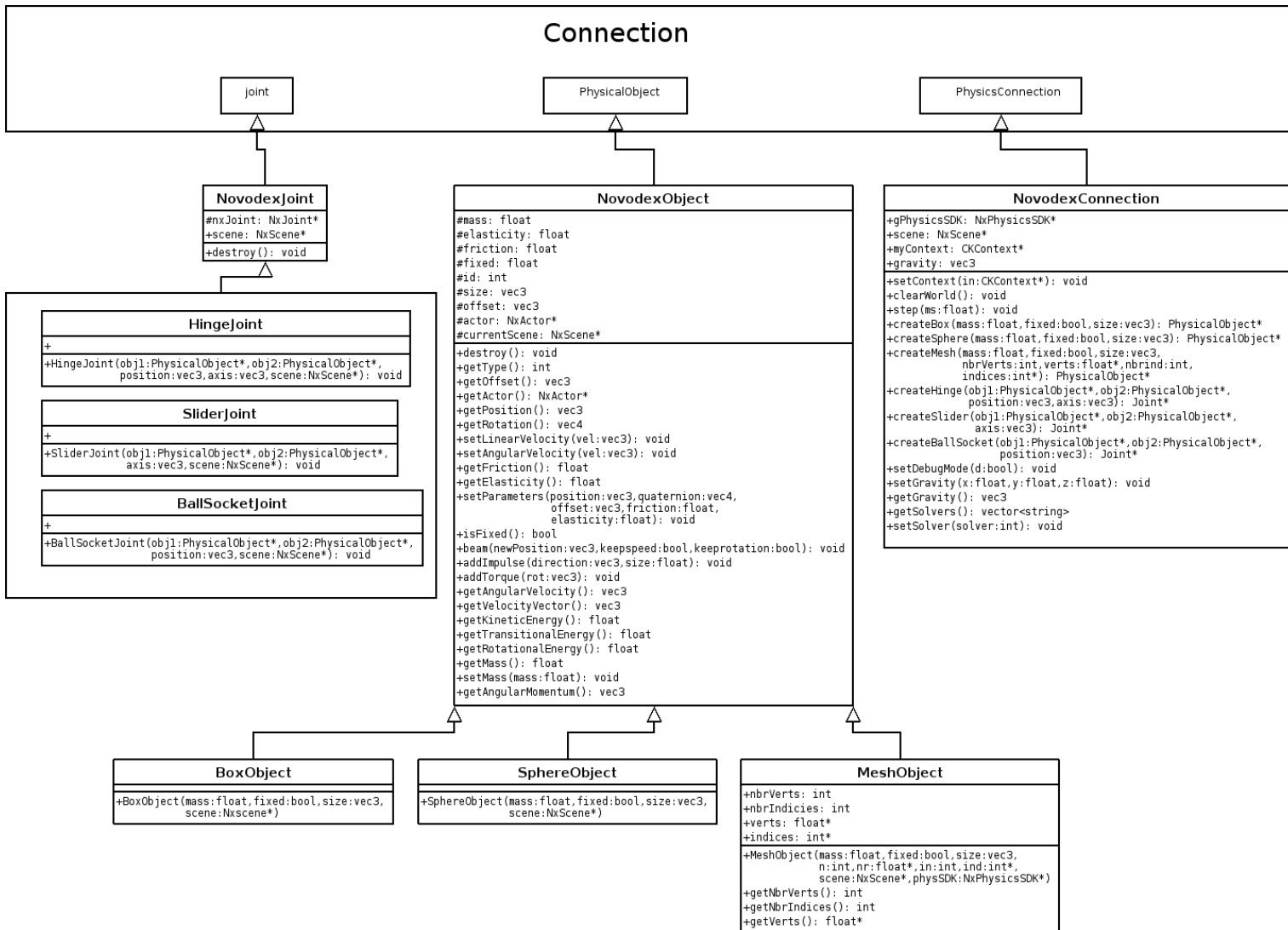


Figure B.3: UML-diagram of the Novodex connection.

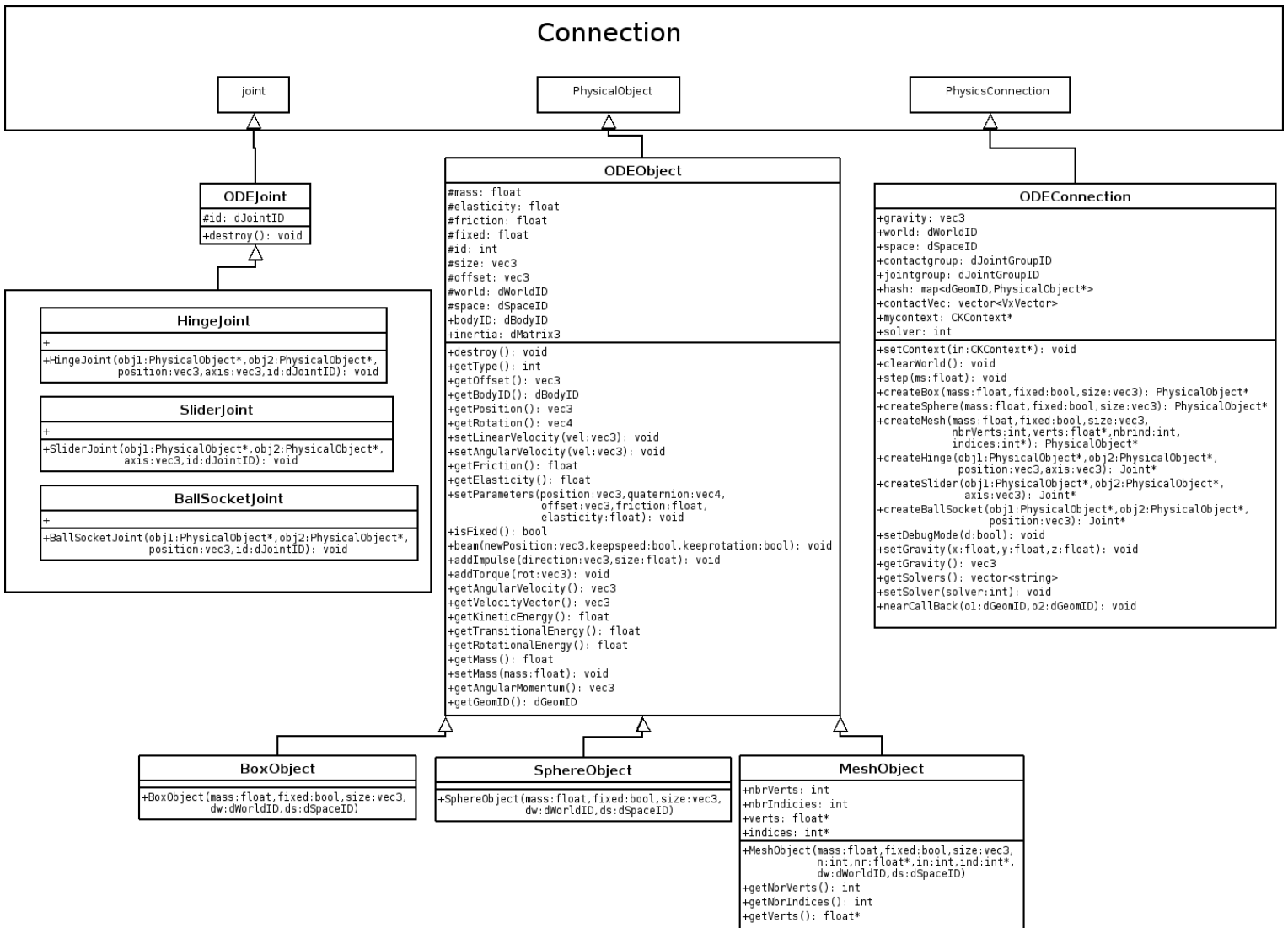


Figure B.4: UML-diagram of the ODE connection.

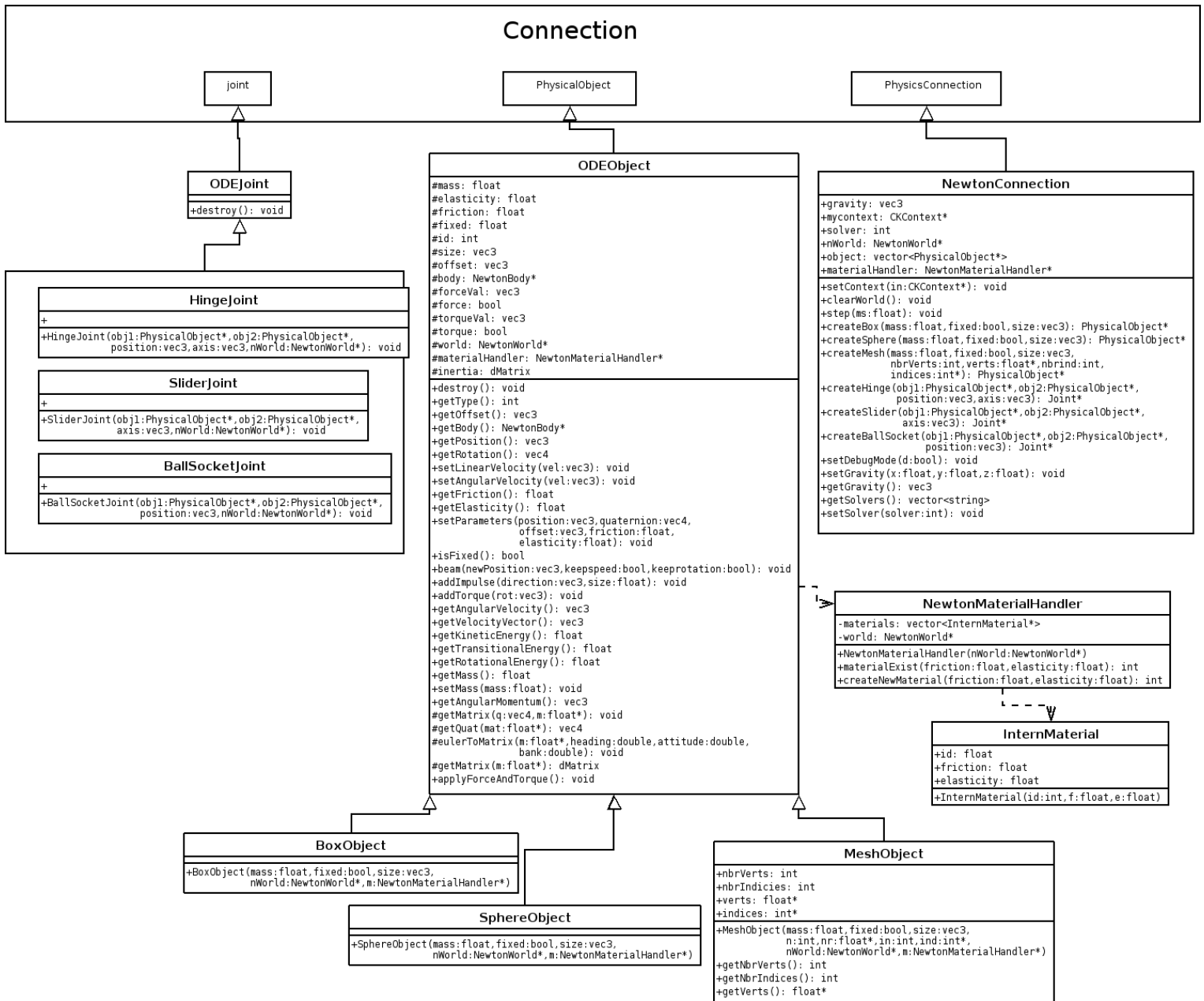


Figure B.5: UML-diagram of the Newton connection.