FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

COLLADA Physics Implementation for the pe Physics Engine

Johannes Bleisteiner

Bachelor Thesis

COLLADA Physics Implementation for the pe Physics Engine

Johannes Bleisteiner

Bachelor Thesis

Aufgabensteller:	Prof. Dr. U. Rüde
Betreuer:	Dipl.-Inf. T. Preclik
Bearbeitungszeitraum:	1.3.2011 – 26.3.2012

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 26. März 2012 ....

Abstract

This thesis aims to provide the **pe** physics engine previously developed by the Chair for System Simulation at the Friedrich-Alexander Universität Erlangen-Nürnberg with a means to read and write files in the COLLADA Physics format by the Khronos Group. This serves two purposes: the ability to checkpoint running simulations as well as to exchange data with third-party software.
In this thesis, we compare the concepts and data structures of **pe** and COLLADA respectively, specifically how the idea of instantiation in COLLADA relates to **pe**. We design and implement classes corresponding to COLLADA objects and develop a conversion scheme based thereon. Finally, we validate the finished program towards consistent data transference with itself and the Bullet engine. The results lead us to conclude that our extension is suitable for the intended purposes with some limitations to the data exchange functionality.

Contents

1 Introduction

Physics engines are software which allow a computer to simulate and predict the behaviour of real objects by representing them in a virtual space and subjecting them to the modeled physical laws. Most physics engines restrict themselves to specific situations like rigid-body, soft-body, or fluid simulations.

In 2006, Klaus Iglberger from the Chair for System Simulation at the Universität Erlangen-Nürnberg initiated the creation of a physics engine dedicated to massively parallel rigid body simulations, called the "**pe** Physics Engine" [1]. It is configurable to be either highly performant or highly accurate, depending on the requirements of the application. The project is still unconcluded, but has been successfully applied and has proven satisfactory. Currently development is focused on extending the **pe** beyond the basic functions and making it versatile with regards to possible simulation scenarios. It is also desired to make the **pe** compatible to third-party tools, allowing to employ it in a larger development process. To do so, it is essential that we enable it to emit and receive data in a common format.

However, there are no universally recognized and established standards yet in the world of physics simulation. In 2006, the COLLADA standard proposed by the Khronos Group [2][3], which previously served as a format to enable horizontal data transfer in a purely graphical context, was expanded to include description of rigid-body physical worlds. Though it has not been universally adopted, several influential and widespread physics engines, like the Bullet Physics Library [4], PAL [5] and PhysX [6], have started to use COLLADA Physics next to their proprietary data formats and use it to allow interoperability with outside sources and other engines.

It was the goal of this thesis to design and implement an extension to the **pe** library able to write and read all pertinent world data to/from valid COLLADA files. This would allow the **pe** to exchange data with other engines and with setup, review or post-processing tools.

This undertaking entailed recognizing the differences between the COLLADA format and the internal **pe** data structures and then finding a way to transform these differences. We designed and implemented classes and functions who, operating out of a **pe** context, import and export content in COLLADA files. Finally we subjected the finished code to a number of validation tests to ensure that the content was transferred faithfully. The Bullet Collada Converter [7] was used as a reference.

2 The pe physics engine

pe [1] is a rigid body physics engine developed by Klaus Iglberger and several other members of the Chair for System Simulation at the Friedrich-Alexander Universität Erlangen-Nürnberg. It offers a C++ classes framework to model rigid bodies and simulate their movement and collision. It may be configured to use one of several interchangeable algorithms to handle certain topics of the simulation, which vary in terms of performance, accuracy, or behaviour in specific situations. The engine can switch between modes with some very simple configuration changes, and is suitable both for scientific simulation as well as for game physics. It allows for large-scale simulations with up to several milliards of bodies, being natively parallelizable for both shared memory and distributed memory paradigms.

2.1 Materials and rigid bodies

In **pe** [1], each body in the simulation world is an instance of the RigidBody class (or one of its subclasses). All bodies are part of only one simulation stage.

To be created, a body needs to be specified by position, basic shape parameters, ID of the constituent material, and a "UserID" tag. Accessor methods allow to read all important data. Properties where it is natural, like position, orientation, velocity or spin can be modified freely after creation, while shape, material, or ID are read-only.

The geometric primitives that **pe** offers are: Box es, Sphere s, Cylinder s, spherocylinders (called Capsule s), and semispace-filling Plane s. A mesh type is planned, but not implemented currently.

To create more complex shapes, a Union can be used to conjoin multiple bodies. A Union is a container for several RigidBody s. The component bodies no longer act independently in the world, but share movement and acting forces with the Union as a whole, while their relative location is constant. Union s can be part of other Union s.

The behaviour of the body in collisions is determined by the Material it is attributed; the material determines variables like friction coefficients, mass (by way of density), the coefficient of restitution, stiffness, and others. Some of the parameters of the material are allowed to vary dependent on the material of the colliding body. Material s are defined in a global matrix-like data structure, allocating space for each such combination. It is to be noted that the data structure is designed for a smallish number of materials and scales badly for a larger quantity.

2.2 Constraints

Constraints establish restrictions on the relative movement of two bodies and allow the user to define more complex interactions than simple particle collisions. **pe** calls constraints Joint s and classifies them into several types, though at the time of writing, only the FixedJoint type is implemented. It represents a fixed connection between the two bodies, locking their relative displacement and rotation. Other types of Joint s would be slider, hinge, or ball-socket joints, which limit movement appropriately. Joint s ever only link exactly two bodies. Similar to bodies, it is not possible to modify a Joint once created except to delete it.

2.3 Example

While a prototype editor exists [8], the **pe** is primarily a code framework. Simulations are setup and run with a C++ program using the features of the **pe** code library.

```
#include <iostream>
#include <pe.h>

int main(){
        WorldID world = theWorld();
        //initialize the simulation world;
        world->setGravity(-0.1,0,0);
        //Set a gravity
        BodyID p = createPlane(1, 1,0,0, 0, copper);
        //create a ground plane with the normal (1,0,0), going
           through the origin , made of copper
        p->setFixed(true);
        //the plane is immovable
        BodyID box1 = createBox(1, 1,0,0, 2,2,2, oak);
        //create a box-shaped body with a side length of 2,
           centered on (1,0,0), made of oakwood
        box1->rotate(0.2,0,0);
        //rotate it by 0.2 radians around the x-axis
        BodyID s1 = createSphere(1, 2,4,0, 0.5, iron);
        BodyID s2 = createSphere(1, 2,5,1, 0.5, iron);
        //create two spheres of radius 0.5 at (2,4,0) and (2,5,1),
           made of iron
        affixFixedJoint(s1,s2,1);
        //impose a constraint on s1,s2.
        s1->setVelocity(0,-0.4,0);
        //give s1 a initial velocity
        world->run(100,0.5);
        //run 100 timesteps 0.5 long
}
```

Listing 1: Basic **pe** world setup example

3 The COLLADA standard

COLLADA [3] is a data format intended to transmit and exchange graphical designs and animations between different applications and development stages. It uses XML as a basic format and is developed and provided by the Kronos Group since 2004. With version 1.4, support for simple physics simulations was added.

3.1 Addressing schemes and instantiation in COLLADA

COLLADA uses a scheme where resources are first defined in libraries, and then instantiated where needed, instead of specifying the data inplace. A COLLADA document contains several libraries for objects like geometries, textures, bodies, materials, points of interest, scenes etc. For example, a body will instantiate the material it consists of to provide friction data, while a model will instantiate its constituent bodies and constraints.

To instantiate a `<*>` element (`<*>` being an instantiatable object element), one needs to create an `<instance_*>` element which contains a reference to the `<*>` element containing the definition of the resource in the appropriate library. An `<instance_*>` element also allows to reexamine and transform the object.

To be able to be referenced and instantiated, elements possess identifiers in the form of (at least) one of the attributes ‘id’ or ‘sid’, whose values are unique in the scope of the document (‘id’) or among its siblings (‘sid’). Referencing uses one of two addressing schemes:

- URI (Unique Resource Identifier) addressing: The target element must have an 'id' attribute, whose value is unique in the document. To reference it from inside the same document, it is sufficient to use that ID preceded by a hash sign. For elements located in another file, the syntax is more elaborate
- SID (Scoped Identifier) addressing: An address consists of zero or more 'sid' identifiers separated by slashes, which form a path leading to the relevant element. The path root is either an 'id' identifier or a "." indicating a relative address.

Some COLLADA elements use URI addressing, some use SID addressing.

3.2 `<technique>` and `<extra>`

COLLADA allows users to define their own data schemes, if they for some reason find the standard scheme lacking. The standard-compliant property data of an object is collected under a `<technique_common>` element. Next to it, it is possible to have alternative specification schemes in the form of `<technique>` elements, who can specify the data in any custom structure. Generally, it is recommended that they contain the same data scope as the common profile. `<technique>` elements possess a 'profile' attribute, and the COLLADA document may specify which profile, i.e. which set of `<technique>` elements is supposed to be used. If the `<technique>` is for any reason unusable, `<technique_common>` is to be used as an universally recognized fallback data provider, and as such must always be present and conforming to the standard.

This only concerns the content data of objects, though. Elements which are located outside of a `<technique_common>` or `<technique>` are invariant to user-specified data schemes, which is the case for those establishing object-relations.

Objects can also possess an `<extra>` element, which allows for any data to be stored, especially such which exceeds the scope of the `<technique_common>` profile.

3.3 COLLADA Physics

The overall container element of physics description is the `<physics_scene>`. Scenes consist of one or several models to instill them with content, which are defined by a `<physics_model>` element. These models contain bodies and constraints using `<rigid_body>` and `<rigid_constraint>` elements. It is also possible for a model to be part of another model by way of instantiation.

Scenes also feature a simple `<gravity>` property defining a global uniform forcefield. While COLLADA has preparations for more elaborate forcefields, there is no profile standard for them yet. It is also possible to define a `<time_step>` that is to be used in simulations.

Both `<physics_model>`s and `<physics_scene>`s are stored in their respective libraries. For these elements to actually appear in the final scene, they have to be instantiated. A `<physics_model>` is instantiated by a `<instance_rigid_model>` under the `<physics_scene>`, which in turn must be instantiated in the global `<scene>` element to be considered in effect.

<rigid_body>s and <rigid_constraint>s are instantiated by <instance_rigid_body> and <instance_rigid_constraint> elements as children of the <instance_rigid_model> of the model they belong to.

A body is represented by a <rigid_body> element and an <instance_rigid_body> instantiation thereof. Every <rigid_body> element contains at least one <shape> element containing either a geometric primitive or an instantiation of a mesh geometry defined elsewhere. If a <rigid_body> has several <shape> elements, they form a compound shape.

Properties like mass or material can be specified by other child elements either per geometry element, or for the entire body. The instantiation also allows to further refine some properties. In case of conflicting definitions, the data provided by the instantiation takes precedence over those by the <rigid_body> definition, who in turn takes precedence over the <shape> definition(s). If a value is left unmentioned and unspecified, the COLLADA standard prescribes a default behaviour.

Also, all <instance_rigid_body> elements link to <node> elements, which stand for points of interest in the graphical scene. This serves to couple the visual and physical components of a COLLADA scene. While the visual aspect of COLLADA is of no interest to us, these <node>s declare a coordinate system which the bodies observe. In other words, the coordinates of the body descriptions are given in a local coordinate system instead of the global one. Optionally, this can also be done for entire <instance_physics_model>s.

A material is defined by a <physics_material> element. It provides three properties <dynamic_friction>, <restitution>, and <static_friction>. The <physics_material> element can be located either locally in the <rigid_body> or <shape> where it is needed, or in the <library_physics_materials>, with the body containing an <instance_physics_material> reference.

In COLLADA, constraints are defined by a <rigid_constraint> element in a <physics_model> and subsequently instantiated just like bodies. <rigid_constraint>s carry two child elements <attachment> and <ref_attachment> linking to the halves of the <rigid_body> pair they impose themselves on. Constraints have property child elements determining the maximally allowed linear and angular displacement, the spring forces developed, and whether the bodies are allowed to interpenetrate. The constraint uses the coordinate system of the body linked to by <ref_attachment>.

The attachment references refer to <rigid_body> elements, not their instantiations [3] . If these <rigid_body>s however are instantiated more than once, which is allowed, it is problematic to determine the intended behaviour of the <rigid_constraint>, as <instance_rigid_constraint> does not give any more information.

3.4 Example

This example describes a box and a sphere with a constraint on their maximal distance.

```
<COLLADA>
  <library_physics_materials>
    <physics_material id="Mustard">
      <technique_common>
```

```
        <dynamic_friction>0.7</dynamic_friction>
        <restitution>1.0</restitution>
        <static_friction>0.8</static_friction>
      </technique_common>
    </physics_material>
  </library_physics_materials>
  <library_physics_models>
    <physics_model id="Model01">
      <rigid_body sid="Body01">
        <technique_common>
          <dynamic>true</dynamic>
          <shape>
            <box>
              <half_extents>2 2 1.4</half_extents>
            </box>
            <translate>0 10.5 0</translate>
            <mass>1.0</mass>
          </shape>
          <mass>2.0</mass>
          <instance_physics_material url="#Mustard"/>
        </technique_common>
      </rigid_body>
      <rigid_body sid="Body02">
        <technique_common>
          <dynamic>false</dynamic>
          <shape>
            <sphere>
              <radius>2 2 3</radius>
            </sphere>
            <mass>1.0</mass>
          </shape>
          <instance_physics_material url="#Mustard"/>
        </technique_common>
      </rigid_body>
      <rigid_constraint sid="Joint01">
        <attachment rigid_body="Model01/Body01"/>
        <ref_attachment rigid_body="Model01/Body02"/>
        <technique_common>
          <interpenetrate>false</interpenetrate>
          <limits>
            <linear>
              <min>-20 -20 -20</min>
              <max>20 20 20</max>
            </linear>
          </limits>
        </technique_common>
      </rigid_constraint>
    </physics_model>
  </library_physics_models>
  <library_physics_scenes>
    <physics_scene name="Main␣scene" id="Scene01">
      <instance_physics_model url="#Model01">
        <instance_rigid_body body="Model01/Body01" target="#Node01">
          <technique_common>
            <angular_velocity>0 0.238 0</angular_velocity>
            <mass>3.0</mass>
          </technique_common>
        </instance_rigid_body>
        <instance_rigid_body body="Model01/Body02" target="#Node02">
          <technique_common/>
        </instance_rigid_body>
        <instance_rigid_constraint constraint="Model01/Joint01">
          <technique_common/>
        </instance_rigid_constraint>
      </instance_physics_model>
      <technique_common>
        <gravity>0 -9.81 0</gravity>
      </technique_common>
    </physics_scene>
  </library_physics_scenes>
  <library_visual_scenes>
    <visual_scene id="Visual_Scene01">
      <node id="Node01"/>
      <node id="Node02"/>
    </visual_scene>
  </library_visual_scenes>
  <scene>
    <instance_physics_scene url="#Scene01"/>
  </scene>
</COLLADA>
```

Listing 2: COLLADA Physics example

The main elements of the file are libraries for visual and physical scenes, materials, and models, defining content.
In the `<library_physics_scene>`, there is one `<physics_scene>`, with a (optional) name “Main Scene” and an id “scene01”. This `<physics_scene>` is instantiated by the `<instance_physics_scene>` element under the `<scene>` element with the URL “#scene01”, referring to the ‘id’ attribute of the `<physics_scene>` (i.e. prefacing it with a hash sign). This declares it the valid scene (considering documents with multiple scenes).

`<physics_scene>` has two child elements: `<instance_physics_model>` and `<technique_common>`. The latter declares global data parameters, which in this example don’t exceed a `<gravity>` element setting the gravity to a (0,-9.81,0) vector. `<instance_physics_model`

url='#Model01'> instantiates the <physics_model> "Model01" and its contents. This model contains two <rigid_body> and one <rigid_constraint>, instantiated by the <instance_rigid_body>s and <instance_rigid_constraint> under <instance_physics_model>. These elements use SID addressing, where the initial path element is the id of the model, followed by a slash and the sid of the <rigid_body> resp. <rigid_constraint> element.

Body01 consists of one <shape>, which is a <box> of size (4, 4, 2.8). This box is subsequently moved (0, 10.5, 0) and imbued with a mass of 1.0. On the <rigid_body> level, the mass is again specified as 2.0, which takes precedence. Also, the body is assigned the material "Mustard". The corresponding <instance_rigid_body> element specifies the angular velocity as (0, 0.238, 0), representing a spin around the y-axis. For demonstrating purposes, it also overwrites the mass of the body again as 3.0. The <node> element it is associated with specifies no coordinate system transformation, so the position is considered as given in global coordinates.

Body02 is an ellipsoid, whose length along the z-axis is 3.0 and along the other axes 2.0. It is imbued with a mass of 1.0 and the "Mustard" material. <dynamic> is set to false, so the body is immovable. No <translate> vector is specified, so the body is centered at the local origin. Like with Body01, the associated <node> does not specify any transformations, it is located also at the global origin. There are no further properties specified in the instantiation.

These two bodies are constrained by Joint01, being referenced by its <ref_attachment> and <attachment> elements respectively. The contents of the <limits> element disallow Body01 to change its relative position towards Body02 by further than 20 along any axis in positive as well as in negative direction from its current state. Within the constraint, they are also not allowed to interpenetrate. They are however allowed to rotate freely.

The material "Mustard", which has been referred to already, is defined in the <library_physics_materials>, providing values for dynamic friction, restitution, and static friction to Body01 and Body02 (due being instantiated by them).

4 Comparison of data structures in pe and COLLADA

4.1 Overall structures

Both **pe** and COLLADA describe the simulated physical world by a number of rigid bodies and constraints/joints. COLLADA allows to logically group bodies into models and reuse them. **pe** does not possess such a mechanism.
Hence, creating such a structure in a written COLLADA file would require fine control by user-level code. Instead we limit ourselves to a simple structure, where each body is contained in a single model and represented by just one <rigid_body> and one <instance_rigid_body> element.

If we encounter such a structure in a foreign file, it is suitable to flatten these hierarchies, since such relations have no influence on the actual physics during the simulation. Though we are then unable to replicate the structure when we re-export into COLLADA.
For checkpointing purposes, this all is irrelevant, since files are both read and written by the **pe** itself, and no concepts foreign to **pe** are introduced.

Global parameters are stored on the `<physics_scene>` level and all models are contained therein, so it makes sense to correlate them to the **pe** World. However, **pe** has only ever one simulation world, while it is allowed to instantiate more than one of them in COLLADA. (Non-instantiated scenes do not pose a problem, since they are merely potential.)

4.2 Geometries

COLLADA and **pe** use the same basic geometric primitive shapes, suitable to model simple particles and composed structures of limited complexity.
For more detailed shapes, COLLADA provides elements to define meshes using vertex lists and polygons. **pe** however, does not possess the capability to handle mesh geometries. While an inclusion of these is planned and basic objects prepared, it is not yet implemented as of 2011. The import code is helpless encountering a COLLADA file containing mesh-type bodies.

pe is also of more limited scope in regards to the customization allowed to the primitives. In **pe**, spheres and spherical objects possess only a single radius property, while COLLADA allows elliptical shapes with different values for the semiaxes. COLLADA also allows for bodies to be hollow, a mechanic **pe** lacks completely.

pe and COLLADA use different definitions for planes. In COLLADA, they are true planes, but in **pe**, they encase all space on one side. They are characterized by a normal vector and a anchor point in **pe** and four real numbers representing the plane equation in COLLADA, respectively. While the conversion is trivial, the equation form loses information about which side is considered inside (although the signs may be an indicator in some circumstances) as well as the exact location of the anchor point.
To conserve this data, we design an alternative data scheme that will be included in a custom `<technique>` to the `<rigid_body>`

```
<rigid_body>
  <technique profile="pe">
    <shape>
      <plane>
        <normal>
          <!-- Normal vector composed of three values -->
        </normal>
        <anchor>
          <!-- Position of the anchor point -->
        </anchor>
        <displacement>
          <!-- Displacement of the plane with regards to the origin;
             Optional -->
        </displacement>
      </plane>
    </shape>
  </technique>
  <technique_common>
```

```
    <shape>
      <plane>
        <equation>
          <!-- Four values A,B,C,D representing the coefficients of
            Ax + By + Cz + D = 0
        </equation>
      </plane>
    </shape>
  </technique_common>
</rigid_body>
```

Listing 3: Custom `<plane>` scheme

4.3 Composite bodies

A COLLADA `<rigid_body>` contains one or several `<shape>` child elements defining its geometric properties. Properties like material can be specified both for individual shapes or for the entire body. In such a general case, the `<rigid_body>` is analogous to a Union in **pe**. A `<shape>` describes either a geometric primitive or refers to a more complex mesh shape. These sub-bodies would then be analogue to the RigidBody s (more specifically GeomPrimitive s) being part of this Union.

In case there is only one `<shape>` element, it is unnecessary to wrap the RigidBody inside a Union, and the `<rigid_body>` can be directly mapped to a RigidBody.

pe however allows a Union to be wrapped in another Union and construct a hierarchical structure. In COLLADA, where each `<shape>` must be a geometric primitive or a mesh, this can not be represented. (Note: While `<physics_model>`s allow a recursive structure, they do not impose any behavioural constraints on their components, unlike a composite body.) We choose to extend COLLADA by introducing `<union>` elements to the custom `<technique>` profile, grouping the `<shape>`s appropriately.

```
<rigid_body>
  <technique profile="pe">
    <union>
      <shape>
        <!-- ... -->
      </shape>
      <union>
        <shape>
          <!-- ... -->
        </shape>
        <shape>
          <!-- ... -->
        </shape>
      <union>
    </union>
  </technique>
</rigid_body>
```

Listing 4: Custom `<union>` scheme

`<union>`s can be translated and rotated as a whole, which affects all subordinate elements. Beyond that, they only specify the UserID. This is analogous to the features of **pe** Union s, which will facilitate the conversion process.

Parallel to the custom profile, we are obligated by the standard to also include a representation in the common profile. The most faithful variant is to flatten the hierarchy and include all primitives directly under the `<rigid_body>`. It is to be noted that we have no guarantee that eventual readers would be able to model multi-level composite bodies, while another **pe** application will use the custom profile.

As a final note, we will also include the UserID tags of RigidBody s in the custom data scheme.

4.4 Materials

Both COLLADA and **pe** use a material library that bodies refer to, though COLLADA also allows for materials to be defined locally for a single body, unable to be reused by other bodies. From the point of view of **pe**, this restriction serves to prevent naming conflicts, but is otherwise artificial. We can extend the scope of the materials and make the names unique again by appending a numerical index.

Material definitions in COLLADA have only values for friction and restitution coefficients, compared to **pe** Materials, which specify density, Poisson's ratio, Young's modulus, stiffness and damping coefficients as well. In COLLADA, most of these values are simply nonexistent, while density is removed from the concept of a material and is instead an intrinsic property of `<rigid_body>`s. This poses problems for an import into **pe**. Without any further information about the missing values, we can only assume standard values. Also, bodies can consist of the same material, yet have varying densities. Barring fundamental changes in **pe**, the only solution is to make a duplicate of the material with the new density each time – potentially one for every rigid body. However, **pe** is optimized for small numbers of materials, and numerous varying densities in the COLLADA file are able to cripple it.
It is even possible in COLLADA to define a body without a material at all and just the density data. Again, the only remedy is to use standard values.

When we export from **pe** to COLLADA on the other hand, we would lose the values unrecognized by COLLADA. Therefore we choose to extend COLLADA and introduce a custom `<technique>` profile to `<physics_material>`s containing these values. If the file is reimported by a **pe** program, these values can be accessed and used to faithfully recreate the materials. It might also be suitable to include the density of the material there, imitating the **pe** concept of materials, which simplifies the parsing process in these cases.

```
<physics_material id="Material_06" name="Mustard">
  <technique_common>
    <dynamic_friction>0.1</dynamic_friction>
    <restitution>0.4</restitution>
    <static_friction>0.5</static_friction>
  </technique_common>
  <extra>
    <technique profile="pe">
      <density>1.2</density>
      <poisson_ratio>0.2</poisson_ratio>
      <young_modulus>1</young_modulus>
      <stiffness>8</stiffness>
      <damping_normal>0</damping_normal>
      <damping_tangential>0</damping_tangential>
    </technique>
  </extra>
</physics_material>
```

Listing 5: Extended `<physics_material>` scheme

4.5 Constraints

Constraints (called Joints in **pe**) always link exactly two bodies. **pe** distinguishes between a number of types of Joints, some of whom are only partially implemented at the current time. COLLADA is more data-driven here, as there is only one type of `<rigid_constraint>`, specifying the limits of rotation and translation it allows, and its spring strength. With these values, it is possible to emulate any of the **pe** Joint types, and even others. The FixedJoint type corresponds to all-zero limits.

On the other hand, this means that it is possible to encounter a joint type in COLLADA which can not be faithfully modeled in **pe**. Since currently only the FixedJoint type, which corresponds to only a single specific set of limits, is implemented, this can even be considered the general case.

5 Implementation

5.1 Choice of an XML parser

COLLADA is an XML-based format. To avoid reinventing the wheel, the implementation uses an imported XML parser library.
Sony Computer Entertainment America created and made open source a DOM XML parser geared specifically for COLLADA, called COLLADA DOM. [9] [10] [11] It provides classes for every type of element in COLLADA and utility functions for URL resolution and data access.
The other parser taken into consideration is written by Dr. Frank Berghen [12]. It is also based on a DOM architecture, and uses a rather simple design with a single class for any elements. It only provides the basic tree navigation and data-attribute access functions, though it is more intuitive to use.

Preliminary trials showed a sharp decrease of the COLLADA-DOM parser's performance with a rising number of bodies. On the other hand, Dr. Berghen's parser scaled better and also fell closer to the desired performance range.

We decided to use the Dr. Berghen's parser. Due to its small size, consisting only of the single class XMLNode, we included it in the module instead of linking it as an external library. For frequent tasks, we also implemented a number of utility functions.

5.2 Data structure

Using these tools, we designed the following classes (see Figure 1 for a class diagram):

ColladaFile Container for a specific COLLADA file. Contains the parser-provided object for the root element and, for ease of access, the relevant `<library_*>` elements. Resolves URLs in this file. Many of the other classes anchor themselves to a ColladaFile object.

ColladaFileHandler Creates ColladaFile objects both from scratch or from a file, and also writes them to file. Provides functionality to manipulate the contained ColladaScene s.

ColladaScene Represents a `<physics_scene>` and `<instance_physics_scene>` pair, and lists the contained ColladaModel s.

ColladaModel Represents a `<physics_model>` and `<instance_physics_model>` pair, and provides the core functions to load and store bodies and constraints.

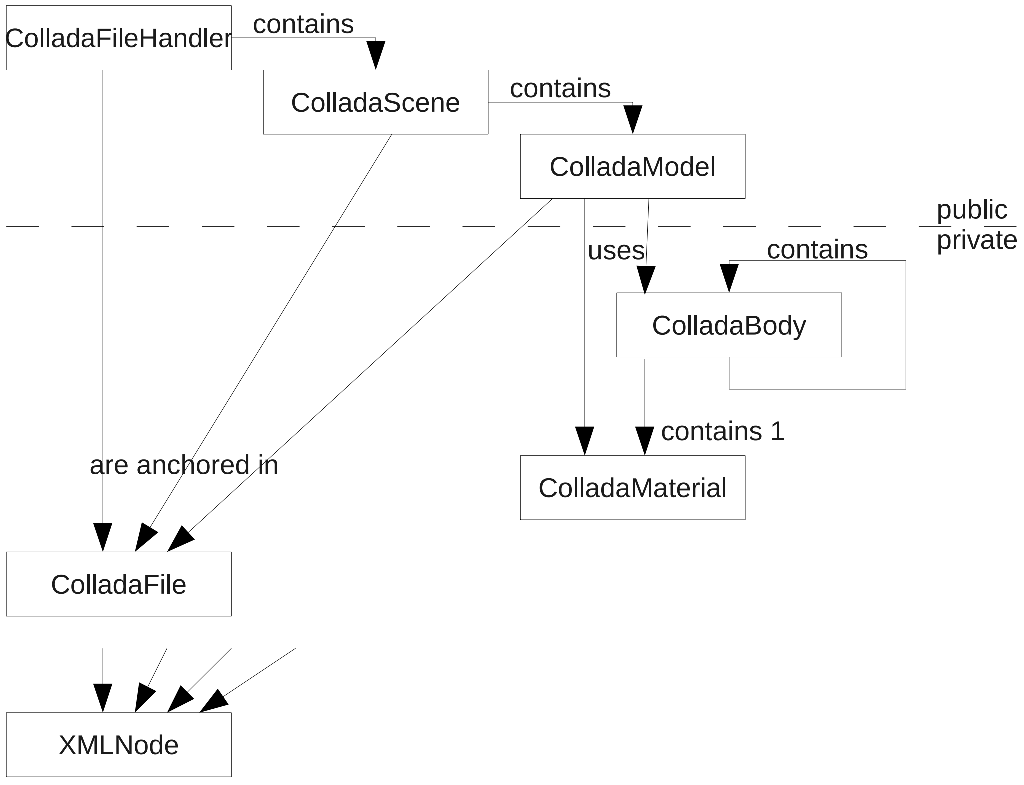


Figure 1: Classes used by the 'colladaphy' module

ColladaBody Contains the necessary values and properties to characterize a RigidBody. Is used by ColladaModel to temporarily store body data and modify it before creating the actual RigidBody objects.

ColladaMaterial Contains the necessary values and properties to characterize a Material. Provides functions to store and load materials in the `<library_physics_materials>`. Allows ColladaModel and ColladaBody to modify the values before creating the actual Material s.

Among these classes, ColladaFileHandler, ColladaScene, (and in some applications ColladaModel) are the only ones accessed by the user.

All these classes and utility code use the XMLNode class provided by the parser. They are part of the regular 'pe' namespace, and the header files are grouped into a 'colladaphy' directory.

5.3 Program flow – Read from file

The ColladaFileHandler class is used to open the file and allows the user to select from a list of ColladaScene objects corresponding to `<physics_scene>`s. In turn, the process can as well select specific ColladaModel objects part of the scene. This second step serves to allow distributing the content over several nodes in parallel simulations. Giving even finer control and enabling to import single bodies or constraints, while theoretically useful, is an unlikely scenario and entails complications in case the constraint links bodies which are not yet present.

Thus, from either all models in the scene or from the selected ones, all bodies and constraints are parsed and created in the World. Subordinate models are loaded as well, the method calling itself recursively.

The intuitive approach to loading bodies would be to iterate over all `<rigid_body>` elements and use the contained data to create the corresponding **pe** object.

This approach has two problems:

1. `<instance_rigid_body>` elements can modify the data and **pe** objects allow only choice properties to be modified after construction.

2. When we import the constraints, we need to be able to identify the created **pe** RigidBody object from the COLLADA URL.

Thus, we use the ColladaBody class, which records all relevant parameters of a body, but in contrast to the RigidBody, is still fully modifyable.
Member variables of ColladaBody:

- Type of geometry
- Basic geometry parameters (radius, height, or sidelength)
- Other simple parameters and values (ID, position, mass and/or density, dynamic/static)
- Material reference

- List of ColladaBody type component shapes (for compound bodies)
- BodyID pointer to the resulting RigidBody object
- Instantiation counter

We first parse the contents of each `<rigid_body>` element and create a ColladaBody object, which is then stored in a associative list using the COLLADA identifier as an index. If the type of geometry is foreign to **pe**, an error message is logged and we continue to the next body. For compound shapes, every `<shape>` component is parsed on its own and represented by a ColladaBody, which are collectively contained in the main ColladaBody object. If we encounter a **pe**-customized structure with `<union>` elements (see 4.3), this can be done recursively. Simple values can be stored both in the primary as well as the component ColladaBody s.

In a second step, we parse the `<instance_rigid_body>` elements. We follow the 'target' URL to the the associated `<node>` element and parse the transformation data determining the body's position in the world frame. However, instead of resolving the 'body' URL inside the document, we retrieve the appropriate body from the list, make a copy and apply any new values given by the instantiation. In compound bodies, some values need to be redistributed or balanced between the main and component bodies (e.g. material, and in some cases mass). After the body is thus finalized, we create the RigidBody object. The pointer to the newly-created body is remembered by the original ColladaBody.

When we later load the constraints, we use these deposited pointers to identify the RigidBody s the Joint will be applied to. The ColladaBody class includes a counter for the number of RigidBody s that are derived from it by eventual multiple instantiation. This counter guards against abnormal cases when a `<rigid_constraint>` attaches itself to such a multi-incarnated body and throws a warning message.
Unfortunately, due to the incomplete implementation of Joint s the only parameters of the `<rigid_constraint>` we are able to respect are the anchor points on the attached body.

The above description omits the matter of materials. Materials are defined by a `<physics_material>` element, which can be located either locally as a child attribute of the body, or in a library with a `<instance_physics_material>` referencing it in the body. (See also 4.4)

Here a naïve approach, where each mention of a material is resolved and parsed as soon as it is encountered, is possible. We have to take into consideration that the body might specify its own, conflicting density, in which case we need to modify the material before finalizing it in **pe**. Alternatively, we can pre-load the material library, which removes the need to navigate the XML structure to resolve the URL. Since we need to still be able to modify the material, we include a temporary storage class ColladaMaterial, analogous to the ColladaBody type. These ColladaMaterial objects are stored in a static library container and referred to by the ColladaBody objects. If the material data values are modified, we duplicate the ColladaMaterial, allowing reuse of the unmodified original as well as of the modified material. Only when we finalize the body we also convert them to a true Material.

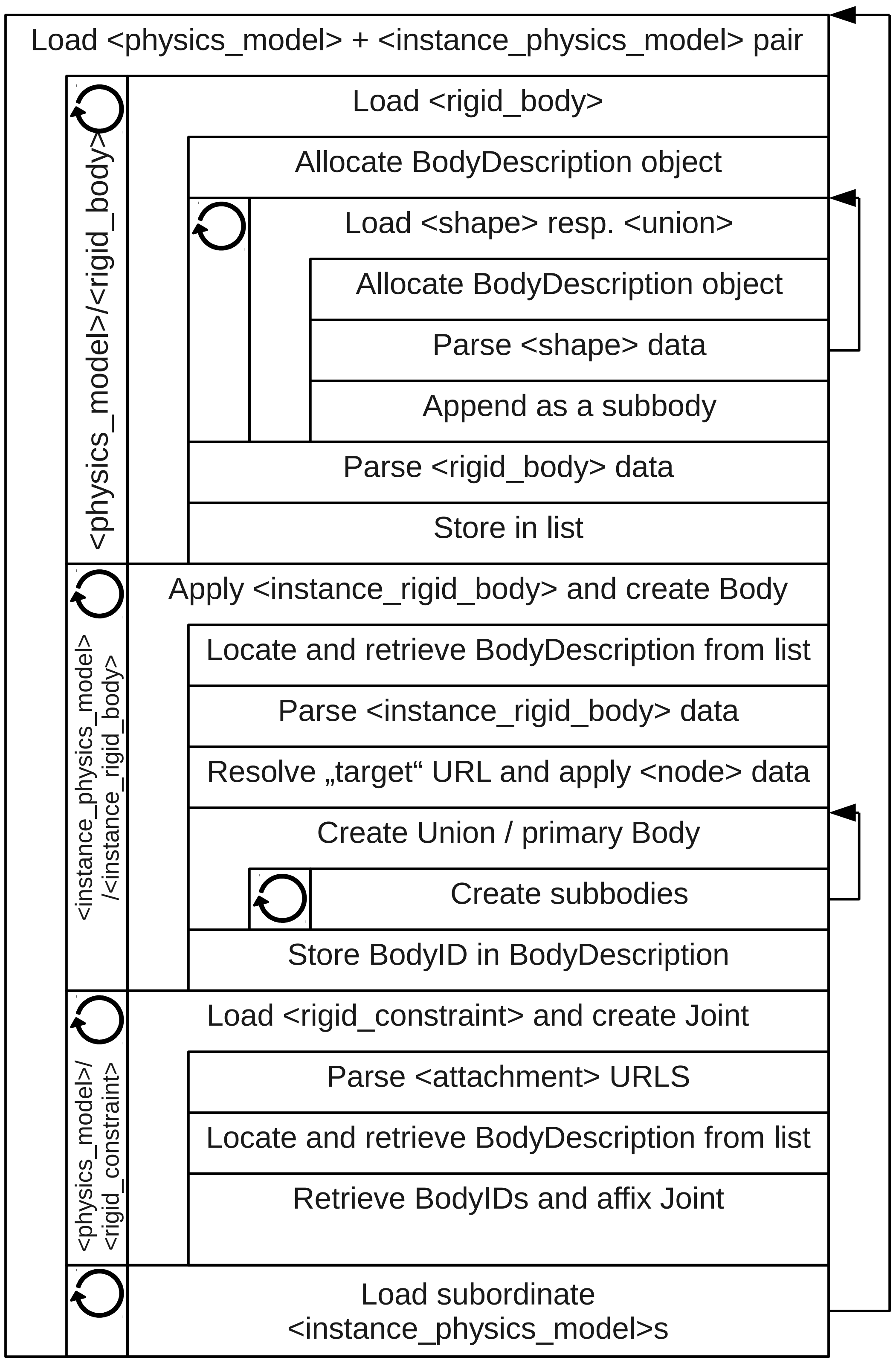

Load <physics_model> + <instance_physics_model> pair
<physics_model>/<rigid_body>
Load <rigid_body>
Allocate BodyDescription object
Load <shape> resp. <union>
Allocate BodyDescription object
Parse <shape> data
Append as a subbody
Parse <rigid_body> data
Store in list
<instance_physics_model> /<instance_rigid_body>
Apply <instance_rigid_body> and create Body
Locate and retrieve BodyDescription from list
Parse <instance_rigid_body> data
Resolve „target“ URL and apply <node> data
Create Union / primary Body
Create subbodies
Store BodyID in BodyDescription
<physics_model>/ <rigid_constraint>
Load <rigid_constraint> and create Joint
Parse <attachment> URLS
Locate and retrieve BodyDescription from list
Retrieve BodyIDs and affix Joint
Load subordinate <instance_physics_model>s

Figure 2: Flow diagram of the read process

5.4 Program flow – Write to file

Writing a COLLADA file is easier than reading one, since we operate from full knowledge and we can choose our style in how to name and group the elements, while the import routine has to be able to read all possible styles.

When a file is created by the ColladaFileHandler, the libraries are set up immediately. Further methods add `<physics_scene>` and `<instance_physics_scene>` elements at the appropriate places and create a ColladaScene object. In a very similar way, the ColladaScene establishes a ColladaModel.

To insert a body in the model, we first append a `<rigid_body>` element to the `<physics_model>`. If the body is a primitive, `<rigid_body>` contains one shape, if it is a Union, several. Each such `<shape>` element contains its basic geometric information, either its mass or its density dependent on configuration, and a `<instance_physics_material>` reference to the constituent material (see below). If several `<shape>`s are present, they include translation and rotation data relative to the center of mass and orientation of the entirety of the body.
Most of the data that `<rigid_body>` can contain has already been covered in the `<shape>` definition. We create a `<dynamic>` element, and we may choose to configure the code to repeat some information. While this is redundant by the standard, it is conceivable that some software may depend on these elements.
The `<rigid_body>` element is given a 'sid' attribute numbered by the **pe** SystemID, which is already guaranteed to be unique. The corresponding URL consists of the `<physics_model>`'s id and the sid, separated by a slash and is deposited as a 'body' attribute to an `<instance_rigid_body>` element, which is added under `<instance_physics_model>`. Conforming to the COLLADA standard, here we specify the velocity data, which is not allowed in the library definition.

Furthermore, a `<node>` element is added to the appropriate library, analogously named and referred to by the 'target' attribute of `<instance_rigid_body>`. Here we specify position and orientation of the body. While this information could have been stored elsewhere, the `<node>` element is common to both the physical and visual scenes. While we do not include visual objects, a graphically-oriented COLLADA reader can still use the `<node>` elements to recreate the spatial arrangement of the bodies. This style was also inspired by the way the Bullet Collada Converter stores this data.

Using the consistent naming scheme, it is also rather easy to create URLs for `<rigid_constraint>`s, assuming that the bodies they are affixed to have already been inserted. As has been stated, the only possible type of constraint that **pe** is currently able to model is that of the FixedJoint, which corresponds to the default values of the `<rigid_constraint>`'s properties. We thus can consider our work done, or optionally include the property elements anyway with the appropriate values. This also prepares for the future, when the **pe** will be able to model more diverse types of constraints.

We chose to not define materials locally inside `<rigid_body>`s but to store them all in the `<library_physics_materials>`. The **pe** MaterialID number serves as id, with the name included as the optional name attribute to the XML element. When a body is stored, the routine checks whether a `<physics_material>` element is present, and if not, inserts it into the library. This has the effect that only the materials which are actually used are stored,

reducing the size of the resulting file.
The content of the `<physics_material>` element observes the scheme designed in 4.4, with the properties prescribed by COLLADA in the `<technique_common>` profile and all further values in the `<extra>` section.

5.5 Performance optimization

Whenever we have to resolve an URI-type URL, the type of element where we encounter the URL gives information as to where the target element is likely to be. For example, the element referenced by an `<instance_physics_material>` is of the `<physics_material>` type and thus most likely located in the `<library_physics_materials>`. So instead of traversing the entire XML document, we first limit our search to the appropriate region. Only if this first search fails to turn up a result (while for some kinds of elements this would already indicate a faulty document, others are allowed more liberty in placement), we proceed to search the entire document. This works in a similar way for SIDref-type URLs prefixed by an id.

Since we decided to pre-load the resource `<rigid_body>` objects into a list, we can forgo resolving these instantiation URLs and navigating the XML structure entirely. We instead use the sid of the body as an identifier in the list and compare the last segment of the SIDref against it to retrieve the right element from the list. If multiple `<physics_model>`s are used and the sids are no longer guaranteed to be unique, the model id can prepended as well (which implicitly replicates the URL). In case this shortcut fails for whatever reason, (e.g. because the content is distributed over several files) the option to resolve the URL is still open to us. We could also have eased the URL resolution process by indexing the XML document, though making such a list sidesteps this issue.

We consider it likely that in cases where a body specifies a variant density to a material, other bodies that also respecify their density do so to the same value. Comparing the material values to the existing ones and thus detecting duplicates allows us to reuse and thus minimize the amount of Material s. The **pe** scales badly for large numbers of Material s, as demonstrated by Figure 10.
Though if there are many materials documented in the COLLADA file which are actually distinct, we have no other choice than to introduce these materials. Even if this bogs down the **pe** engine, subsuming data-distinct materials to reduce their number is not a decision the import module can make.

5.6 Peculiarities of the Bullet COLLADA Converter

We observed that the .dae files created by the Bullet COLLADA Converter [7] do not fit the standard in several regards.

1. SIDref-type URLs consist of nothing but the sid of the target `<rigid_body>` element, without any lead-up or identification of the `<physics_model>` it resides in.
2. The parameters of the `<plane>` equation are interpreted as $Ax + By + Cz = D$ instead of $Ax + By + Cz + D = 0$; effectively, this toggles the sign of the displacement.
3. `<rigid_constraint>` linear limits are relative not to the current distance, but to the position of the reference body.

This also affects the reading process: bodies which are referenced by SID paths are not found, planes reflect around the origin and (0,0,0)-constraints make the bodies snap together.
To accommodate this, we introduce a Bullet compatibility mode to the **pe** colladaphy module which takes these subtle differences into account.

6 Validation

To verify that the COLLADA files are written and read correctly, the following verifications were employed:

- Write content to a COLLADA file and read it back into a new world; assert that the content is unchanged. (6.1)
- Write content to a COLLADA file and have it read by the Bullet COLLADA Converter; assert that the content is unchanged. (6.2.1)
    - Subsequently modify the content in Bullet, then restore it to a COLLADA file and read it with **pe**; assert that the bodies behave as expected
- Read a COLLADA file by an external source and assert that the content is accurate. (6.2.2)

As an external program, we employed the COLLADA Demo of the Bullet Physics Library.

To analyze the contents of a **pe** world, the following avenues were used:

- Debug output of bodies, containing all relevant properties
- Content of the logfile, which contains entries from important steps during the reading/writing process
- Rendering of the world using the povray module

6.1 Self-consistency validation

One of the goals of the COLLADA extension of **pe** is to allow checkpointing, which has the essential requirement that the data is not changed in any way. To test the 'colladaphy' module's suitability for this task, preexisting **pe** worlds were exported into a COLLADA file and then imported into a new world, simulating a checkpoint and restore. If the process was successful, these worlds would be identical, which we determined by comparing the debug output. We used the following contents as test cases:

1. An open box consisting of five boxes, with a dozen spheres pairwise constrained by fixed joints (Figure 3)
2. A large number of bodies of all kinds
3. Unions of several capsules, forming rings
4. Boxes with successive rotations

5. Custom materials
6. A union which contains another union

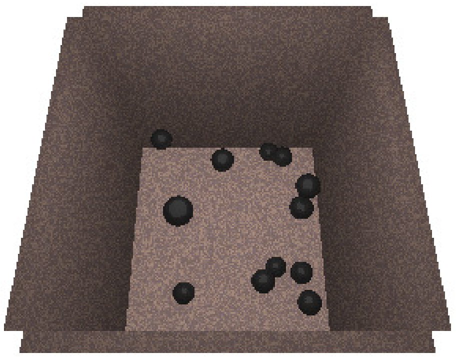


Figure 3: Test case 1, Joints not rendered

The tests were mostly successful; we observed the following problems and discrepancies:

1. Numerical data was distorted by rounding errors, most often in the rotational data.
2. The SystemIDs were not guaranteed to be conserved. It was however assured that bodies are created in the same order they were written to file earlier.

Discussion:

1 is due to the conversion of binary floating point numbers to an ASCII representation of lesser precision, as well as the conversion between the quaternion and axis-angle formats for rotational data. This is to some degree unavoidable, but the precision of the ASCII floating point numbers can be artificially high, keeping the error artificially low.

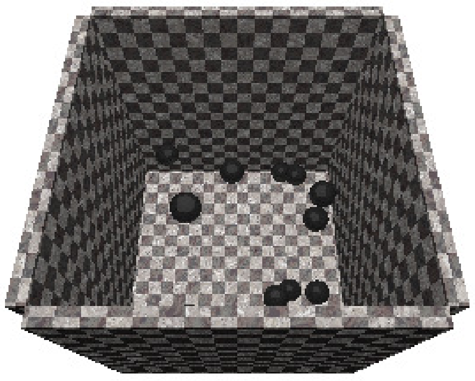
2 In many practical cases, the bodies are written to file in the order of their SystemIDs. Thus we can guarantee the conservation of the relative order of these. We can extend this guarantee to all cases by explicitly ordering the `<instance_rigid_body>` elements by the SystemIDs when we write to file. It is to be noted that a completely faithful reproduction of the SystemIDs may be impossible if there are preexisting bodies in the world when the COLLADA file is imported.

6.2 Validation of interaction with the Bullet engine

6.2.1 Loading a pe-created file with Bullet

We were not able to use the debug output for direct comparisons with content in Bullet. Instead, we relied on the log output of both **pe** and Bullet as well as renderings to assert that the content was faithfully transferred. We moved and added bodies with the interactive tool of the Bullet application as an additional measure to assert that the bodies behave correctly and that constraints and centers of mass had been correctly transferred.

We used the same test cases as in 6.1 for these tests with the exception of 5 and 6, as they rely on custom profiles and as such would not be interpretable by Bullet.


(a) POVray rendering of the **pe** data

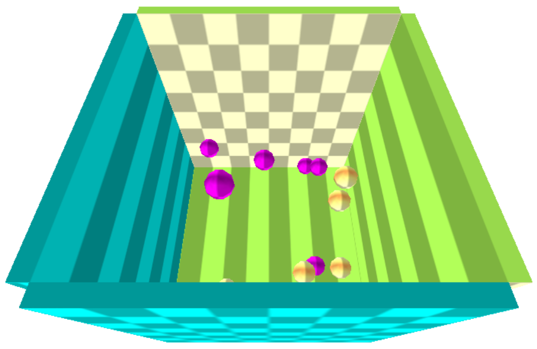
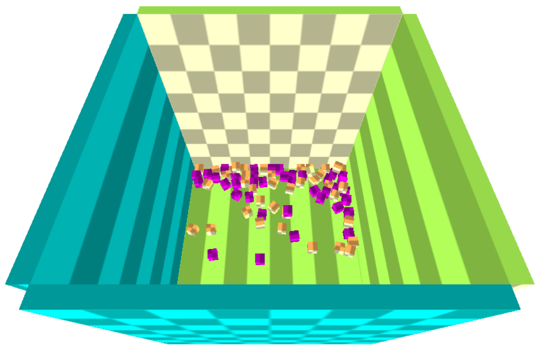

(b) Bullet display of content

Figure 4: Test case 1


(a) Bullet display of content

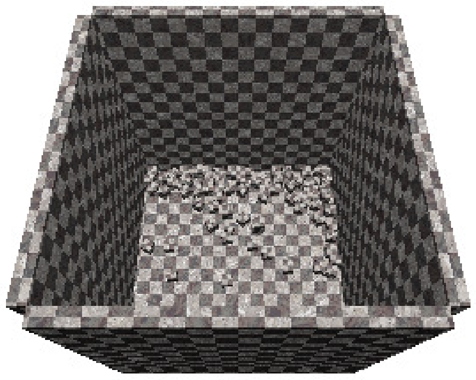

(b) POVray rendering of the **pe** data

Figure 5: Modified test case 1

Observations:

1. The Bullet COLLADA Converter keeps the raw XML data of the read file and modifies it with the new data instead of creating the new file from scratch.
2. In the context of 1, new content is not necessarily added to the present structures. In the encountered examples, newly added bodies were contained in a separate model, and new materials were located in a second `<library_physics_materials>`.
3. Cylinder and capsule bodies appear with double length in Bullet compared to **pe**.
4. Numerical data was distorted by rounding errors

Discussion:

1 incidentially ensures that included **pe**-specific data that Bullet can not interpret, like UserIDs and the additional material values, is conserved for an eventual reimport. On the downside, any changes the Bullet-based program makes are imparted only on the common profile, but not the custom profile. If these files are reimported into **pe**, the custom profile is usually given preferential treatment by **pe**, even though it still holds the old content in this case.
Affected is content that is part of the custom profiles parallel to the common scope, i.e. body parametrization. Other data, like body position and velocities, lie outside the custom profiles and thus won't be affected.
This warrants a careful, application-dependent configuration of the module with regard to whether to use the custom profile.
The discrepancies we observed in 6.1 may also persist.

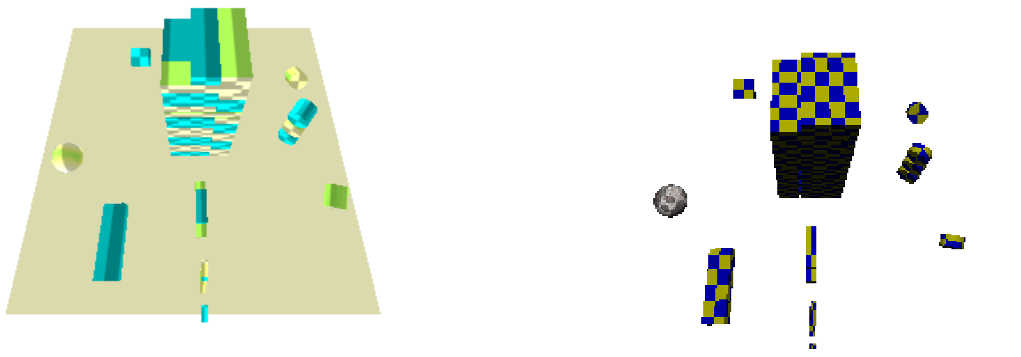
2 is not a problem as long as the code takes proper care to parse all `<library_*>` elements beyond the expected one.

3 The raw XML data supports the dimensions shown by **pe**. Code analysis of the Bullet COLLADA Converter [7] showed that this is a mistake on its part, failing to convert between the entire length given by COLLADA and the half-extents Bullet uses internally.

4 The conclusions made in 6.1 apply.

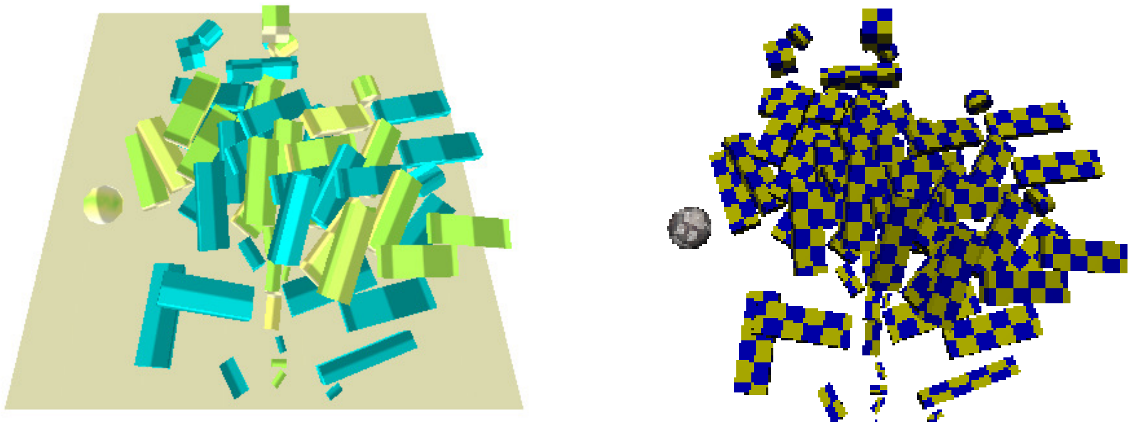
6.2.2 Loading a Bullet-created file with pe

The Bullet COLLADA Demo provided some example files, which we loaded into **pe**.
Testing the import of constraints from external sources would be futile (see 4.5), so the test cases did not contain any.



(a) Bullet display of content (b) POVray rendering of the **pe** data

Figure 6: jenga.dae example file



(a) Bullet display of content (b) POVray rendering of the **pe** data

Figure 7: jenga.dae example file, modified

Observations:

1. Bodies which consist of a mesh geometry are not loaded in **pe**.
2. Bodies whose dimensions include a zero (e.g. the two-dimensional flat finite plane in the example above) are not loadad in **pe**.
3. Cylinder and capsule bodies appear with double length in Bullet compared to **pe**.

Discussion:

1 was expected during the design phase. Even beyond the import process, **pe** is currently unable to handle mesh geometries.

2 is a restriction imposed by **pe**, not allowing bodies with a volume of zero. A possible remedy would be to instead use the smallest nonzero value, though this can have unpredictable consequences to the physics.

3 See 6.2.1 above

7 Scalability

Figure 8 and 9 show the runtime of the procedure to store and reload respectively a rising number of unmoving spheres randomly distributed in the world.
We observe that the runtime scales with the number of bodies, linearly for the store procedure and superlinearly for the load procedure.
The superlinear scaling can be explained by the effort it takes to resolve the URLs to the `<node>` elements for every body, which scales linearly by itself with the number of `<node>`s.

For Figure 10, the material duplicate detection (see 5.5) has been deactivated, so that a new material has been created for every body. The results demonstrate how important it is to limit the amount of materials registered in **pe**.

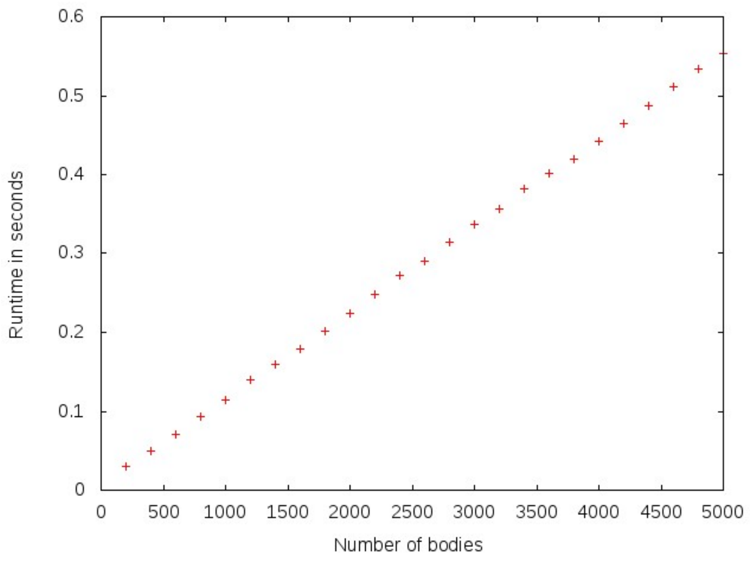


Figure 8: Runtime of the store procedure

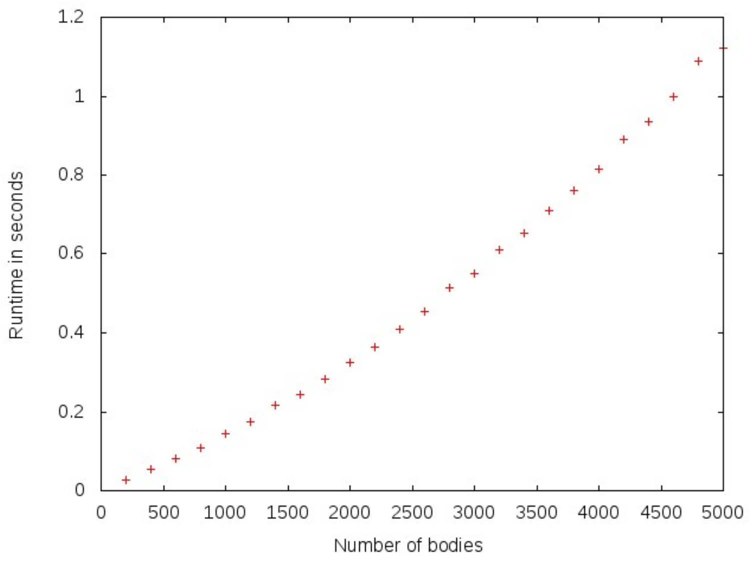


Figure 9: Runtime of the load procedure

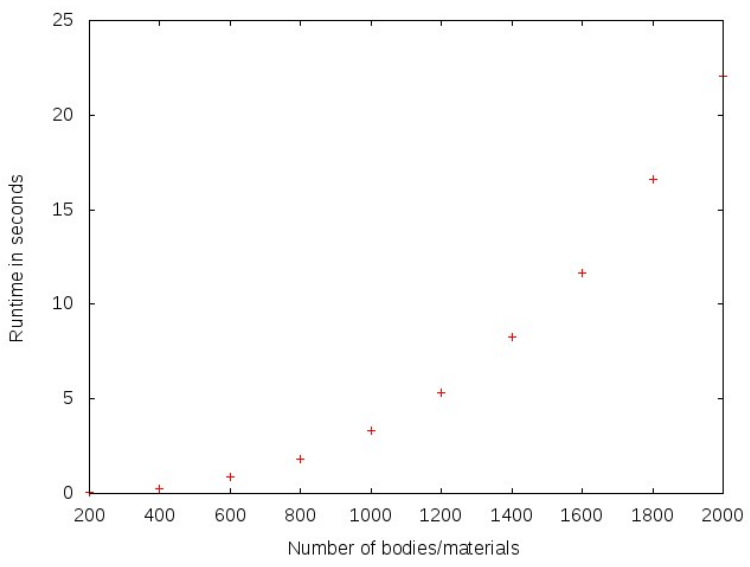


Figure 10: Runtime of the load procedure, with 1 material for every body

8 Conclusion

8.1 Suitability for Checkpointing

The runtime of the store process scales linearly with the number of bodies. For the expected scale of a typical **pe** multi-processor simulation, the runtime is well under 1 second. The inaccuracies are under control or are irrelevant to the simulation.

It has to be said that there are some disadvantages to COLLADA as a storage format for **pe**. COLLADA Physics considers several properties of objects optional or even lacks provisions for them completely. We had to add extensions for these dates.
Also, the concept of instantiation of resources is unsuited to the **pe** data structure. Actually exploiting the extra capabilities of the instantiation scheme represents an additional effort, and as such the reading code has to collect the data for one body from at least three places in the file, which is inherently inefficient.
The files being human-readable is convenient, but for a strict checkpointing application unnecessary, while the conversion of data between ASCII-encoded and binary formats adds to the runtime.
Nevertheless, while an original binary data storage may theoretically outperform it, storage using COLLADA Physics qualifies as sufficiently fast. We conclude that the module is suitable for checkpointing a simulation.

8.2 Suitability for data exchange

With some limitations, we were able to successfully exchange data with the Bullet engine.

However, the prognosis for other software is doubtful. The **pe** is a work in progress and still unable to model certain characteristics, most notably mesh shapes and arbitrary constraints. Attempts to load such data are doomed to failure.
On the other hand, several aspects of the COLLADA Physics standard are of an abstract,

conceptual kind and deliberately open to interpretation. As the Bullet COLLADA Converter demonstrated, this leeway is exercised and the data schemes can be and are interpreted differently by each application. Even establishing limited compatibility with this single sample required numerous custom adaptions of the code to accommodate its peculiarities. Extrapolating from this experience, it is not realistic to expect the colladaphy module to be able to read any COLLADA documents from an unknown source right away. Most basic concepts are indeed common, but full functionality is likely to require a case-by-case configuration or even extension of the code to accommodate newly-encountered peculiarities.

Only with these caveats we can consider the module suitable for exchanging data with any third-party software. If a way to reliably exchange data using the full scope with a specific application is sought, we recommend to add a further extension to **pe** handling the associated format. In the absence of such a thing, the COLLADA Physics module remains a useful addition to **pe** due to COLLADA's widespread adoption.

8.3 Summary

The COLLADA Physics extension to **pe** is functional and fully adequate for the purpose of checkpointing a long-term simulation as well as, to a lesser degree, for the exchange of content with third-party software.

References

[1] pe Rigid Body Physics Engine
Chair for System Simulation, Universität Erlangen-Nürnberg
http://www10.informatik.uni-erlangen.de/Research/Projects/pe/

[2] Khronos Group
http://www.khronos.org

[3] Collada 1.5.0 Specification, Khronos
http://www.khronos.org/collada

[4] Bullet Physics
http://bulletphysics.org/wordpress/

[5] Physics Abstraction Layer
http://www.adrianboeing.com/pal/index.html
http://www.adrianboeing.com/pal/benchmark.html#COLLADA

[6] NVIDIA PhysX
http://developer.nvidia.com/technologies/physx
http://developer.nvidia.com/physx-partners

[7] dynamica Plugin for Bullet Rigid Body Dynamics
http://code.google.com/p/dynamica/downloads/list

[8] Fabian Schönfeld
ped - A Physics Engine Editor
Diplomarbeit, Universität Erlangen-Nürnberg, 2009

[9] COLLADA Document Object Model
http://collada.org/mediawiki/index.php/COLLADA_DOM

[10] COLLADA Document Object Model
sourceforge.net/projects/collada-dom/

[11] COLLADA-DOM Portal
http://collada.org/mediawiki/index.php/Portal:COLLADA_DOM

[12] Dr. Ir. Frank Vanden Berghen
C++ XML Parser
IRIDIA, Université libre de Bruxelles
http://www.applied-mathematics.net/tools/xmlParser.html

[13] Klaus Iglberger, Ulrich Rüde
Large-scale rigid body simulations
Multibody System Dynamics 25(1), 2010

[14] E. Coumans, K. Victor
COLLADA Physics
Proceedings of the 12th International Conference on 3D Web Technology, 2007

(All web addresses accessed up to at least March 2012)