Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS (2013)
J. Bender, J. Dequidt, C. Duriez, and G. Zachmann (Editors)

Initial Steps for the Coupling of JavaScript Physics Engines with X3DOM

L. Huber[1]

[1]Fraunhofer IGD, Germany

Abstract

During the past years, first physics engines based on JavaScript have been developed for web applications. These are capable of displaying virtual scenes much more realistically. Thus, new application areas can be opened up, particularly with regard to the coupling of X3DOM-based 3D models. The advantage is that web-based applications are easily accessible to all users. Furthermore, such engines allow popularizing and presenting simulation results without having to compile large simulation software.
This paper provides an overview and a comparison of existing JavaScript physics engines. It also introduces a guideline for the derivation of a physical model based on a 3D model in X3DOM. The aim of using JavaScript physics engines is not only to virtually visualize designed products but to simulate them as well. The user is able to check and test an individual product virtually and interactively in a browser according to physically correct behavior regarding gravity, friction or collision. It can be used for verification in the design phase or web-based training purposes.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Physically based modeling I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual reality I.6.4 [Computer Graphics]: Model Validation and Analysis—

1. Introduction

Physics engines are systems whose importance has increased within the past twenty years. They are already inherent parts in the development of computer games, in the production of films and of complex software systems in the areas of entertainment, robotics, digital prototyping or training simulators. They are able to fill virtual scenes with life and give them the appropriate physical behavior. [BKJC12]

The motion of bodies is approximated according to basic concepts of classical mechanics. Depending on the properties of bodies there are three paradigms: rigid body dynamics, soft dynamics and fluid dynamics. Each paradigm uses different methods for describing motion. Rigid body dynamics are used for solids. Solids are bodies which are not deformable. In this connection, collision detection plays an important role since it prevents rigid bodies from penetrating. Soft dynamics is concerned with the motion of deformable bodies. Classic examples are clothes or substances in general or things consisting of deformable matter such as rubber. The third paradigm, fluid dynamics, is used if motions of water or particle systems, e.g. the visualization of gas dispersion, are to be described. [Jon11, Diz12]

This paper, however, focuses on the first paradigm. So that in the following, physics engines for rigid body simulations shall be discussed.

A rigid body simulation is a timed continuous simulation for the motion of rigid bodies which uses differential equations in order to describe physical laws [mac13]. It is executed cyclically and in several phases.

One important requirement for physical simulations is the definition of a physical model consisting of several rigid bodies. A rigid body is ideally defined as a solid object, meaning that the distance between each pair of points of the body never changes even if huge forces are applied. Each rigid body has a mass which is equally distributed throughout the entire volume and a center of mass. Rigid bodies are divided into moveable and non-moveable bodies. Bodies with a mass of zero are static. Dynamic bodies have a positive mass. Besides, physics engines provide different shape

types for rigid bodies which range from geometric primitives (e.g. box, sphere, cylinder) to complex 3D objects. Generally, rigid bodies have six degrees of freedom and can be moved and rotated in all three dimensions (Figure 1). [BKJC12]

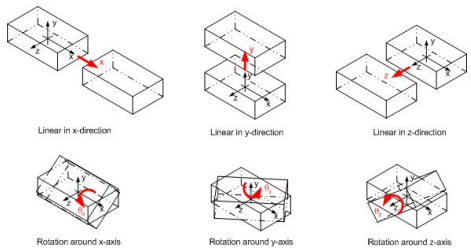

Figure 1: *The six degrees of freedom of rigid bodies (Source: http://basieandnewton.blogspot.de/).*

Several rigid bodies that are connected by constraints are called multi-body systems. Depending on the usage, different types of constraints are available (e.g. PointToPoint, MaxDistance, Hinge, Degree6Dof). In complex models (e.g. control of equipment and machines, animation of humans) that consist of a greater number of parts, constraints play a particularly important role. For each constraint two rigid bodies are connected with each other. Constraints limit the motion of bodies and specify what happens if these limits are reached. [Jun11] Section 4.2.2 examines the functionality of constraints in detail.

Figure 2 illustrates the phases of a simulation.

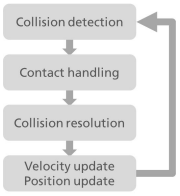

Figure 2: *Phases of a rigid body simulation [BKJC12].*

The simulation starts with a collision detection. This phase is to check if there is a contact between two rigid bodies. If so, the contact points have to be determined.

Since collision detection has to be virtually tested for all pairs of bodies, it may lead to performance bottlenecks especially if more bodies are tested at once. Thus, the phase is divided into a broad phase and a narrow phase. During the broad phase, the bodies are enveloped by the smallest enveloping body that completely contains them. Enveloping bodies are usually simple geometric primitives, e.g. spheres. Only in case of the enveloping bodies overlapping, the high-performance narrow phase has to be started in order to test the two rigid bodies for collision.

The following contact handling phase uses the calculated contact points to derive the appropriate contact forces that cause a repulsion of the bodies as a result of the collision. This effect leads to an immediate change of the velocity and motion of the bodies. These values are calculated during the collision resolution. Before repeating the simulation run in the last phase, the position and velocity of the bodies are updated. [BKJC12]

The physical bases of the simulation are Newton's laws and other concepts of classical mechanics. For more information please refer to [BKJC12] and [Jun11].

2. Related Work

In the field of entertainment, physics engines play an important role especially in the development of computer games and movies. In game engines, it was initially not important to simulate physically correct behavior. Plausible behavior was perfectly adequate. Instead, game engines had to be real-time capable to respond immediately to user interaction. [BKJC12] With the increase of CPU performance available and the ability to run simulations on the GPU, the desire to develop more realistic simulations with higher quality that support real-time capability rose. In this context, [Cou13] introduces an approach to the GPU rigid body simulation. Furthermore, [Pla12] discusses a sampler-based motion-planning approach by the use of physics engines in order to make the motion of virtual agents with regard to collision detection much more efficient. The popular game Angry Birds for example was developed by applying the 2D physics engine Box2D [Box13]. Games like Toy Story or movies such as 2012, Hancock and Sherlock Holmes were implemented by using the popular open source 3D physics engine Bullet Physics [Cou12].

In the field of digital prototyping, physics engines are used for the verification of virtual products in the design phase to check the functionality of products before they go into development. This mainly counts for new developments and intends to reduce development costs, save development time and improve the quality of products. Since they represent an important base for the development of real products, physics engines are required to meet extremely high needs of simulation accuracy and results. [BKJC12] Webots and Microsoft Robotics Developer, for instance, are development environments to model, program and simulate mobile robots [CYB13,MIC13]. The physics engine PhysX was used by the European Space Agency to perform the verification of the Mars rover sample for the ExoMars program to study the Martian environment [KJ11]. Furthermore, the company machineering GmbH & Co. KG uses physics engines to visualize the flow of materials and the behavior of realistic systems in real time [mac13].

In the area of industrial and training simulators, physics engines are used to train the operation of vehicles, machines

or equipment. Computer simulations are much cheaper and less dangerous. Trainings on real objects or in real situations are expensive and sometimes hardly possible to do, especially if trainings are to be made in critical or life-endangering environments. Such an area of application demands both, the accuracy simulations and the real-time capability to immediately respond to events and actions of the trainees. [BKJC12] The commercial driving simulator National Advanced Driving Simulator (NADS) is one of the biggest and most complex driving simulators [NAD10]. Vortex is also a commercial simulator that trains and tests the behavior of vehicles, robots and heavy equipment [CML13]. Algoryx is a physically-based commercial simulator for vehicles, ship management and robot simulations. As a multi-physics solution, it is able to simulate rigid bodies, fluids, clothes or particle systems. [Alg13] Gazebo is an open source solution for multi-robot simulations in free environments in consideration of rigid body physics [Koe13].

In the field of digital prototyping and in the area of industrial training simulators, the visualization and simulation of virtual products put high requirements on hardware and software. It is difficult to make them useable and accessible to all kinds of users. The option of visualizing and simulating virtual products by using standard web technologies, however, extends the field of the application of virtual products. Now, not only designers are able to use and access such application but all persons involved in the lifecycle of a product, i.e. marketing or web-based training purposes. [Gar13]

Since HTML5 has been introduced as the new web standard, it is possible to visualize 3D models via X3DOM in the web. X3DOM is a framework for the visualization of X3D models in a web browser based on WebGL. X3D, the successor of VRML, is a description language for 3D models which is based on XML. The use of X3D has several advantages. First, X3D is an ISO standard for dynamic and interactive 3D content. It is not only supported by X3DOM but by many different plug-ins as well. Besides, WebGL-based libraries such as SceneJS or Three.js provide a JavaScript API, meaning that 3D and DOM contents are decoupled from each other. The X3DOM framework directly integrates the X3D node into the HTML DOM tree and thereby follows the current HTML specification. This way, 3D content can be generated, manipulated and removed the same way as text, images, audio and video. Furthermore, X3DOM requires no additional plug-ins. Issues like the incompatibility on different platforms or missing rights for the installation of required plug-ins that are not installed by default on most systems can be avoided. [BEJZ09, BJK*10, JLS*13]

Especially for the development of web-based virtual products, for training purposes or for the verification of product designs, the application of X3D offers a key advantage. The bases for the production of products are CAD data from the design engineering. With the aid of suitable CAD programs that provide VRML or X3D export interfaces, CAD models can be converted into a VRML or X3D format. By using X3DOM, these models can be included and visualized on websites and can also be manipulated and interactively animated in the web browser. However, physically correct behavior such as gravity and collision detection is not included. There are only a few very basic functionalities. Therefore, first physics engines based on JavaScript were developed for the web in the last years. They often are ports for already existing physics engines, written in Java or C++.

The use of standard web technologies makes applications accessible to all users. All that is required is a web browser which is compatible with WebGL in order to validate and test the functionality of virtual products (machinery, equipment, e.g. cranes) according to physical laws.

3. Selecting a Suitable Physics Engine for the Coupling with X3DOM

This paper intends to present a guideline for developing an interface between a physics engine and X3DOM. For the coupling with X3DOM a suitable physics engine is required. It has to meet the following criteria:

- based on JavaScript
- 3D Physics Engine
- Open Source

For this purpose, a number of physics engines are available. [Pra12] provides a comparison of the 3D physics engines Ammo.js, Cannon.js and JigLibJS and the 2D physics engine box2dweb. Based on his findings, the test environment (see section 3.3) of [Pra12] was adopted and the 3D engine Three.js was replaced by X3DOM to visualize the scene. While [Pra12] provides a general overview, this comparison exclusively focuses on the coupling with X3DOM. For the comparison, the following criteria are considered:

- features
- integration capability
- performance

In addition to the 3D physics engines comparison of [Pra12], a fourth engine Bullet.js was selected. While Ammo.js is based on the C++ library of Bullet, Bullet.js is based on the Java library of Bullet. In conclusion, the following libraries have been considered and compared with each other:

- Cannon.js (http://schteppe.github.io/cannon.js/)
- Bullet.js (http://code.google.com/p/bulletjs/)
- JigLibJS (https://github.com/supereggbert/JigLibJS)
- Ammo.js (https://github.com/kripken/ammo.js)

3.1. Features

Talking about features does not only address general functions of each library as in the case of [Pra12]. It has to be

particularly evaluated, if necessary features for the coupling with X3DOM are available. The main focus is on types of shapes and constraints that are supported by each physics engine.

In X3DOM there are built-in shapes (e.g. boxes, spheres, cylinders) as well as user-defined shapes (e.g. IndexedFaceSet). Especially the IndexedFaceSet node plays a significant role in the generation of user-defined 3D objects in X3DOM. It consists of a set of points, faces which are generated from the points and normals. The individual physics engine has to provide appropriate shape types to derive an equivalent physical object based on the data of a 3D object in X3DOM. Beyond that, appropriate constraints must be available to connect several objects to a multi-body system as well.

Cannon.js has a limited range of shape types. These include geometric primitives (boxes, planes, spheres), composite shapes (compounds), particles and the user-defined shape Polyhedron. The constraints include Contact, Distance, Point2Point and Hinge. In comparison to the other physics engines, Cannon.js does not have as many features but important shape types and constraints which are necessary for the coupling with X3DOM.

Bullet.js partly features the extensive functionality of the Java library JBullet. While numerous built-in and user-defined shape types are available, the amount of constraints is rather small but important constraints such as Hinge and Generic6Dof are supported. However, the application of the constraints and the meaning of the function parameters and their settings are difficult to understand.

JigLibJS has a limited set of shape types for basic geometric primitives. Shapes for generating user-defined 3D objects are missing. Similar to Cannon.js it also provides constraints such as MaxDistance, Point, PointWorld and Hinge, whereby the hinge constraint does not seem to run stable or at least faulty. JigLibJS has a wider range of features compared to Cannon.js but lacks essential features for the coupling with X3DOM.

Ammo.js has the most extensive library compared to the other physics engines. Since the C++ source code of Bullet has been automatically translated into JavaScript, Ammo.js should have the same functionalities as the original physics engine. Accordingly, a number of built-in and user-defined shape types and many constraints are supported which are also necessary for the coupling with X3DOM.

3.2. Integration capability

Integration capability means to determine how high the effort is to develop an interface between a physics engine and X3DOM. It is particularly important whether or not 3D data can be used directly in X3DOM to generate an equivalent physical object or if complex calculations are needed for conversions in advance. It is also examined if the required position and rotation data which were calculated by the physics engine can be extracted and transferred directly to X3DOM or whether conversions are required, too.

Cannon.js: The derivation of a physical model based on the 3D model can be easily applied not only on geometric primitives but also on user-defined objects based on an IndexedFaceSet in X3DOM. The user-defined shape Polyhedron is made the same way as an IndexedFaceSet and consists of points and associated normals and faces which can be extracted directly from an IndexedFaceSet in X3DOM. The necessary values for the position and rotation of the objects in the 3D model may be extracted from the physical model and transferred to the 3D scene without further conversions.

Bullet.js: The necessary values for the position and rotation of the objects in the 3D model may be extracted from the physical model without further conversions and transferred to the 3D scene. Since dealing with the constraints already was inconclusive and led to unsatisfactory results, no time has been invested to examine the feasibility of generating user-defined shapes based on the data of an IndexedFaceSet node.

JigLibJS: JigLibJS has fundamental differences in its application compared to other physics engines that need to be figured out. The specification of the dimension is not always in the typical xyz but xzy order. The derivation of a physical model based on a 3D model in X3DOM can be implemented easily for geometric primitives. Corresponding user-defined shapes are missing. While the position of the objects can be directly extracted and transferred to the 3D scene, the rotation data are available in a 4x4 matrix and have to be converted to the quaternion format.

Ammo.js: The derivation of a physical model can be easily implemented for geometric primitives. Since many user-defined shape types are available, it is initially quite cumbersome to prove which shape is the best in order to create an equivalent object in the physics engine based on an IndexedFaceSet in X3DOM. However, it cannot be denied that user-defined shapes that are used to derive a physical object based on the data of an IndexedFaceSet are available. In Ammo.js, the necessary values for the position and rotation of the objects in the 3D model may also be extracted from the physical model and transferred to the 3D scene without any further conversions.

3.3. Performance

The physics engines are JavaScript libraries which can be included as an external file in the HTML page with the 3D model. An additional script must be developed that provides an interface between the 3D model in X3DOM and the JavaScript library of the physics engine by deriving a suitable physical model based on the 3D model in X3DOM

and facilitating a permanent synchronization between the physics engine and X3DOM during the simulation. In this way, the 3D model permanently receives the calculated motion data (position, rotation) of the individual objects.

According to the 3D scene from [Pra12], the test scene for the performance test consists of a ground plane and two ramps. Balls are generated in predefined time intervals. Running over the ramps, they drop to the bottom (Figure 3).

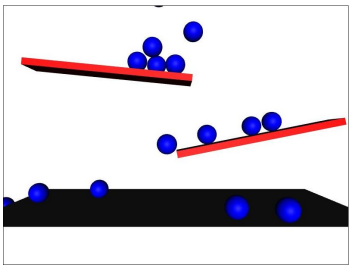

Figure 3: *Virtual test scene.*

The test was performed on a workstation with an Intel®Core™i7-3720QM CPU and an NVIDIA Quadro K1000M, running Windows 7 Enterprise 64 Bit (Service Pack 1). The web browser was Firefox 22.0.

In order to evaluate the performance, the runtimes of 1,000 simulation steps were measured and compared with each other. Additionally, the frame rate was determined for this period.

Table 1: *Runtime over 1000 simulation steps.*

Engine	Runtime
Cannon.js	23.823 ms
Bullet.js	69.0171 ms[1]
JigLibJS	25.407 ms
Ammo.js	44.426 ms

Table 1 lists the values of each measured runtime for each physics engine. Figure 4 shows the frame rates of the individual physics engines over the period of 1,000 simulation steps.

It must be considered that the performance is influenced by routines for measuring the time and the frame rate. At the beginning of the simulation the start time is measured and after 1,000 simulation steps the end time is taken. Determining the total running time is done by computing the difference between end and start time. An additional external script is included which measures the frame rate and is called for each simulation step to determine the frame rate over the entire term. Furthermore, it needs to be stated that the measured values for the duration and frame rate are only snapshots, which may slightly vary with any repeated tests. They, however, indicate a clear trend that can be used to assess the performance.

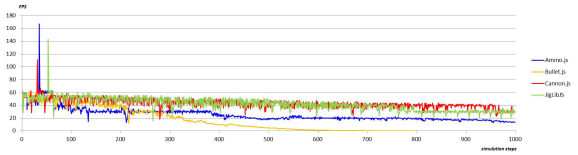

Figure 4: *Frame rate over a period of 1,000 simulation steps.*

Cannon.js is a lightweight engine which has been newly introduced and is still under development. It does not feature any port of an existing physics engine but has been written from scratch in JavaScript and developed appropriately optimized for the web. In comparison to the other physics engines, Cannon.js delivers the best performance with the shortest runtime and a frame rate which remains relatively constant with only minor fluctuations over the entire runtime of the simulation. The frame rate only drops slightly at the end of the term.

Bullet.js is a port of parts of the physics engine JBullet written in Java. It does not constitute an automatically generated port and, in comparison to the other physics engines, has the worst performance with the longest runtime. The frame rate also falls significantly. Both duration and frame rate could only be determined for 800 simulation runs due to its fairly poor performance. After that, the browser crashed and could not run the script anymore.

JigLibJS is a port of the physics engine JigLib written in C++. JigLibJS is not an automatically generated port but has been developed in JavaScript and is optimized for the web just like Cannon.js. Generally, the whole library will be integrated as a complete script in the website. In JigLibJS, each class of the library must be included separately in the website. This improves performance since only the necessary classes have to be included and loaded. Therefore, JigLibJS has a similarly good performance as Cannon.js regarding the duration with a relatively constant frame rate. Anyway, at the end of the term the frame rate decreases more than with Cannon.js.

Ammo.js is a port of the existing physics engine Bullet and is also written in C++. In this case it is an automatically generated Emscripten port. Basically, the JavaScript code of automatically generated ports is not optimized for the web. The run time is longer compared to Cannon.js and JigLibJS but nevertheless, Ammo.js provides a good runtime result compared to Bullet.js. The frame rate is slightly worse compared to Cannon.js and JigLibJS. It falls between 50 and 100

[1] Run time for only 800 simulation steps due to browser crash and performance issues.

simulation steps the first time and then between 350 and 400 simulation steps the second time. Otherwise, the frame rate shows only minor fluctuations.

It is remarkable that at the end of the term the frame rate is systematically reduced. The reason for this is the increasing number of 3D objects because new balls are generated in predefined time intervals. For each 3D object a collision detection needs to be performed. This requires high computing power. That means an increasing number of 3D objects continues to be spawned at the price of lower performance.

3.4. Summary

Before a conclusion is presented, the following table is meant to provide an overview of pros and cons for all four engines in the first place.

Table 2: *Pros and cons of each physics engine.*

Criteria	Cannon.js	Bullet.js	JigLibJS	Ammo.js
Performance	very good	poor	very good	good
Amount of shape types	small	numerous	more than Cannon.js	numerous
Important built-in shape types	yes	yes	yes	yes
Important user-defined shape types	yes	no statement possible	no	yes
Amount of constraints	small	small	small	numerous
Important constraints	yes	yes	no	yes
Functionality of constraints	feasible	inconclusive	inconclusive	feasible
Data exchange	no conversions required	no conversions required	conversions required	no conversions required
Developing activities	yes	inactive for more than two years	inactive for more than two years	yes

It can be concluded that the application of the constraints basically was consuming in all four physics engines. It always required a little effort before the optimal settings could be figured out. The best credits were given to Cannon.js and Ammo.js. Compared to Ammo.js, Cannon.js is more compact, more comprehensible, more powerful with regard to its performance and also easier to understand. It is surely preferred over Ammo.js. However, depending on the use, it must be taken into account whether the available features are already sufficient. For the prototypical development of an interface for the coupling with X3DOM which is presented in the following chapter the Ammo.js physics engines was chosen.

4. Interface between X3DOM and a physics engine

This chapter presents a guideline for developing an interface between the physics engine Ammo.js and X3DOM.

The 3D model used is based on a davit system courtesy of Davit International GmbH. It is written in X3D. A davit system is a life-saving system for life or rescue boats which are installed on larger vessels like cruise liners etc. The model represents one of the cranes which are used to lift lifeboats over a ship's side.

The system consists of two fixed components - the base and the fixation which are stationary. The other components - main arm, small arm, outer push cylinder and inner push cylinder - are moveable parts. All components have to be connected by constraints. The connection points are circled in red (hinge constraint) and orange (slider constraint) (Figure 5).

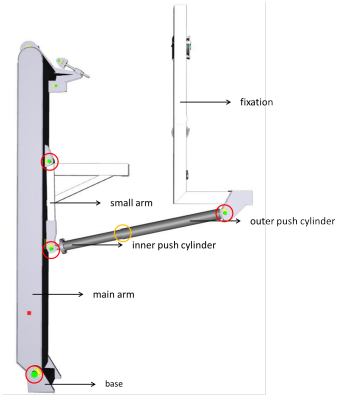

Figure 5: *3D model of a davit system.*

An XHTML file holds the X3D model within the HTML DOM tree and includes the library for the physics engine Ammo.js and the JavaScript interface.

The coupling process is divided into three phases which are described in detail in the following chapters:

- pre-processing of the 3D model
- derivation of the physical model
- simulation

4.1. Pre-processing of the 3D model

Depending on the complexity of the model, a reduction of the geometry might be required. Complex models in X3D consisting of a large number of IndexedFaceSets can be very performance-hungry for the physics engine, because each IndexedFaceSet represents a single rigid body in the physics engine. In a car model for instance, every single minor component such as bolts, door handle and hubcaps, is described by an IndexedFaceSet. However, for the physics engine it is quite sufficient if several objects such as tires, hubcaps,

rims, bolts, etc. are summarized as a single object wheel. In the field of virtual product development, so-called Digital-Mock-Ups (DMU) are developed for the verification in the design phase, such as the simulation of geometrical problem definitions during the reassembly of the components of a product. Instead of the geometry of a product being generated in a CAD system, a DMU only contains the enveloping geometry of the components of a product thus reducing the amount of data. [BNM13]

Different tools are available in order to reduce the geometry of 3D models. In the presented application, the command line tool aopt and the graphics manipulation tool Right Hemisphere were used. Aopt is part of the mixed reality system Instant Reality.

Removing non-visible and non-relevant objects

To optimize the model in a first step, all non-visible and non-relevant parts not necessary for the physical simulation of a 3D object are removed (e.g. bolts). The tool Right Hemisphere allows to manually remove non-relevant and non-visible objects from the scene tree where all the objects of a scene are hierarchically structured.

Merging of several IndexedFaceSets

Afterwards, several IndexedFaceSets of an object are merged into a useful single IndexedFaceSet with the help of the command line tool aopt by using the following command line:

```
-i [input.x3d] -F Scene:"maxtris(4000000)" -x [output].x3d
```

Reducing the geometry

Depending on the requirements, the remaining IndexedFaceSets can be compressed again by reducing the number of triangles. As a result, a more abstract 3D object can be generated. The difference between reduced and non-reduced geometry is illustrated in Figure 6. In the case of the 3D object with reduced geometry, non-visible and non-relevant objects were removed and the number of triangles was reduced by 90%.

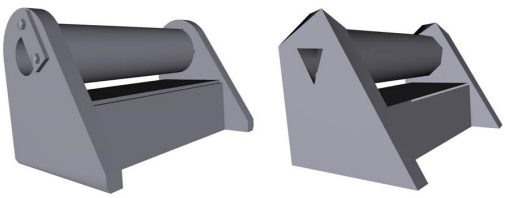

Figure 6: *3D object without and with reduced geometry.*

Table 3 shows the potential of compression by illustrating the differences between the original object and the compressed 3D object based on the criteria number of objects, points and triangles.

Table 3: *Comparison of original and compressed 3D object.*

	level 0[2]	level 1[3]	level 2[4]
objects	16	16	6
points	3300	340	66
triangles	6596	660	120

Reducing the geometry improves the performance, especially for very complex 3D models. However, for the quality of the simulation two criteria have to be taken into account - performance und accuracy. Accuracy mainly concerns the coupling of individual rigid bodies by constraints, which significantly affects the movement of rigid bodies to one another and the quality of the simulation results regarding physically correct behavior. It can be stated that the reduction of geometry is limited because an increased reduction will affect the quality of the simulation. There is a conflict between performance and accuracy needs since good performance often comes at the expense of accuracy. The challenge is to achieve a good balance between these two requirements.

Converting the X3D model into a X3DOM model

After having reduced the geometry, the given X3D model can be converted into a X3DOM model by using the command line tool aopt while typing the following command line:

```
-i [input.x3d] -M [output].xhtml
```

Adding metadata and required X3D nodes to the 3D model

Another precondition for the derivation of the physical model is the compliance with relevant modeling conventions in X3DOM. It must be ensured that each object in X3DOM is included by a transform node. Transform nodes hold information on translation and rotation in order to correspondingly move and rotate the object in all three dimensions. The two attributes are addressed during the simulation in a way that they permanently receive the current position and rotation data from the physical model. Therefore, missing transform nodes in the 3D model must be generated automatically. A script parses the whole X3D tree and checks if every shape node has a transform node as a parent node. Otherwise, the missing transform node will be generated. Additionally, each object in the 3D model must be provided with an Id that is defined in the accompanying transform node. In the derivation of the physical model, the equivalent physical object receives the same Id as the 3D object in X3DOM.

[2] No compression.
[3] Compression of the geometry by 90%.
[4] Removing non-visible and non-relevant objects, followed by a compression of the geometry by 90%.

That way, a distinct mapping is guaranteed for the physical and the 3D object, so that the calculated position and rotation data of the physical simulation can be transferred to the associated 3D object in X3DOM. While parsing the whole X3D tree in order to generate missing transform nodes, an additional attribute Id will be added to each transform node with an associated value.

Each generation of a physical model requires the specification of a mass which also affects the movement of objects. Based on the 3D model, no information of the mass of the individual objects can be derived. Depending on the application, realistic data are required to simulate a realistic expressive behavior. If the necessary data are available, they can be added to the individual 3D objects in X3DOM as metadata. These metadata can then be extracted automatically in the derivation of the physical model. This paper, however, merely examines the functionality and feasibility of such an application, so that only assumptions on the mass of objects were made.

In this phase, only the last step can be automated and represents the first function of the interface between X3DOM and the physics engine. The previous steps manipulate the 3D model with the use of different tools. The whole workflow is illustrated in Figure 7.

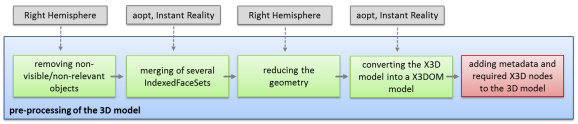

Figure 7: *Pre-processing of the 3D model.*

4.2. Derivation of the physical model

In the derivation of the physical model based on the 3D model a distinction is made between the generation of rigid bodies and the definition of constraints for connecting rigid bodies to a complex multi-body system.

4.2.1. Rigid Bodies

As a first step, the derivation of the physical model requires the identification of all the geometric objects in the X3DOM model. For this purpose, the whole scene within the X3D node has to be parsed in order to find all transform nodes. At first, the transform node holds the value of the associated Id of the object which has to be extracted and assigned to the physical model in order to match physical und 3D objects during the simulation. In the next step, the transform node includes a shape node that consists of a geometry node, which indicates the type of the shape, and an appearance node. For every geometry node, the required geometric data have to be extracted. Depending on the shape type, the library of the physics engine offers different functions for different shape types. In order to match a shape type with the appropriate function, it is necessary to predetermine what kind of shape type calls which function. These functions indicate which geometric data are needed and how they are processed in order to generate an equivalent physical object.

The derivation of physical objects for built-in shapes such as geometric primitives is very simple, because geometric primitives are defined by using fixed principles. While a box is defined by the edge lengths of width, depth and height in computer graphics, a ball is basically defined by its radius.

For user-defined shape types the creation process is much more complicated because there are different types of modeling depending on the physics engine.

In X3DOM user-defined 3D objects are usually defined as an IndexedFaceSet (illustrated in Listing 1), which consists of points, normals and faces.

```
<Transform Id='IFS01' translation='0 0 0' rotation='0 0 1 0'>
  <Shape>
    <Appearance></Appearance>
    <IndexedFaceSet coordIndex=' '/>
      <Coordinate point=' '/>
      <Normal vector=' '/>
    </IndexedFaceSet>
  </Shape>
</Transform>
```

Listing 1: *Definition of a user-defined 3D object as IndexedFaceSet in X3DOM*

Extensive libraries as offered in Ammo.js mean different types of user-defined shape types that must first be tested for their suitability.

In early implementations the following shapes have been found potentially suitable convex shape types in Ammo.js which can approximate the geometry of a 3D object for the physical model:

- *btConvexHullShape*: implements an implicit convex hull of an array of points [Cou12], the points of the IndexedFaceSet nodes can be extracted in X3DOM
- *btConvexTriangleMeshShape*: implements a convex hull of a *btTriangleMesh*, that defines a mesh of an array of triangles (each consisting of three points) [Cou12], the triangles can be derived from the points and faces of the IndexedFaceSet nodes in X3DOM

For performance reasons it is recommended to use *btConvexHullShape* for user-defined 3D objects [Cou12].

The whole workflow can be automated and represents a further function of the interface between X3DOM and the physics engine.

4.2.2. Constraints

So far, there is no mechanism to automatically derive constraints based on a 3D model in X3DOM. The 3D model lacks required data that indicate which objects have to be

connected by a constraint. This requires the semantic enrichment of the 3D model while the parameter settings for each constraint are usually very individual. Due to the complex dependencies when using constraints, it is normally required to test the settings of constraints in the finished simulation model in order to get their optimal setting. Thus, it is not possible to define the optimal settings and add them as metadata to the 3D model in advance.

So far, each constraint has to be defined individually. First and according to the application it must be selected between different types of constraints (e.g. PointToPoint, Hinge, Slider). Depending on the type of the constraint various parameters and functions are available. Each type of constraint has its own settings in order to manipulate the behavior of objects.

In the following, parameters and feature settings of the hinge constraint shall be presented which were increasingly used in the prototypical implementation. First, two bodies that are meant to be connected by a constraint must be specified. Then, a pivot point has to be determined for each body. By default this is the center of each body with the local coordinate (0, 0, 0) (Figure 8). If two boxes, for example, are supposed to be connected to each other on their edges along the z-axis, the pivot points of both bodies must be shifted right or left along the x-axis (see Figure 8), so that both pivot points are between the boxes. The hinge constraint limits the bodies in two degrees of freedom so they can only rotate around one axis. This axis is called the hinge axis, and must also be specified for each of the two bodies when defining a constraint. Furthermore, optional maximum and minimum limits can be set to limit the angle of body rotation around the hinge axis. With the help of motors, the movement of the bodies can be controlled. For this purpose, forces and pulses, the direction of the force and the speed can be set to simulate the automatic opening and closing of a door for instance. Listing 2 shows an example of a definition of a hinge constraint.

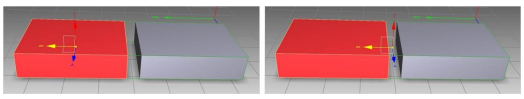

Figure 8: *Box without and with shifted pivot point.*

```
var hingeConstraint = new Ammo.btHingeConstraint(
    bodyA,\\bodyA
    bodyA,\\bodyB
    new Ammo.btVector3(0,0,0), \\pivot point of bodyA
    new Ammo.btVector3(0,0,0), \\pivot point of bodyB
    new Ammo.btVector3(0,0,1), \\hinge axis of bodyA
    new Ammo.btVector3(0,0,1), \\hinge axis of bodyA
    false); \\use reference frameA

var min = -Math.PI/2, max = Math.PI/2;
hingeConstraint.setLimit(min, max);
```

Listing 2: *Definition of a hinge constraint*

The above settings must be set individually for each defined constraint even if it is the same kind of constraint. The values of the parameters can vary depending on the desired behavior. Currently, the process of constraint definition cannot be automated and is very expensive.

4.3. Simulation

During each simulation step, the position and rotation data are updated in the last phase of the simulation loop as described in chapter 1. These data can be extracted for each body and are transferred to the corresponding 3D object in X3DOM. In Ammo.js no data conversion is required before transferring the data. Each physical object has an ID. It is used to get access to the transform node in the X3D tree with the same ID via JavaScript DOM by using the getElementById command to update the corresponding attributes of translation and rotation with the calculated position and rotation data (see Listing 3 and 4). In this way, a permanent synchronization between the physical and 3D model is possible.

```
<Transform Id='IFS01' translation='0 0 0' rotation='0 0 1 0'>
  <Shape>
    <Appearance></Appearance>
    <IndexedFaceSet coordIndex=' '/>
      <Coordinate point=' '/>
      <Normal vector=' '/>
    </IndexedFaceSet>
  </Shape>
</Transform>
```

Listing 3: *X3DOM - transform node with enclosed IndexedFaceSet node*

```
for(var i=0; i<bodies.length; i++){
  //Get position and rotation
  bodies[i].getMotionState().getWorldTransform(transform);
  var origin = transform.getOrigin();
  var r = transform.getRotation();
  quat.setValue(r.x(),r.y(),r.z(),r.w());
  var a = quat.getAxis();

  document.getElementById(' ').setAttribute('translation',
        origin.x()+''+origin.y()+''+origin.z());
  document.getElementById(' ').setAttribute('rotation',
        a.x()+''+a.y()+''+a.z()+''+quat.getAngle());
}
```

Listing 4: *Ammo.js - access to the transform node via JavaScript DOM*

The update of the 3D model during the simulation represents the last function of the interface between X3DOM and the physics engine.

Figure 9 illustrates the data exchange between the 3D model and the physical model via the interface.

Presently, all functions can be automated except the derivation of constraints. Besides, all steps that are responsible for the reduction of the 3D model are not automated and decoupled from the interface. They require the use of external tools.

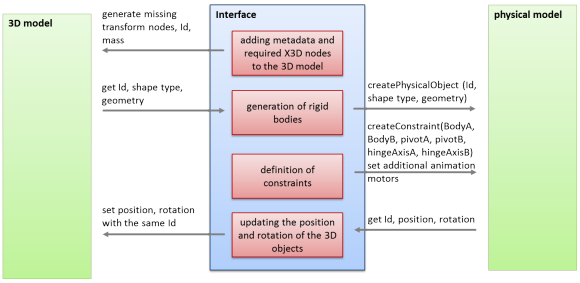

Figure 9: *Data exchange between the 3D model and the physical model via the interface.*

5. Conclusion

The current progress of coupling X3DOM with JavaScript physics engines shows the possibilities, potential and challenges of such an interface. Basically, a coupling of X3DOM with physics engines is possible and offers great options of extending the application field of web-based 3D models with respect to physically realistic behavior. That is particularly in terms of the interactive control of virtual products in a web browser to verify their functionality while being in the design or testing phase.

There are, however, still some challenges to overcome. This will be the subject of future work. Firstly, solutions for the derivation of the physical model need to be optimized in order to develop an appropriate method that meets the requirements of both accuracy and performance. Secondly, the issue of authoring should be in the focus of attention plus the question on how to develop an interface between X3DOM and physics engines as much generically as possible in order to simplify the process of pre-processing the geometry of the 3D model and transferring the geometric data to the physical model. In this context, [CP04] proposes a generic framework which takes every step of the coupling process into account. The generic formalism, a so called Physical Modeling Language (PLM) based on XML, organizes the geometry representation, the definition of the properties and the behavior and the specification of the constraints and animation motors in order to make the process of deriving a simulation model much more comfortable. The approach was originally developed for continuous and discrete models in the field of medicine but can be used for any type of approach.

References

[Alg13] ALGORYX: Multiphysics and 3d simulation, 2013. URL: http://www.algoryx.se. 3

[BEJZ09] BEHR J., ESCHLER P., JUNG Y., ZÖLLNER M.: X3dom - a dom-based html/ x3d integration model. In *Web3D 2009, Darmstadt, Germany, June 16 - 17* (2009). 3

[BJK*10] BEHR J., JUNG Y., KEIL J., DREVENSEK T., ZOELLNER M., ESCHLER P., FELLNER D.: A scalable architecture for the html5/ x3d integration model x3dom. In *Web3D 2010, Los Angeles, California, June 24 - 25* (2010). 3

[BKJC12] BENDER J., KENNY E., JEFF T., COUMANS E.: Interactive simulation of rigid body dynamics in computer graphics. In *EUROGRAPHICS 2012, the 33rd Annual Conference of the European Association for Computer Graphics* (2012). 1, 2, 3

[BNM13] BEUTNER E., NEUKIRCHNER H., MAAS G.: *Virtuelle Produktentwicklung*. Wuerzburg: Vogel Buchverlag, 2013. 7

[Box13] BOX2D: A 2d physics engine for games, 2013. URL: http://box2d.org/. 2

[CML13] CMLABS: Vortex by cmlabs - behaviour in motion, June 2013. URL: http://www.vxsim.com. 3

[Cou12] COUMANS E.: *The bullet physics library*, 2012. URL: http://bulletphysics.org. 2, 8

[Cou13] COUMANS E.: Gpu rigid body simulation. In *Game Developer Conference 2013* (2013). 2

[CP04] CHABANAS M., PROMAYON E.: Physical model language: Towards a unified representation for continuous and discrete models. *Springer-Verlag Berlin, Heidelberg* (2004). 10

[CYB13] CYBERBOTICS: Webots 6, 2013. URL: http://www.cyberbotics.com/products/webots. 2

[Diz12] DIZIOL R.: *Simulation inkompressibler deformierbarer Körper*. PhD thesis, Karlsruhe Institute of Technology (KIT), Department of Informatics, 2012. 1

[Gar13] GARIN F.: Digital mock-up management inside the web browser using standard html. In *Web3D 2013, San Sebastian, Spain, June 20 - 22* (2013). 3

[JLS*13] JUNG Y., LIMPER M., STEIN C., WAGNER S., STORK A.: Fast delivery of 3d web content: A case study. In *Web3D 2013, San Sebastian, Spain, June 20 - 22* (2013). 3

[Jon11] JONES M. T.: Open source physics engines. *developerWorks* (2011). URL: http://public.dhe.ibm.com/software/dw/opensource/os-physicsengines/os-physicsengines-pdf.pdf. 1

[Jun11] JUNG T. J.: *Methoden der Mehrkörpersimulation als Grundlagen realitätsnaher virtueller Welten*. PhD thesis, RWTH Aachen University, 2011. 2

[KJ11] KAPELLOS K., JOUDRIER L.: Planetary exploration missions simulation using 3dro. In *Euromech colloquium on "Nonsmooth contactand impact laws in mechanics", Grenoble, France, July 6th-8th 2011* (2011). 2

[Koe13] KOENIG N.: Gazebo, 3d multiple robot simulator with dynamics, 2013. URL: http://gazebosim.org/. 3

[mac13] MACHINEERING: Simulaton - automation, 2013. URL: http://www.machineering.de/simulation.html. 1, 2

[MIC13] MICROSOFT: Microsoft robotics, 2013. URL: http://www.microsoft.com/robotics. 2

[NAD10] NADS: National advanced driving simulator, 2010. URL: http://www.nads-sc.uiowa.edu/. 3

[Pla12] PLAKU E.: Motion planning with discrete abstractions and physics-based game engines. *Springer-Verlag, Berlin Heidelberg, 2012 7660* (2012), 290–301. 2

[Pra12] PRALL C.: Javascript physics engines comparison, August 2012. URL: http://buildnewgames.com/physics-engines-comparison/. 3, 5