2014-08

Enhancing X3DOM Declarative 3D with Rigid Body Physics Support

Stamoulias, Andreas

þÿ W e b 3 D 2 0 1 4 , A u g u s t 0 8 1 0 , 2 0 1 4 , V a n c o u v e r , B r i t i s h C o l u m b i a , C a n a d a
http://hdl.handle.net/10945/46360

Enhancing X3DOM Declarative 3D with Rigid Body Physics Support

Andreas Stamoulias, Athanasios G. Malamos, Markos Zampoglou
Department of Informatics Engineering
Technological Educational Institute of Crete
Heraklion, Greece, GR 71004
andr.stamoulias@gmail.com

Don Brutzman
Naval Postgraduate School, Code USW/Br
Watkins 270, MOVES Institute
Monterey CA 93943-5000 USA
+1.831.656.2149
brutzman@nps.edu

Abstract

Given that physics can be fundamental for realistic and interactive Web3D applications, a number of JavaScript versions of physics engines have been introduced during the past years. This paper presents the implementation of the rigid body physics component, as defined by the X3D specification, in the X3DOM environment, and the creation of dynamic 3D interactive worlds. We briefly review the state of the art in current technologies for Web3D graphics, including HTML5, WebGL and X3D, and then explore the significance of physics engines in building realistic Web3D worlds. We include a comprehensive review of JavaScript physics engine libraries, and proceed to summarize the significance of our implementation while presenting in detail the methodology followed. The results obtained so far from our cross-browser experiments demonstrate that real-time interactive scenes with hundreds of rigid bodies can be constructed and operate with acceptable frame rates, while the allowing the user to maintain the scene control.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—;

Keywords: X3D, X3DOM, Web3D, HTML5, WebGL, Bullet physics, Ammo.js, real-time, physics engines, rigid body, constraint, interactive 3D

1 Introduction

The technologies designed for 3D visualization within the web started by HTML in 1995, when a text based meta-language inspired by HTML was proposed for describing 3D scenes in terms of geometry and material properties, the so-called Virtual Reality Modeling Language (VRML). The concept of real-time 3D content rendering on the web was a reasonable next step following the first 2D multimedia content modalities introduced in HTML pages. In these first implementations, it was required to install a platform-specific plug-in for the rendering of the scene, either an open one, such as VRML or its XML-based successor, X3D [Daly and Brutzman 2007], or a proprietary one, such as Flash, Silverlight, ShockWave and Quicktime.

OpenGL is widely known as low-level programming language used for direct communication with a system's graphics card. A novel variation of OpenGL, especially modified for JavaScript wrapping and browser compatibility, called WebGL (Web Graphics Library) [WebGL 2014], is a JavaScript API designed and maintained by the non-profit Khronos Group, based on OpenGL ES 2.0. Using WebGL, the improved HTML5 canvas element can be used to display 3D environment on a browser. The successful mixing of the above technologies has led to the advent of X3DOM [Behr et al. 2010], a technology that allows for client-side rendering of a 3D virtual world with no external plug-in requirements. The X3DOM framework comprises the X3D API and WebGL capabilities to display a dynamic and interactive 3D environment on a simple web page natively, without the need for plugins. Furthermore, the X3D physics component provides the ability to enhance a scene with a subset of the laws of physics, known as Rigid Body Physics, in order to influence the elements of a 3D scene. This is essentially a process in which, at each time point, the position and motion of all objects is estimated from their previous parameters, the influence of various physical forces on them (such as gravity or collisions) and a set of natural laws - typically through the use of differential equations.

However, while X3D includes the specification of Rigid Body Physics, there is no current X3DOM implementation. In truth, there is little work in general, for Web3D physics running natively on a browser. In this paper, we present our implementation of a physics support framework for X3DOM, and our performance evaluations of this framework for a range of platforms and scenarios. Our work aims to produce an interactive 3D graphics system, based on declarative, text-based 3D scenes and able to natively run within a browser. The rest of the paper is organized as follows: Section 2 gives the relevant background for our work. Section 3 presents our proposed and implemented system architecture, and Section 4 presents the experimental evaluations of our framework. Finally, Section 5 summarizes our conclusions, and Section 6 lays the groundwork for our future steps.

2 Background

2.1 HTML5 and WEBGL

WebGL is a 2D rendering API designed as a drawing/rendering context for the HTML Canvas element that provides rendering functionalities similar to OpenGL ES 2.0, by giving access to the GPU hardware from JavaScript. The HTML Canvas provides a destination for programmatic rendering in web pages, which allows the use of different rendering APIs for this task. The Canvas specification comes only with the CanvasRenderingContent2D Interface, which defines the 2D drawing context for the HTML canvas element and provides a basic set of methods and properties that enables us to draw and manipulate graphics on the canvas drawing surface. WebGL, on the other hand describes another such interface, named WebGL Rendering Context, which provides its own special properties and methods to allow 3D content rendering and manipulation within an HTML canvas element.

WebGL technology is based on scripts called shaders that work to-

Web3D 2014, August 08 – 10, 2014, Vancouver, British Columbia, Canada.

gether for the visual representation of an object on the screen. There are essentially two shaders, the Vertex Shader and the Fragment Shader. The former is responsible for vertex coordinate transformations and, as a result, estimation of the final vertex positions, an operation that may include per-vertex projection matrix calculations, normal vector and texture coordinate generation and transformation, and lighting/color calculations. The minimum task for the VS is to set the GL position of the object, which is then passed to the GPU and stored internally in order to be then passed to FS. In fact the two shaders, VS and FS, work together for the final render, but, if needed, the values passed from the VS can be overridden. FS is responsible for the final color of each pixel, but can optionally perform texture lookups and discard fragments, in cases of objects obscured by another object in the scene, an operation that takes into consideration the viewport dimensions and the projection matrix.

2.2 X3D-X3DOM

X3D is a royalty-free ISO standard defining an XML-based file format for representing Web3D computer graphics. It was first accepted as an ISO standard in 2004, and was designed by the Web3D consortium with the aim of replacing the then-dominant Web3D standard language, VRML (Virtual Reality Modeling Language), which was an ISO standard since 1997 (thus often known as VRML97). X3D defines various data-encoding formats but still maintains compatibility with its predecessor technology. There are three ISO documents that compose the X3D specification, namely Abstract Functionality [X3Da], Encodings [X3Db] and Language Bindings [X3Dc]. The first document defines the architecture and abstract definitions of all X3D components alongside the X3D abstract API. The second defines the three different file encoding types, namely XML, classic VRML and Binary, while the third document defines the X3D language API bindings, which includes an ECMA-Script API to the X3D Scene Access Interface (SAI) and a Java API to SAI. The X3D standard supports a wide number of advanced 3D graphics functionalities, such as multi-texture rendering, real-time reflections and lighting, shaders with texture, normal-mapping, light-mapping, movie-texture, a deferred rendering architecture, animations, humanoid animations (H-Anim) and geospatial positioning.

X3D was designed to deliver a lightweight and balanced web-based application to offer easy to use and develop interactive real-time 3D content. To this end, Web3D Consortium introduced a profile system which is used to group the various X3D components into profiles. X3Ds component-based architecture is composed of several profiles including Core, Interchange, Interactive, Immersive, Full, MPEG-4 Interactive, CDF (CAD Distillation Format) and Medical-Interchange. This profile/ component-based architecture offers multiple advantages: For one, its possible to create or extend specific X3D functionalities without affecting the rest of the framework. Furthermore, we can add new components, and directly embed them in X3D. Finally, the profile system allows for browser/player specialization. Since a scene always specifies the profile it requires to play correctly, the browser is always aware in advance of the component requirements of the specific scene, and can only load the necessary functionalities.

X3D has already been proposed in the HTML5 specification as the technology for declarative 3D scenes. At the same time, however, the X3DOM framework has recently been created. X3DOM is an open-source project implementing an HTML5-WebGL version of the core X3D API. X3DOM operates as a connector, responsible for the synchronization of the browser frontends and the X3D backend, by monitoring DOM updates in the X3D code. Alongside the connector, X3DOMs architecture consists of the User Agent responsible for composing the final rendering output and the X3D runtime responsible for updating and rendering the scene, but also handling user interactions. The X3DOM implementation supports the X3D/SAI plugins and the WebGL backend renderer in its scene-graph construction, which means that communication, rendering and interaction are not managed by external plugins or applications, but natively handled within the web browser. Still, X3DOM also features a fallback model that handles any potential browser incompatibilities, offering the ability of switching the backend to Flash [Behr et al. 2010].

In the past, we have explored the potential of interactive 3D collaborative games for the web using X3D-X3DOM technologies [Kapetanakis et al. 2013a], [Kapetanakis et al. 2013b]. In our current work, we are more interested in extending the current limits of X3D technologies by incorporating physics functionalities, towards enhanced realism and a broader field of potential applications for interactive Web 3D graphics.

2.3 Physics Engines

The term physics engine refers to any software system that can simulate physical phenomena in the domains of graphics, video games, the film industry, scientific simulation and research. A physics engine provides an approximate simulation of certain physical systems, such as Rigid Body Dynamics, which treat all bodies as being solid, Soft Body Dynamics which can handle the physics of deformable physical bodies, and Fluid Dynamics. There are generally two classes of physics engines: real-time and high-precision. Real-time physics engines use simplified calculations, which can consistently produce results within a desired frame rate by sacrificing accuracy in their estimates. They are popular for video games and other forms of real-time interactive computing. High-precision physics engines, on the other hand, are usually used by scientists and computer animation studios for difficult or critical physics calculations. They require more processing power and completion time than real-time physics engines, but can produce high-precision results.

The main task of all physics engines is to solve the so-called forward dynamics problem. Every physics engine has a set of specific characteristics, such as the simulation paradigm, the collision detection algorithms, the collision dispatcher, the memory pool size, the simulation sub-steps and the error correction factor. The design choices for all these characteristics can greatly influence the output of a physics engine, and provide significantly different results, even when attempting to simulate the exact same system.

Real-time physics engines come in various license formats. Some 3D physics engines like Bullet [Bullet], Open Dynamics Engine [Smit 2007] and Newton Game Dynamics [Newton] are open source, while others like Havok [Havok] and PhysX [PhysX] are closed source with limited free distribution. Recently, a number of JavaScript physics engines have appeared. As we are interested in physics for Web3D scenes, we compared all available JavaScript physics engines and the JavaScript ported versions of the above C/C++ open source physics libraries, in order to find the most suitable for our implementation purposes. The most popular JavaScript physics library for WebGL is Ammo.js [Ammo]. Ammo.js is a direct port of Bullet, a real-time physics simulation engine that supports rigid-body and soft-body dynamics as well as collision detection in 3D environments. Bullet was developed by Erwin Coumans while he was at Sony Computer Entertainment. The engine is supported on a large number of platforms and most operating systems. As a production physics engine, Bullet has wide support both in games and movies and includes a rich API and an SDK.

The Bullet port to JavaScript named Ammo.js is built using Emscripten, and as a result it is not optimized for the web. Other than

that, Ammo.js is a very feature-rich library that includes many built-in shapes, continuous collision detection, constraints, and a powerful vehicle system. The build-in shapes that Ammo.js offers can describe any primitive mesh such as sphere and box, but also provide ConvexShape-type constructors for user-defined objects. Ammo.js constraints can bind one object to a specific world position, or constrain two objects relative to each other. Each type of constraint has its own settings, corresponding to the type of movement the constrained objects can perform and their limits. As for collision detection, Ammo.js, unlike any other library, does not support collision events, but a contact manifold list of these events can be accessed via the physics world, in order to search for collidable objects and the point of contact. Ammo.js also offers methods to detect all the rigid bodies and constraints in the world, so as to keep track of their parameters such as position, rotation, or velocity. Ammo.js, being a direct port of the Bullet engine does not come with its own API, but uses Bullets instead. As a result, some knowledge of Bullets library class structure is needed. Although Bullets API is not very well-documented in all areas, it can cover most of the capabilities Ammo.js can offer.

A second JavaScript physics library is JiglibJS2.js [JigLibJS2], a port of yet another C++ library named JiglibFlash. JiglibJS2.js code is automatically generated from the AS3, but with many hand-made tweaks and optimizations for the web. Also, unlike Ammo.js where there is only one file to import in a page, JiglibJS2.js offers a separate file for each class. This customizations gives JiglibJS2.js better performance and memory usage compared to Ammo.js. However, it does not offer certain functionalities that Ammo.js does, including constraints.

Another JavaScript physics library is Cannon.js [Cannon], which is inspired by Ammo.js. Cannon.js is written by Stefan Hedman and is described as a lightweight 3D physics engine for the web. Cannon.js supports rigid-body simulation with primitive shapes and custom convex polygons, constraints like point-to-point, motor and hinge joints, but is still under development. Lastly there is one more JavaScript physics library, named Physijs.js [PhysiJSa], based on Ammo.js. In fact, Physijs.js is actually build on top of Ammo.js and runs the physics simulation separately via a web worker, which leads in better computational performance. As an overview, we can see that Cannon.js and Physijs.js are based on either the actual Ammo.js or its features and class structure for their development, while JigLibJS2.js is based on JiglibFlash but lacks certain functionalities compared to Ammo.js. Ammo.js, even if it is a non-trivial physics simulation engine, nor particularly well documented, stands out as a complete and powerful physics engine that can support most of the features any other real-time physics engine comes with.

As a direct result of the breadth of implementation choices and the large number of physics engines (including multiple open and/or free ones), the need to experimentally evaluate them has arisen from relatively early on. An evaluation of a set of open source real-time physics engines, was presented in [Boeing and Brunl 2007]. As Web 3D graphics at the time was not a visible prospect, no JavaScript engines are considered. Bullet, however, in its original C++ implementation is found to be the best performing engine overall, although the results are far from conclusive with respect to individual tasks. A more recent evaluation [Yogya and Kosala 2014] compares JavaScript-based engines, but the comparison is limited between Cannon.js and Bullet.js (an older attempt to port Bullet to JS). While Cannon.js is shown to be faster but less accurate than Bullet.js, the absence of Ammo.js from the testbed is problematic.

2.4 Related work

Recently a lightweight JavaScript library/API named Three.js [PhysiJSb] was developed and released by Richard Cabello. Three.js can be used to create and visually render animated 3D graphics on a HTML5 Canvas element using SVG or WebGL. It is a plugin-free library, thanks to WebGL, supporting various lighting techniques, shaders using the OpenGL shading language, basic or custom geometry, and 3D math functions. Three.js is presented as a very low-complexity library even for a beginner graphics creator. With respect to physics, Three.js inherently supports its own limited library. However, one can achieve significantly more realistic and interactive results by combining Three.js with Ammo.js, even though this combination is not officially supported. A second example of Ammo.js usage in a WebGL visual engine is that of CubicVR.js [CubicVR]. CubicVR.js is a lightweight and high-performance JavaScript library/API, port of the CubicVR 3D Engine developed by Charles J. Cliffe, with a collection of built-in features for the production of high quality real-time 3D graphics. However, to our knowledge, no attempt to incorporate physics in declarative Web3D graphics has been made so far.

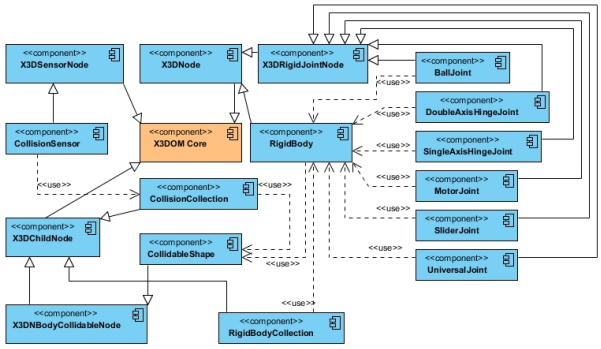

Figure 1: *X3D Rigid Body Physics component diagram.*

3 Architecture

The X3D Rigid Body Physics component was proposed as a custom extension of Xj3D [Xj3D] some ten years ago. Xj3D is a Java project focused on creating a toolkit for VRML97 and X3D content, which over time was used as one of the testing grounds for the verification of the X3D specification. The proposed X3D Rigid Body Physics component provided the ability to influence the visual output of the scene graph according to a Rigid Body Dynamics (RBD) physics model. In RBD, an object is represented by a mass, a density model, a world space position and orientation, both linear and angular velocities, and any forces acting upon it. According to the Rigid Body Physics component proposal [Matsuba et al. 2005], it allows content developers to produce ”cutting edge” real-time interactive 3D graphics applications. Our implementation of the rigid body physics elements into X3DOM is based on that same specification component.

Our work is composed of five phases: the rigid body element registration into X3DOM, the X3D scene parsing, the bullet world construction, the bullet world constraining and the realism and interactivity phase. The first phase of our work is the registration of rigid body elements inside the X3DOMs core. The second phase is responsible for the X3D scene parsing of the HTML DOM. In this phase the JavaScript collects all the CollidableShape nodes in the X3D scene and procedurally creates the HTML objects by accessing the CollidableShape, Transform, RigidBody, RigidBodyCollection and CollisionCollection nodes added into X3DOM. In the case

of joint type nodes the JavaScript code collects all the available information, in order to create the additional HTML objects that represent them. The third and fourth phases are responsible for the description of the subset of the law physics inside our scene, and the construction of the rigid bodies and the constraints between them, using bullet API. The last phase is where interactivity and realism are applied. Here we define the physics simulation step for the Bullet physics, update our X3D objects that compose our scene using the per-frame callback of the X3DOM runtime function, and lastly we define the extra event listeners in the interactive objects of our scene.

3.1 Rigid Body Physics Component at X3DOMs Core

Based on the X3D Rigid Body Physics component, we have registered the nodes described in that component inside the X3DOM core. The newly registered nodes are: CollidableShape, X3DNBodyCollidableNode, RigidBody, RigidBodyCollection, CollisionCollection, CollisionSensor, UniversalJoint, BallJoint, SingleAxisHingeJoint, DoubleAxisHingeJoint, SliderJoint, X3DRigidJointNode and MotorJoint, as defined by the specification both in terms of name and structure. Following this step, X3DOM became aware of these nodes, which can in turn allow an internal method of accessing, comparing and traversing them. The structure of the implemented components regarding the X3DOM core can be seen in Figure 1.

The RigidBody node describes the properties of a body that can be affected by physics. This body's geometry shape is defined by a CollidableShape element that the RigidBody node uses, which in turn controls the appointed shape defined in a Transform node. The RigidBody node defines properties like position and orientation vectors, mass, center of mass and inertia, angular and linear velocity vectors and damping factors in both linear and angular systems.

The Transform node is a grouping node that defines a local coordinate system with a scale, a position and an orientation for all its children. In order to connect a Transform node with a CollidableShape we have to define a Transform node using the DEF attribute and another Transform node, positioned as a child in the CollidableShape node, using the USE attribute. The first Transform node contains a child node that is responsible for rendering the visual representation of a geometry shape. This is called the Shape node. The Shape node has two fields that describe an object, namely appearance and geometry, which contain respectively an Appearance node and a geometry node. The Appearance node specifies the visual attributes like material and texture of the object, while the geometry node gives all the information about the shape of the geometry.

The RigidBodyCollection is used as a container for multiple RigidBody nodes defined by the same properties, such as gravity, error correction factors, surface thickness, or switching parameters like "enable," which tells our script if that group of objects will be considered as active.

The CollisionCollection node acts similar to the RigidBodyCollection node, but holds together multiple CollidableShape nodes under the same properties. These properties define the bounce level of the objects, the friction and the softness of the collision, slip factors, and the surface speed.

The X3DNBodyCollidableNode is the CollidableShape's abstract node type that represents an interface between the visual object in the scene and the rigid body collision geometry object.

The CollidableShape node is the one responsible for the representation of the collider's geometry shape. Without it no physics can be applied to an object, and as a result no interaction with the objects can be performed. As we mentioned above, in order to match every collider with the corresponding visual object that will be rendered into the canvas element, we have appended a Transform child node within the CollidableShape node, which USEs the already DEFined Transform inside the scene. The first Transform, placed in the scene wherever the X3D specification allows for a Transform node to be, serves for the visualization of the object. The second transform, placed inside the CollidableShape node and using the first one, has to be placed below the first one in the XML tree, for the DEF-USE relationship to work. The second Transform can either be used for the representation of the physical shape of a rigid body by defining a Shape node, or as a connection reference point to another previously defined CollidableShape node for the RigidBody element.

The X3DRigidJointNode is an abstract node type that represents all the joint type nodes and defines basic attributes like the first and second body children node fields. The joint type nodes defined into X3DOM are:

- The BallJoint type node that represents a joint between two rigid bodies, free from other constrains, that pivot about a common anchor point.
- The UniversalJoint type node represents a joint between two rigid bodies like the BallJoint node, but constrains an extra degree of rotational freedom by keeping the axes perpendicular to each other.
- The MotorJoint type node that can control the relative angular velocities between two rigid bodies.
- The SliderJoint node that represents a joint between two rigid bodies constrained along a single axis.
- The SingleAxisHingeJoint node that represents a type of joint like the traditional door hinge, which define a single axis about which the rigid bodies rotate.
- The DoubleAxisHingeJoint node, just like the SingleAxis, represents a hinge joint but this time with two independent axes that are located around a common anchor point.

3.2 Parsing the X3D Scene

Initially our JavaScript searches the HTML body it is attached with, for the X3D ¡Scene¿ element. Then it continues inside its children, searching for CollidableShape nodes, being the ones responsible for the representation of the collider's geometry shape. At first we collect all the CollidableShape nodes in the X3D scene and procedurally create an Array of JavaScript objects by accessing the CollidableShape, RigidBody, RigidBodyCollection and CollisionCollection nodes that we implemented into X3DOM. In the case of existing joint type nodes, JavaScript collects all available information and creates an additional JavaScript Object Array to store it. The connection between those nodes is performed via the DEF/USE attribute that is defined in all our rigid body elements. Specifically, the CollidableShape uses an already defined Transform in our scene and is used by the RigidBody and CollisionCollection node elements. In turn, RigidBody is defined as a child of a RigidBodyCollection, while the CollisionCollection is defined as a child of a CollisionSensor. In the case of a joint, there is one or more extra joint type nodes defined as siblings of the RigidBody. All joint type nodes use two CollidableShape nodes that have to be already defined into the scene and into their corresponding RigidBody.

3.3 Building the Physics Simulation World

The third phase of our implementation is defined as the "simulation world builder, as it is responsible for creating the physics world. In order to add physics using Ammo.js, we first need to create and

set up several Ammo.js objects. We need a collision configuration, which is responsible for the collision detection algorithms and the collision manifold pool size, a collision dispatcher that can handle collision pairs, and a broadphase that uses two dynamic AABB bounding volume hierarchies, one for static objects and another for moving objects. Also, we have to define a constraint solver and a discrete dynamic world that controls the simulation of all rigid bodies using the above Ammo.js objects and algorithms. This dynamic world is used to append any rigid body, joint or collision object in our simulation process.

All the JavaScript objects created in the previous phase are then being used by Ammo.js classes for the construction of the physical geometry models. Ammo.js supports all the physical geometry models defined by the X3D specification, such as Box, Sphere, Cone, Cylinder and complex user-defined shapes. For the construction of the shapes, our JavaScript provides Ammo.js with constructors and their attributes, drawn during phase 2 from the RigidBody, RigidBodyCollection, CollisionCollection and CollidableShape elements.

All the objects that are being created at this phase and prior to appending them to our simulation world, follow the same methodology: We begin by defining a new transform object in Ammo.js, that stores the position and orientation coordinates corresponding to the X3D object's translation and rotation. Then, we create a new Ammo.js object based on the defined shape, and pass to it any relevant attributes, such as size and radius. For this new shape, we can set up margin options and calculate its local inertia. In order for the object to be part of our simulation, we have to create a Rigid Body Object, which will control the collision shape as a solid body. The rigid body is created by passing to a RigidBody constructor all the needed information, such as a motion state object based on a Transform object, a mass, the local inertia and the shape's physical object. At this point, we can configure our rigid body according to the attributes described in the RigidBody, RigidBodyCollection and CollisionCollection elements in our X3D scene, like angular and linear velocities, friction and restitution. All rigid bodies are added into the dynamic world for the simulation, but we also keep them in an Array so we can iterate over them and keep track of their position and rotation coordinates when updating the scene graph.

3.4 Constraining the Rigid Bodies

The X3D rigid body specification under component support level 2 defines multiple constraint types, like BallJoint, UniversalJoint, SliderJoint, SingleAxisHingeJoint, DoubleAxisHingeJoint and MotorJoint. In the description of phase two in Section 3.2, we mentioned that, if one or more joint type nodes exist, we keep track of them in a separate Array. After the completion of the third phase, the created objects can be constrained, either with the world or with one another. We will consecutively proceed to present the procedure followed to achieve this, but first we have to present the class types used in every case.

The BallJoint joint type is defines a point-to-point constraint, which takes two rigid bodies and constraints them along a common pivot point. The pivot point functions as the centre of a "ballsocket" in local space coordinates. For the UniversalJoint type we use the Ammo.js universal constraint, which defines two rotational degrees of freedom based on two axes defined within the UniversalJoint node, created to constrain the two rigid bodies. In the case of SliderJoint, we use the Ammo.js slider constraint, that constraints a rigid body to rotate and translate on a single axis in conjunction with another body. SingleAxisHingeJoints and DoubleAxisHingeJoints are created using the hinge constraint, which constrain two rigid bodies to act as a hinge -as the name implies. The only difference between them is that for the latter we use two rotational degrees of freedom while for the former we use only one. Lastly, in the MotorJoint type, we use the generic 6DoF constraint, which is a general type of constraint that can be used in most cases. This type of constraint can be used to lock or free any of the six degrees of freedom. By using this constraint we limit the rotation motor axis based on our X3D MotorJoint element.

In order to achieve this constraining of the rigid bodies we iterate over the joint array we created in a previous phase and based on the joint type we call the corresponding class from Ammo.js. The rigid bodies can be found by accessing the children elements and all extra attributes like anchor point, axis freedom and angle limit from the actual X3D joint type element. By iterating over our joint object list and then over our rigid body object array we can find the correct rigid bodies stored in the list compared with the RigidBody children nodes of any joint. Then the only thing left is to create the constraint and added in the simulation process by appending it in our dynamic world.

3.5 Realism and Interactivity

In this section, we will describe the functions that take place during the final phase, when all the rigid bodies have been created and constrained using the Bullet physics library API and appended into our X3D scene. We first create an update function that is called at every frame, using the X3DOM built-in function that is a common operation to the WebGL per-frame callback. In this function we define an internal simulation step that controls the flow of data that we pass from Ammo.js in each frame. This step simulation is used in order to iterate and update the position and rotation of all the rigid bodies of the scene prior to rendering, which in turn control a CollidableShape node. But this implementation is not only an animation output of the computed physics: we have also added to all the shapes that are not flagged as static, the appropriate event listeners that can be triggered by the user's mouse actions.

Mouse interactivity inside the X3DOM is now defined by the X3DOM.moveable component that defines an internal method of interactivity with the objects. That component is used in our work as a sensor for the interactive elements of the user activity. User interaction with the scenes rigid bodies is limited to grab, drag and release. DOM element changes are triggering X3DOM to redraw the scene, but for Bullet we have to manually recreate the scene or -in the case of grab, drag, release- a part of it. We iterate over the objects using the functions of phases 3 and 4, in order to reconstruct the rigid bodies that are being grabbed and moved along with their siblings in the case of joint types, using the mouse position to re-position and instantiate the same object inside the scene. Recreating an object and its constraints is not the optimal way as it leaves behind unused Ammo.js objects which have to be cleaned up in order to maintain the scenes frame rate in high complexity levels. Also, with this procedure we force Ammo.js to update the position and rotation of the rigid bodies of any joint system based on the Jacobian determinant function, while the user interacts with one of the bodies associated with it.

4 Experimental Evaluation

For the purposes of our work with X3DOM we have created various scenario scenes, in order to cover every possible rigid body state and all the constraining types. These examples were implemented in both HTML and XHTML file formats, and are available at our project website . For all the simulations and benchmark tests we were using the following browser versions: Google Chrome 33.0, Mozilla Firefox 29.0, Opera 20.0 and Maxthon 4.3.2. The

benchmark computers characteristics were: CPU: Inter Core i7-3630QM 2.40GHz, GPU: AMD Radeon HD 7600M Series 2GB, RAM: 4GB, OS: Windows 8.1. In our benchmark test, we have chosen to not include Microsoft Internet Explorer and Apple Safari browsers as they do not support WebGL render functionalities in X3DOM environments but rather fallback to the Flash renderer.

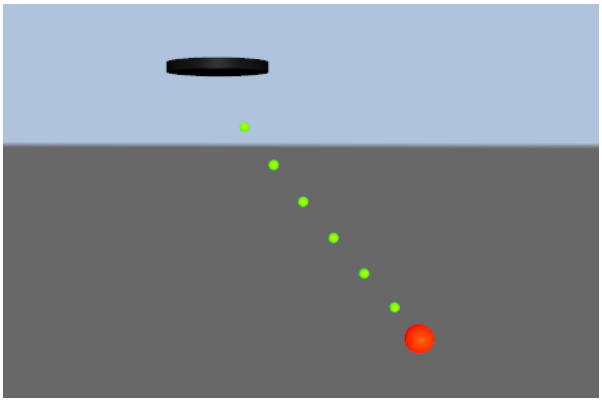

Figure 2: *The first example: a scene of a pendulum created using BallJoints.*

For our tests, we have created four X3D scenes, which include basic rigid body physics examples. The first scene, as seen in Figure 2, comprises of 8 rigid bodies, constrained with BallJoints in doublets in a way that they create a sort of chain. First we defined a starting point for this chain by pinning the first rigid body, with zero mass, to the world, which acts as a static Ammo.js object with no influence by other objects. All intermediate rigid bodies are set to ignore world gravity, have a mass of 0.1 kg each and are being influenced only by the last rigid body, which has a mass of 2 kg. The whole pendulum system is created not in an equilibrium position but rather with an amplitude, so it will start immediately simulating an oscillation after the page finishes loading. This example comprises of 8 RigidBody and 7 BallJoint elements defined in a RigidBodyCollection node. Also there are 8 CollidableShape nodes used by the RigidBody elements, which control the position and rotation of 8 transform elements in our scene that drive our final visual output.

Figure 3: *The second example: a scene of two colliding convex hull rigid bodies constrained using the BallJoint and MotorJoint type.*

The second example, Figure 3, used for our measurements is an extended version of the previous example. We have replaced the final sphere shape with an IndexedFaceSet, which was created in Ammo.js as s convex hull model and, using the inline method, we have imported an external X3D file into our scene node.

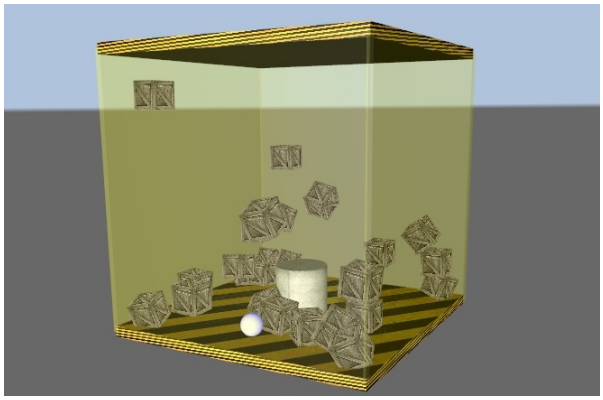

Figure 4: *The third example: a motor joint constrained sphere interacting with free rigid bodies dynamically created in real-time.*

That X3D file that we have loaded in our scene is composed of 6 rigid bodies, one of which defined as static, and used as the sibling rigid body for each of the four rigid bodies constrained by the BallJoint node type and the one with the MotorJoint. We also apply a torque along the Z axis in the last rigid body constrained by the MotorJoint, which is also an IndexedFaceSet object, which forces it to rotate around the static body and collide with the rest of the bodies. Inside the X3D file we have also defined the appropriate RigidBodyCollection and CollisionCollection nodes that complete the X3D physics scene.

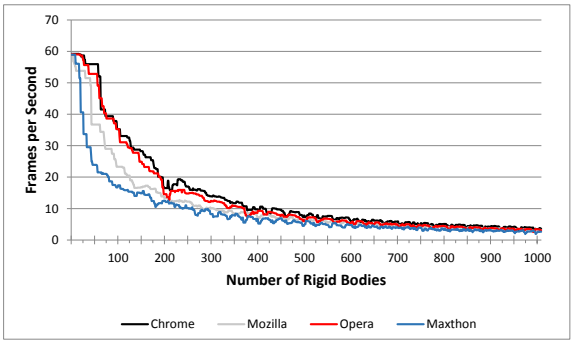

Figure 5: *A cross-browser line chart showing the frames per second in respect to the number of rigid bodies.*

A third example (Figure 4) was constructed in order to measure the fps rate with respect to the number of rigid bodies in a scene. For that case we created a scene with a confined space where a MotorJoint constraint controls a ball by applying, in every single frame, a torque along the Y axis. That confined space is constructed using 6 boxes which are placed in the right places in order to assemble the walls of our hollow box. We then dynamically append new boxes inside that area over time, to measure the browser capabilities in respect to frame rate.

For our benchmark we built a setup, where we create one rigid body at every frame in a random position inside the container, then pass it to the simulation process by adding it to the Ammo.js dynamic world and to the X3D scene by appending the appropriate node elements. We collected the frame rate at each time we added a body in our scene, a process followed for the 1000 bodies. We repeated the same simulations five times in each browser in order to get an average score between the measurements. Our benchmark

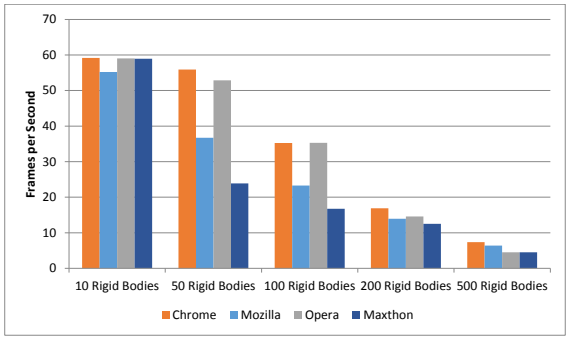

Figure 6: *A cross-browser column chart showing the frames per second in cases of 10, 50, 100, 200 and 500 rigid bodies.*

statistics are being visualized in Figure 5 and Figure 6, where we can clearly see that Google Chromes performance is better than that of every other browser we tested, closely followed by the Opera browser. Firefoxs performance seems to be significant lower than the previous two, but a lot better than Maxthons fps rate.

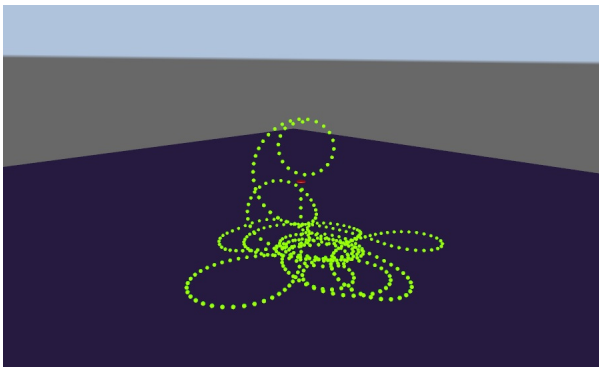

Figure 7: *The fourth example: dynamic real-time creation of constrained rigid bodies using single axis hinge joint.*

As soon as simulation starts, the two X3D scene systems, the one defined in our HTML page and the other inside the inline X3D start colliding. The results shown in Figure 5 were collected from 10 continuous simulations for every case and in each browser. Specifically, for the fps rate of each browser we measured the average frame rate for the first 2000 frames of each simulation in each test example scene and then we computed the average rate of all those simulations.

For our test we captured the average frame rate in each browser by simulating the scene in total 10 times. The first 5 were used in order to measure the X3DOM frame rate using its built-in function, whereas with the last 5 simulations we measured Ammos average frame rate, using the clock built-in function that returns the time in milliseconds. Figures 8 and 9 show the cross-browser fps rate in respect to the number of rigid bodies constrained with a joint. In Figure 10 we can see five snapshots of our captured data in a comparison of physics and render fps rate of the same scene as measured in our tests in the case of Chrome browser.

The final example that completes our benchmarking tests is comprised of a fully dynamic scene, starting only with two static rigid bodies, one for the ground representation and the other for the definition of the first link of a joint. In this test (Figure 7) we dynamically append rigid bodies in our scene and in our simulation, which

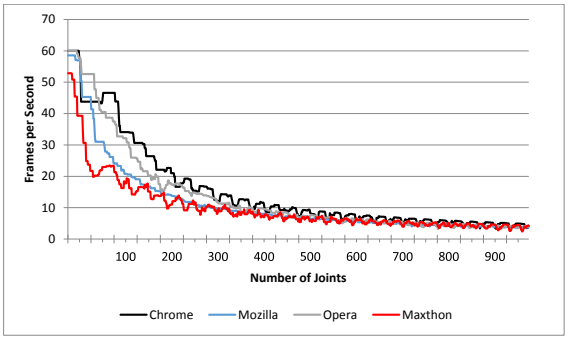

Figure 8: *A cross-browser line chart showing X3DOMs frame rate in respect to the number of joints.*

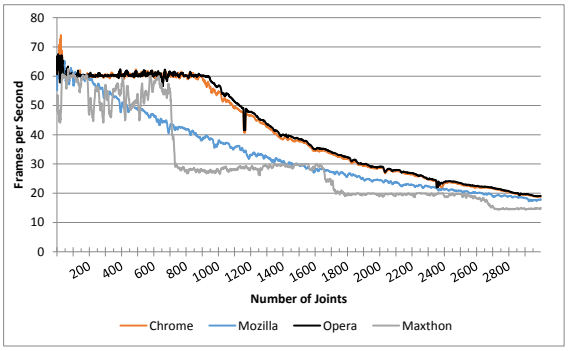

Figure 9: *A cross-browser line chart showing Ammos frame rate in respect to the number of joints.*

are first linked and constrained by a single axis hinge joint, limited to a Y-axis rotational freedom. Newly created rigid bodies are being constrained with the last object appended in the scene. Ammo.js now has to take into consideration the movement of every single object in the scene and uses the Jacobian determinant function to solve the differential equation systems near the equilibrium point. Rigid bodies are created in every frame and in parallel measure X3DOMs frame rate as well as the time (in milliseconds) needed by Ammo.js to calculate the physics. Those two measurements have been collected separately, as in the second case we turned off every visually rendered moving object, in order to measure the Ammo.js frame rate in every browser, using the same internal step for all the physics simulation.

Measurements show that there is no significant difference between the HTML and XHTML file formats regarding X3DOM's FPS rate (Figure 11). Furthermore in Figure 12 we observe a small delay in scene loading time in the case of XHTML files in Mozilla Firefox. With respect to the loading time, we should note that we had disabled the cache of each browser when our tests were measured.

5 Conclusions

WebGL has brought about a new fascinating world where interactive and realistic 3D applications for the web can be easily achieved. Although WebGL has been a big leap forward in 3D visualization, it is still in its early stages and there is a lot of work that has to be done in order to tackle many fundamental problems, from security issues to performance drawbacks. As technology advances more and more, desktop-like applications can be expected to become avail-

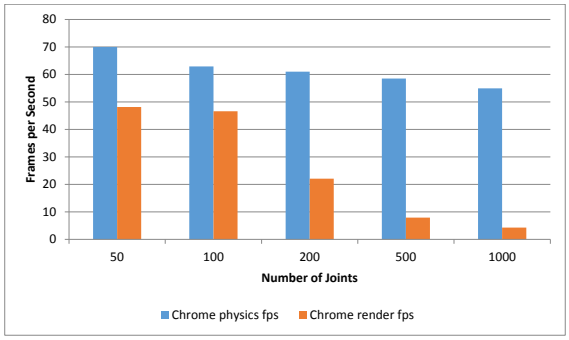

Figure 10: *Physics and render frame rate comparison in chrome browser in cases of 50, 100, 200, 500 and 1000 joints.*

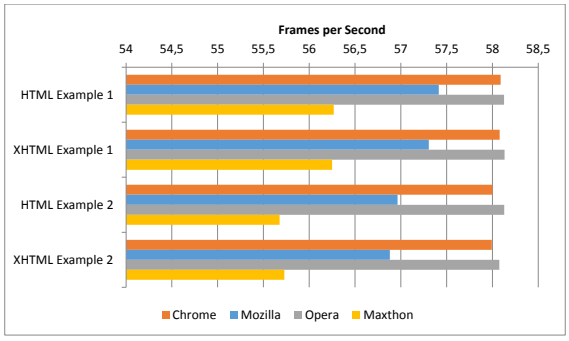

Figure 11: *A cross-browser X3DOM frame rate comparison in HTML and XHTML cases.*

able for the web. Our integration of an efficient and feature-rich JavaScript physics library named Ammo.js into the X3DOM framework, paves the way for X3DOM-based applications with high levels of interactivity, incorporating powerful real-time physics models that can lead to deeply immersive, real-time, interactive VR experiences for users, directly within web contexts, without the need for additional software besides a standard web browser. Our experiments showed us that real-time interactive X3DOM scenes with hundreds of rigid bodies can be constructed and function smoothly. Prior to our implementation, such a potentiality could only be an object of speculation: besides demonstrating its feasibility, we have also set up an experimental framework for evaluating different setups, platforms and scenarios, in terms of responsiveness and performance.

6 Future work

Our implementation of physics support for X3DOM aims in enabling the development of highly realistic Web-based simulations and interactive real-time 3D applications and games based on X3D technology. X3DOM physics support creates a new testing ground for several potential physics-driven systems. Based on our implementation, some of the future work that can be done could be the implementation of a physics-driven particle system based on the Particle Systems component of the extensible 3D (X3D) specification into X3DOM. Another potential research direction could be the creation of different physics sub-system inside X3DOM, that of a Soft Body simulation system. Such a system is already supported by Ammo.js: by registering new X3D(OM) nodes and attributes

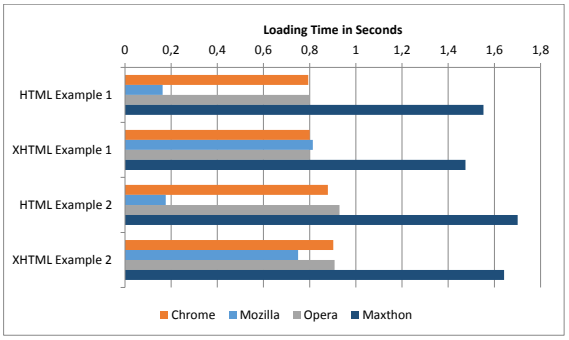

Figure 12: *A cross-browser X3D scene loading time comparison in HTML and XHTML cases.*

that describe deformable bodies, and in collaboration with our current work, we can create even more complex and realistic simulation applications. Finally, an implementation of Fluid Dynamics could also be incorporated, either based on the aforementioned Soft Body system, or as a separate autonomous system. In the meantime, however, we should keep in mind that the core of our engine is Bullet 2.82, which is written in C++, and the Ammo.js implementation is merely a JS port. To the extent that C++ remains the de facto language for such frameworks, for our code to be properly maintained, new Bullet versions (including the upcoming Bullet 3.0) will have to be systematically ported to JS. We can honestly expect, however, that in the future a pure, native JS physics engine will appear, as more and more technologies become web-oriented. In any case, X3DOM can now claim to feature a working real-time interactive physics component, which brings it yet another step closer to supporting fully immersive 3D games for the web.

Acknowledgements

The research of this paper is granted by the European Union and the Hellenic General Secretary of Research and Technology under the COOPERATION 2009 / 09SYN-72-956 Framework.

References

AMMO. Ammo.js home page. [accessed May 2014]. https://github.com/kripken/ammo.js.

BEHR, J., JUNG, Y., KEIL, J., DREVENSEK, T., ZÖLLNER, M., ESCHLER, P., AND FELLNER, D. W. 2010. A scalable architecture for the HTML5/X3D integration model X3DOM. In *Web3D*, ACM, D. G. Aliaga, M. M. Oliveira, A. Varshney, and C. Wyman, Eds., 185–194.

BOEING, A., AND BRUNL, T. 2007. Evaluation of real-time physics simulation systems. In *In Proceeding of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia. GRAPHITE '07*, ACM, 281–288.

BULLET. Real-Time Physics Simulation. Bullet Physics Library, 2013. [Online]. http://bulletphysics.org/wordpress/.

CANNON. Cannon.js home page. [accessed May 2014]. http://schteppe.github.io/cannon.js/.

CUBICVR. CubicVR home page. [accessed May 2014]. http://www.cubicvr.org/.

DALY, L., AND BRUTZMAN, D. 2007. X3D: Extensible 3D graphics standard (standards in a nutshell). *IEEE Signal Processing Magazine 24*, 6 (November), 130–135.

HAVOK. Havok. [accessed May 2014]. http://www.havok.com/.

JIGLIBJS2. JigLibJS2 home page. [accessed May 2014]. http://www.brokstuk.com/jiglibjs2.

KAPETANAKIS, K., ANDRIOTI, H., VONORTA, H., ZOTOS, M., TSIGKOS, N., AND PACHOULAKIS, I. 2013. Collaboration framework in the EViE-m platform. In *In Proceedings of the 24th European Association for Education in Electrical and Information Engineering, EAEEIE Annual Conference*, 178–183.

KAPETANAKIS, K., PANAGIOTAKIS, S., AND MALAMOS, A. G. 2013. HTML5 and websockets; challenges in network 3D collaboration. In *17th Panhellenic Conference on Informatics, PCI 2013, Thessaloniki, Greece - September 19 - 21, 2013*, ACM, P. H. Ketikidis, K. G. Margaritis, I. P. Vlahavas, A. Chatzigeorgiou, G. Eleftherakis, and I. Stamelos, Eds., 33–38.

MATSUBA, N. S., HUDSON, D. A., AND COUCH, J. 2005. The rigid body physics component: A proposed amendment to the x3d specification. In *In Proceeding ACM SIGGRAPH 2005 Web program. SIGGRAPH '05*, ACM.

NEWTON. Newton Game Dynamics. [accessed May 2014]. http://newtondynamics.com/forum/newton.php.

PHYSIJS. PhysiJS.js home page. [accessed May 2014]. http://chandlerprall.github.com/Physijs/.

PHYSIJS. Three.js home page. [accessed May 2014]. http://threejs.org/.

PHYSX. Nvidia PhysX, GeForce. [accessed May 2014]. http://www.geforce.com/hardware/technology/physx.

SMIT, R., 2007. Open Dynamics Engine. [accessed May 2014]. http://www.ode.org/.

WEBGL, 2014. Khronos Group, WebGL specification, editor's draft, March 2014. http://www.khronos.org/registry/webgl/specs/latest/1.0/.

X3D. ISO/IEC 19775:2004 Extensible 3D (X3D).

X3D. ISO/IEC 19776:2005 X3D encodings (XML and Classic VRML).

X3D. ISO/IEC FDIS 19777:2005 X3D language bindings (ECMAScript and Java).

XJ3D. Xj3D home page. [accessed May 2014]. http://www.xj3d.org/.

YOGYA, R., AND KOSALA, R. 2014. Comparison of physics frameworkds for webgl-based game engine. In *International Conference on Advances Science and Contemporary Engineering*.