Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS (2011)
J. Bender, K. Erleben, and E. Galin (Editors)

XML3D Physics: Declarative Physics Simulation for the Web

Kristian Sons[1] and Philipp Slusallek[1,2]

[1]DFKI, Saarbrücken, Germany
[2]Computergraphics Lab, Saarland University, Germany

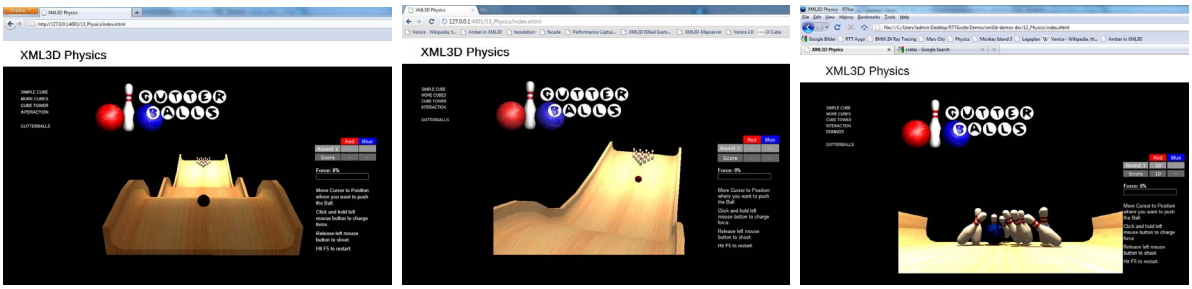

Figure 1: *Bowling game using XML3D Physics. From left to right: Firefox 5 & WebGL, modified Chromium & OpenGL, modified Firefox & RTFact ray tracer. Invisible geometry objects along the track generate events on collision with the ball, used to control the camera position.*

Abstract

As interactive 3D graphics has become an integral part of modern Web browsers via the low-level WebGL API [Khr11], it now becomes apparent that higher-level declarative approaches integrated with HTML5 are needed in order to make 3D graphics broadly and easily accessible to Web developers. A key component of interactive 3D scenes are physics simulations. However, specifying the physics properties required major changes to the scene graph in the past. In this paper, we present a declarative and orthogonal physics annotation framework that nicely separates the generic 3D scene description from its physics properties. The approach is based on the declarative XML3D format as an extension to HTML5, has been implemented as a plug-in that runs in three browsers that support XML3D and is demonstrated with a number of examples and performance evaluations.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques—Standards

1. Introduction

Interactive 3D graphics is becoming a commodity: Graphics hardware support has now been integrated onto all new desktop CPUs and even most mobile devices, most new TV-sets are expected to be 3D-stereo capable, and even mobile Internet speeds support larger 3D scenes and realtime remote interaction.

Still, 3D graphics remains a broadcast medium where the content is produced by only a few specialized companies (mostly for games). This is in stark contrast to the rest of the Internet, where user generated content (web sites, blogs, twitter, etc.) now plays a major role. One reason for that is the high complexity of authoring interactive 3D scenes. Another one is that there has been no widely accepted format for the distribution of high-fidelity interactive 3D content on the Web that tightly integrates with HTML. We are still missing the ability of users to *create, share, and experience* 3D content in the same way as the use of video exploded due to its integration into the Web by YouTube in 2005.

XML3D [SKR*10] has been proposed as such a format. It is a minimal and declarative 3D extension to HTML5

DOI: 10.2312/PE/vriphys/vriphys11/055-063

that provides new tags for describing geometry, shaders, lights, and other 3D scene objects based on a generic, extensible, and orthogonal framework. Wherever possible XML3D re-uses existing Web technology (DOM, CSS, Events, JavaScript, etc.) thus making it simple for Web developers to start using interactive 3D graphics fully integrated in their Web projects.

Physics simulations greatly improve the realism of and intuitive interaction with the scene and its use can simplify their design, e.g. of animations. However, in the past declarative physics specifications have been very invasive to the design of the 3D scene graph requiring the addition of many new nodes distributed across the entire scene graph. This would have been in clear conflict with the design goals of XML3D and would not scale to the addition of other simulations such as acoustics, haptics and others.

In this paper we propose XML3D Physics, a novel way to provide physics properties to existing scenes descriptions via orthogonal and optional physics annotations. These annotations are ignored by the visual rendering and are instead operated on by a separate physics engine that communicates with the rendering part through modifications of the scene graph (e.g. motion of objects, etc.). Through events, callbacks, and an additional API object, physically annotated objects can be directly manipulated and interacted with through JavaScript.

We aim to put Web developers to a position they are capable to use physics simulations without the need to cope with a physics engine API and its internals. Thus XML3D Physics was also designed with respect to this user group.

2. Related work

There are few portable 3D graphics formats that also contain physics information. The most widely-known is the open interchange format COLLADA [Khr08] from the Khronos Group. As of Version 1.4, physics support was added to the COLLADA standard. The COLLADA physics component includes the description of physical material and object parameters, proxy geometry for collision detection, constraints, and force fields. This functionality set was derived from the functionality of popular rigid body dynamics engines [CV07] and builds also the basis for the functionality of XML3D Physics.

As a main difference, in COLLADA the physics informations are kept inside a physics scene separate from the visual scene. The physics scene consists of rigid body instances. These reference to a rigid body object configuration, a physics material, and to the object that represents the rigid body in the visual scene. Connecting physical object parameters, material parameters, and a visual representation in a separate context makes sense for an interchange format where the main intention is to preserve as much information as possible.

For a web delivery format such as XML3D, this concept is way too complex and not intuitive. It would require the user to update all the references in the physics scene every time there is a structural change in the visual scene. Also, the inclusion of physics into COLLADA lead to a significant extension of the COLLADA format, something we wanted to prevent for XML3D.

X3D is an ISO standard XML-based file format for representing 3D computer graphics. In X3D, there is a Rigid Body Physics component [Web08] since Version 3.2. The feature set is similar to that of COLLADA. Like in COLLADA, physics parameters and objects are defined using a dedicated set of X3D nodes. Basis is the newly introduced non-hierarchical *RigidBodyCollection*. *RigidBody* nodes in this collection define the physics objects parameters. In contrast to COLLADA, the RigidBody objects do not reference their visual representation directly, but an *CollidableShape* node that wraps a single *Shape* node in the scene graph. This invasive design causes, that X3D Browser without support for the Rigid Body Physics component are not even able to display the visual part of the physics-enabled X3D scene.

To detect collisions, the author has to group objects of interest (again the *CollidableShapes*) into a *CollisionCollection*. Placing a *CollisionSensor* in the scene graph that references the CollisionCollection, allows listening to events generated when those objects collide. This design requires a whole range of new nodes, two collections alongside the actual scene graph, references between these collections and a special arrangement of the nodes in the scene.

Although it allows a granular configuration on object participation in collision detection and rigid body simulation, the complexity is unreasonably high and prevents non-experts from using it. This might be a reason why few X3D scenes using this component can be found on the web and none of the three major X3D browsers (BS Contact [Bit11], instantreality [Fra11] and Xj3D [Web11b]) are able to handle the basic example files [Web11a] provided by the Web3D Consortium. Instead, some vendors offer proprietary physics extensions [Bit08].

Alongside XML3D there is a second approach for declarative 3D within the DOM: X3DOM [BEJZ09] allows to integrate X3D nodes and concepts into HTML5. The functionality of X3DOM is restricted to the proposed HTML-Profile, which does not include the Rigid Body Physics component of X3D.

Few recent games can manage without physics simulated by some means or another. There are a number of commercial and non-commercial physics engines available: Havok Physics [Hav11], PhysX [NVI11], Bullet [C*11], and ODE [S*11], just to name a few. Additionally there are frameworks available, that abstract away implementation details of the various physics engines, provide a uniform API, and make implementations comparable [BB07]. Although

these API solutions are not directly related to our declarative approach, they give a good overview on the common functionality and parameter sets of recent physics engines.

Also most DCC tools offer functionality to rig graphical objects with physics parameters and run a physics simulation during rendering. Again, this demonstrates how helpful even simple rigid body physics and collision detection can be to allow authors the creation of realistic appearing virtual environments.

Extending a format by annotation is a common approach in the XML/XHTML world. One example is the annotation of semantic information to e.g. HTML5 using RDF [W3C04]. Consequently, RDF can also be applied to XML3D (as a HTML5 extension), as done in [KLN*10], where an AI engine is identifying semantic entities in the scene graph from the semantic annotation and applies an agent simulation on the XML3D world.

3. Design of XML3D Physics

The following sections explain the requirements for the XML3D Physics extension, give an overview of the syntax of the physics annotations, and show possibilities to interact with the physics objects.

3.1. Requirements

The design of the physics integration into XML3D was derived from following requirements:

- **Usability**. The usage of the physics component should be as simple and intuitive as possible to also enable Web developers and other non-graphics experts to use it in a HTML5/Web context.
- **Functionality**. We wanted to achieve functionality comparable to the physics components of COLLADA and X3D. This includes rigid body physics, constraints, and collision detection.
- **Orthogonality & Robustness**. The base XML3D specification should not be changed because of the physics extension. Also, the interpretation of physics related information should be independent from other simulations such as visual rendering. Adding physics information should not break rendering or other components in case the available runtime environment cannot deal with the physics information.

To fulfill the required functionality we had to provide a way to define physics object parameters (e.g. mass), physical material parameters (e.g. friction and restitution), intra-object constraints (e.g. hinge), global and local forces, and proxy geometry for collision detection. Additionally we wanted to provide a way to detect collisions between objects using DOM Events [W3C00]. It should also be possible for the user to easily detect collision with invisible objects. This functionality enables authors to detect if for instance an object penetrates a certain space in the scene.

3.2. Mark-up

The key design decision for XML3D Physics was to use annotations to enhance the scene with physics information. Firstly, this enables us to keep XML3D as lean as it is. There was no need to introduce a whole collection of new nodes, elements, and attributes in the base specification. Secondly it enables us to design XML3D Physics in an orthogonal way, keeping the physics simulation independent from other components such as the visual rendering and other simulations. The XHTML encoding of XML3D allowes us to use the XML namespace concept for these annotations. Thus they are not in the scope of the visual rendering, which just ignores the additional information.

A second major decision was to annotate the information in place where possible. We decided against having a second parallel graph (as in COLLADA) or collection (X3D). Also the references point in the opposite direction: Where references are needed e.g. for the reuse of physical materials, we reference the physics parameters from the object in the scene graph using CSS for instance. In contrast, COLLADA and X3D reference the visual representation from the physics scene. Our approach makes it much easier and more intuitive for the user to augment the 3D scene with application-relevant physics parameters and keeps the 3D scene as the main reference model tying together visual and other simulations.

The following section describes a selection of parameters and how they are connected to corresponding XML3D elements [SKR*10]. All line references refer to Listing 3.2. The physics parameters need to be defined in a dedicated namespace, identified with a specific URL identifier (Line 3). Typically the namespace is bound to a prefix. This prefix indicates the namespace for an element and all its children, and attributes.

An XML3D scene within the website is enclosed within an <xml3d> element. With respect to physics, it also starts a self-contained physics world. Here we can also define global physics parameters, for instance *gravity* (Line 4). It describes a global force that applies to all dynamic physics objects in the scene; a vector defines the magnitude and direction of this force.

In XML3D, visual surface parameters are described within a shader element. Shader are defined typically in the <defs> section of the XML3D scene, a concept borrowed from SVG [W3C09b], defining all reusable resources of a scene. In computer graphics, the description of surfaces has moved from a fixed to a programmable approach, where the surface appearance is computed in a short program code (the shader). This code can have nearly arbitrary parameters. XML3D has a generic parameter model for shaders as well as for meshes and light shaders. This parameter model allows to append a child node to the shader element for each parameter. The element type of the child node specifies the type of the parameter, the name attribute defines its name

and the text content specifies the value. Line 12 shows the definition of a float-triple with the name *diffuseColor*.

Material parameters for a physics simulation correspond very much to visual surface parameters for rendering. Thus we decided to use the same mark-up and introduced a <material> element for XML3D Physics. Lines 36-40 show the definition of a physics material and its parameter set. Additionally, the material defines the role of the object within the physics world. There are three kind of physical objects in the scene. Objects marked as *static* contribute as collision objects but are neither moved nor any force is applied on these objects. *Kinematic* objects are like static objects but can be moved within the physics world. Forces are applied to all objects that are marked as *dynamic*. Objects that do not have a physics material associated, are not taken into account by the physics simulation. Currently supported material parameters are friction, restitution, linear and angular damping.

The assignment of visual surface descriptions (shaders) to a group of meshes is done via CSS [W3C09a]. This concept has many advantages: CSS has a powerful selection mechanism and the rendering parameters are kept separate from the 3D structure. It's even possible to use different material descriptions depending on the target device using CSS media queries [W3C10]. XML3D also provides the possibility to assign surface description via attribute as fallback.

Again, we use the same mechanism for the assignment of physics materials. A physics material gets assigned to a group (see Line 48). The physics interpreter uses the referenced material for all children of the group as long as it's not overwritten by another assignment.

The generic parameter model can also be used to describe physics *object* parameters. If the user adds the *mass* parameter to a <mesh> element directly as shown in Line 24, it is not passed to the surface shader because it is defined in the physics namespace. The user can skip the prefix for the <float> element, if he wants to use the parameter in the surface shader. Another optional object parameter is the *centerOfMass* parameter. If not given, it is calculated from the geometry.

The definition of proxy geometries requires structured data, thus we introduced a <shape> element for XML3D Physics (see Line 25). Again, the generic parameter model is used to define the parameters of the proxy geometry. This way it is much easier to extend the number of supported proxy geometries. If no proxy geometry is given or is the given proxy type is not supported by the physics interpreter, the geometry as specified in the base XML3D structure should be used. If the physics interpreter detects more than one proxy geometry it uses the union of these geometries.

A somehow little more complex task is to describe constraints between objects. While the annotation of parameters and assignment of physics materials was straightforward, a constraint defines a relationships between objects that (in most cases) differs from the parent-child relationship within the DOM. If one considers for instance a hinge constraint, the author might not want to structure the scene so that the two parts of the hinge are children of the constraint but rather wants to follow some other application-specific logic. As a result, it is necessary to define the objects of a constraint using second-class references, references that are only known by the application logic and not by the DOM. If such a reference cannot be resolved or points to a unsupported element type, the reference is dangling and the constraint is not taken into account by the physics simulation. As common for XML3D, we use the URI syntax to describe references to other objects. Listing 3 shows

```
1  <xml3d style="width: 300px; height: 200px;"
2    xmlns="http://www.xml3d.org/2009/xml3d"
3    xmlns:physics="http://www.xml3d.org/2010/physics"
4    physics:gravity="0 -9.81 0">
5    <defs>
6      <transform id="boxTransform1"
7          translation="0.1 7 5"
8              rotation="0.5 0.9076 0.7066 0.9" />
9
10     <shader id="crateMat"
11       script="urn:xml3d:shader:phong">
12       <float3 name="diffuseColor">1.0 0.9 0.8</float3>
13       <float name="ambientIntensity">0.4</float>
14       <texture name="diffuseTexture" ...>
15         <img src="textures/crate.png" />
16       </texture>
17     </shader>
18     ...
19     <data id="box">
20       <int name="index">0 1 2 2 ...</int>
21       <float3 name="position">-1.0 1.0 1.0 ...</float3>
22       <float3 name="normal">0.0 0.0 -1.0 ...</float3>
23       <float2 name="texcoord">1.0 0.0 1.0 ...</float2>
24       <physics:float name="mass">10.0</physics:float>
25       <physics:shape>
26         <string name="type">box</string>
27         <float3 name="extends">2 2 2</float3>
28       </physics:shape>
29     </data>
30     <data id="ground">
31       <int name="index">0 1 2 2 3 0</int>
32       <float3 name="position">-50.0 -4.0 ...</float3>
33       <float3 name="normal">0.0 1.0 0.0 ...</float3>
34       <float2 name="texcoord">1.0 0.0 ...</float2>
35     </data>
36     <physics:material id="phCubemat">
37       <string name="type">dynamic</string>
38       <float name="friction">0.5</float>
39       <float name="restitution">0.2</float>
40     </physics:material>
41     <physics:material id="phGroundmat">
42       <string name="type">static</string>
43       <float name="friction">1.0</float>
44       <float name="restitution">1.0</float>
45     </physics:material>
46   </defs>
47   <group transform="#boxTransform1" shader="#crateMat"
48                        physics:material="#phCubemat">
49     <mesh type="triangles" src="#box" />
50   </group>
51   <group shader="#groundMat"
52   physics:material="#phGroundmat"
53   physics:oncollision="alert('Object dropped on the floor')">
54     <mesh type="triangles" src="#ground" />
55   </group>
56 </xml3d>
```

Figure 2: *An XML3D scene containing physics annotations.*

```
...
<physics:constraint objects="#box1 #box2">
  <string name="type">slider</string>
  <float2 name="linearXMinMax">4.0 8.0</float2>
  <float2 name="angularXMinMax">-0.1 1.5</float2>
</physics:constraint>
...
```

Figure 3: *Definition of a "'slider"' constraint between two objects in the scene.*

the definition of a constraint and the references to the involved objects. XML3D Physics currently supports following constraint types: *point2point*, *hinge*, *slider*, *conetwist*, *generic6dof*, and *generic6dofspring* [C*10].

3.3. Interaction

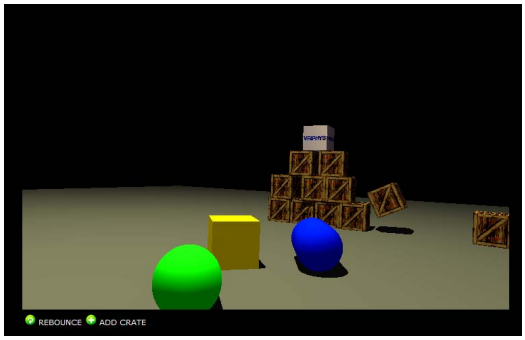

Figure 4: *Scene demonstrating the interaction capabilities of XML3D Physics. It includes pushing of objects (implemented as shown in Listing 5), several proxy geometries, and dynamic addition of new objects by clicking on a HTML button.*

There are two ways to interact with the physics scene: Events and JavaScript calls. Events can be thrown, when two physics objects collide. It is necessary to register a listener to the object of interest to receive those events. This is done exactly the same way as for other DOM events. We introduce a new event type with the string identifier *collision*. Then the user can either use the event attribute *oncollision* as shown in Line 53, or use the element's *addEventListener* method to register a JavaScript callback function. The received collision event contains information about the involved objects. XML3D geometry objects that have the *visibility* flag set to **false** are not considered during rendering, but they can still be used to detect collision events as explained above. This is very useful to detect when objects enter a certain area in a scene.

A second way to interact with the physics world is via JavaScript calls. Each object that is marked as dynamic implicitly has an additional property called *physics*. This property contains a *PhysicsObject* with an interface that allows to apply a force or an impulse to the center of the object or to a specific position. This makes it very simple for the user to interact with the physics engine. Listing 5 shows the few lines of code needed to push an object with a mouse click. Figure 4 shows a scene demonstrating the interaction capabilities of XML3D Physics.

```
<script type="text/javascript">
 function pushObject(event) {
  var po = event.target.physics;
  if (po) {
   po.applyImpulse(event.normal.scale(-20),
                   event.position);
 }
}
</script>
...
<mesh src="#box" onclick="pushObject(evt);"/>
```

Figure 5: *Applying an impulse to an object using the physics interface. The impulse goes along the negative normal at the position where the object was clicked.*

4. Implementation

There are three implementations of XML3D. First there are the ones natively implemented in C++: We have added XML3D to two of the major browsers, Firefox and Chromium, both supporting all XML3D features. Internally we use a separate data structure for the 3D scene graph [RGSS09] that uses typed data structures and wrappers to implement the DOM interfaces. The rendering of the XML3D scene happens asynchronously on this data structure and the result is embedded into the browser's compositing engine. Several renderers can be attached to the internal scene graph. There are currently two renderers available: a CPU ray-tracer and a hardware renderer based on OpenGL. A third portable implementation is done in JavaScript and uses WebGL for rendering. This version runs only on browsers supporting WebGL. All implementations are available on http://www.xml3d.org.

For prototyping we implemented the physics interpreter for XML3D as a plug-in. This decision was mainly based on the fact, that this way we forced ourselves to keep renderer and physics engine separate. The communication is only possible through the DOM and DOM-API calls. As a result, this guaranteed us an orthogonal design. Of course, the implementation could easily be integrated into the XML3D core of the browsers. As the physics engine we chose Bullet [C*11], because it was free, open-source, and known to support all needed features robustly.

There are several ways of communication between the DOM and the physics plug-in. In the initialization phase the plug-in sets up the physics simulation and creates all physics objects with references to their corresponding objects in the DOM. It parses all the required information through the standard DOM API. This includes all the attributes and elements

Scene	1 crate	10 crates	100 crates
Sim. time (ms)	474	521	1991
Pose updates (PU)	298	2523	80851
Bullet PU/s	63.1	484.3	4060.8
DOM v1 PU/s	63.0	482.6	630.6
DOM v2 PU/s	62.9	480.6	785.6
DOM v3 PU/s	62.9	480.2	1233.5
DOM v4 PU/s	62.9	481.5	2270.3

Table 1: *XML3D Physics performance in comparison to the raw physics simulation. The test scenes consist of 1, 10 and 100 falling boxes. The simulation steps, time and pose updates are measured until no further updates appeared due to convergence of the simulation.*

from the physics namespace as well as necessary data from the reference 3D scene, such as the structure of the tree, transformations, and geometry where no proxy geometry is given. The physics simulation can be explicitly started (and stopped) calling a corresponding JavaScript method on the <xml3d> element. As soon as the physics simulation has started, the calculation for the objects in the physics world are performed and changes were applied on the corresponding objects in the DOM. This involves mainly changes on the dynamic objects' transformations and firing of collision events. The plug-in monitors the scene in the DOM for:

- Parameter changes. These changes are then reflected in the physics world
- Transformation changes on those objects marked as kinematic or dynamic. Events generated by the plug-in itself are marked as such using a dynamic object property and then filtered out. The remaining foreign transformation events are either generated by user interaction or by other components and are applied on the objects in the physics world.
- Insertion or removal of DOM elements. It then adds or removes the corresponding physics objects if necessary.

The plug-in also adds the *physics* property to the JavaScript API of dynamic elements and thus allows to apply forces as explained above.

4.1. Performance

The major concern in using the HTML DOM as a physics scene graph together with the browser's plug-in API (NPAPI [Moz11]) was the overhead generated to update the DOM. The DOM was originally intended for XML data processing not as a base data structure for a runtime environment. The DOM is a very generic data structure, primary designed for string operations. Browser vendors recently improved the performance of DOM operations, but the basic problem still remains.

Performing the physics simulation, there are three threads involved: The physics simulation thread, the plug-in thread and the browser's main thread. The simulation thread runs the bullet engine's simulation loop in an arbitrary and configurable update frequency. The plug-in thread interfaces between the simulation and the browser. In the plug-in thread, the results of the simulation are collected, the inverse operation of flattening the scene graph structure is performed, and finally the DOM operations are called asynchronously. These calls are executed in the browser's main thread.

The NPAPI has a reflection model that allows to query object pointers for methods and property using names. This query in turn returns a pointer to the corresponding object. Functions can then be called with a list of variant objects as parameters. Ultimately, all calls need to be broken down to primitive data types or strings. This overhead combined with the overhead coming from the thread communication and from the browser's thread event queue is the overall overhead we expected when we decided for a plug-in prototype.

We tested four variants of passing the information to the DOM from simple pose updates. We measured the time needed until the information is available in the DOM and compared them to the raw output of the physics engine. For the tests, we run the simulation at ∼60Hz, resulting in a maximum of 60 pose updates per object per second. As it can be seen in Table 1, the overhead for small and medium scenes is negligible. As of more than 1000 updates per second, the DOM updates start lagging behind. We were able to improve this behavior comparing different kind of DOM assignments:

1. V1: Two *setAttribute* calls, one for rotation, one for translation. This method is a generic string based DOM setter method, thus the arguments need to be parsed after assignment.
2. V2: Gathering of all assignment calls in one large string and execute those in JavaScript. Each assignment consists of a *getElementById* call to get the targeted element in the DOM and the two *setAttribute* calls as explained in V1.
3. V3: The XML3D specification defines *rotation* and *translation* properties of the <transform> element. These are of type *XML3DRotation* and *XML3DVec3*, which exist as native or JavaScript objects in the browser depending on the XML3D implementation. It's possible to create new instances of these types from within the plug-in. Then one DOM call is needed to assign those objects to the rotation property and translation property respectively.
4. V4: We used a *setPose(translation, rotation)* call to assign rotation and translation in a single step. As this method is not defined in the XML3D specification we simulated it in JavaScript. Just as in V3, we used native or JavaScript objects as parameters for the function.

As it can be seen from Table 1, the performance varies depending on the type of DOM assignment. It can be said, that an asynchronous DOM call is expensive and should be avoided. The JavaScript engine seems to have better access

to the DOM: In V2 the instructions need to be parsed before execution and three DOM-API calls are performed. Nevertheless, this solution is faster than calling just two methods from within the plug-in. We propose the introduction of a setPose method to the XML3D <transform> element, that allows setting of translation and rotation using a vector and a rotation object in one single step. Using this method we reach up to 2300 DOM updates per second, which seems to be the limit for the browser frameworks with DOM updates running in a single thread. The simulation produces approximately 4000 updates per second running at 60 simulation steps per second with 100 objects and lots of intra-object collisions. Figure 6 shows the rendering of a similar scene as used for the performance tests, other scenes provide similar results.

From our observations above, we did a comparison using just the JavaScript engine for pose updates. Using the same kind of assignment as in V2, we can perform 12500 pose update per seconds. Using XML3DRotation and XML3DVec objects for the assignment as done in V3, we can update the pose even 54975 times per seconds. These results from these tests show clearly, that the current bottleneck is the asynchronous DOM update from within the plug-in. Avoiding the NPAPI interface to access the DOM would gain a significant performance improvement.

The performance we measured for the XML3D Physics plug-in is much lower (∼56%) than the fully native implementation (e.g. on the GPU). Nevertheless, the implementation is fast enough for a first revision that has to deal with the overhead coming from the plug-in API.

5. Conclusions and Future Work

In this paper we presented XML3D Physics, a novel approach for rigid body physics and collision detection support within a declarative DOM-based interactive 3D graphics format on the Web. We added the physics informations using annotations instead of adding extra nodes to the XML3D specification. Thus we proposed a fully orthogonal design based format, that for the first time allows to keep the visual rendering and physics simulation independent and exchangeable. We demonstrated the orthogonality of our approach with a physics simulation plug-in that works for all XML3D implementations, the two native XML3D browsers and the WebGL/JavaScript renderer.

The physics annotations can be easily added by hand or – more convenient – through our XML3D Blender exporter that we extended for that purpose. This way it is easy to configure the geometry and appearance as well as the physical parameters within Blender, export the result as XML3D embedded in a web page, and then to view the result immediately in the browser. The physics simulation combined with the JavaScript API and the collision events greatly simplifies the creation of interactive 3D web applications. It enabled bachelor students with basic XML3D knowledge to create an interactive 3D bowling game within two days, including modeling, game logic development, and design of the webpage (s. Figure 1).

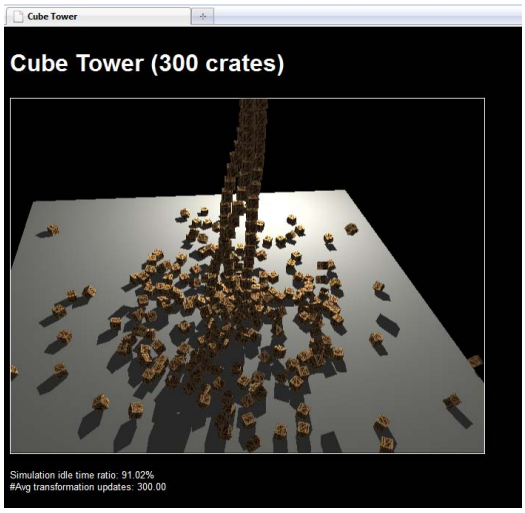

Figure 6: *Performance test scene. 300 crates in a colliding tower, rendered using real-time ray tracing.*

Within the enlightenment project [Zak11], an extension to the the compiler framework LLVM was developed, that allows to compile from C to JavaScript. As an example, the Bullet physics engine was compiled to JavaScript. We would like to integrate this version of Bullet as it could prove that also the physics simulation is exchangeable and would allow performance comparison between the native plug-in and JS interpreted version. Finally, a combination of WebGL/XML3D and Bullet in JavaScript would allow the worldwide exchange of 3D graphics with rigid body physics simulation without the need of a specific program installed other than a standard browser.

The approach we used for physics annotation is applicable to a number of other simulation domains, such as sound rendering and haptics by simply adding other annotations to the scene description. Some of them might also simply use generic semantic annotations via RDF for their purpose (such as identifying cars in a scene). In general, XML3D can be used as the central 3D model reference, while simulations monitor the scene and expose their results to the scene. The generality of the DOM, the fast scripting engines and the ubiquity of browsers (also on mobile devices) give good reason to run such simulations in the browser context for making them broadly available.

From our measurements we see, that the system is running reasonable fast for small and medium sized scenes. We found that the communication between browser and plug-in is not yet fast enough to fully reach the performance level

of native physics engines. In this first revision we preferred the guaranteed orthogonality over performance. But we intend to integrate the physics component into the native implementations with its optimized data structure. This will bypass the communication overhead and DOM issues and thus will allow us to fully exploit the performance of the physics engine.

However, as the Internet has moved to an area of user generated content, we have to expect arbitrary content, rather than hand optimized scenes. Thus we want to investigate more in the optimization of browser frameworks and their underlying data structures and runtime models for 3D graphics, simulations and other domains dealing with large data sets, and high data transmission and update rates.

We would also like to integrate other aspect of physics such as soft body physics and fluid dynamics. These aspects require frequent and computationally expensive geometry processing. In physics and game engine, these operations are typically optimized for a multi- and many-core architectures on CPUs or GPUs. Thus a pure DOM-based solution is not applicable. But as dynamic meshes are required in many applications, we developed XFlow as an extension to XML3D. This system allows the declarative description of a flow graph and a novel programming abstraction. It allows scripting of flow graph node operations and compiles optimized code for CPUs and GPUs. The publication of this system is pending.

We started to standardize our XML3D ideas within a W3C Community (formerly Incubator) Group. In this group we set amongst others a focus on usability and content creation. Physics could eventually become a part of these efforts.

5.1. Acknowledgment

We would like to thank Daniel Hofmann and Pascal Liedtke for the implementation of the first prototype version.

References

[BB07] BOEING A., BRÄUNL T.: Evaluation of real-time physics simulation systems. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia* (New York, NY, USA, 2007), GRAPHITE '07, ACM, pp. 281–288. 2

[BEJZ09] BEHR J., ESCHLER P., JUNG Y., ZÖLLNER M.: X3DOM: a DOM-based HTML5/X3D integration model. In *Proceedings of the 14th International Conference on 3D Web Technology* (New York, NY, USA, 2009), Web3D '09, ACM, pp. 127–135. 2

[Bit08] BITMANAGEMENT: BS Contact Physics Support, 2008. http://www.bitmanagement.com/developer/contact/relnotes71.html#physics. 2

[Bit11] BITMANAGEMENT: BS Contact, 2011. http://www.bitmanagement.com/en/products/interactive-3d-clients/bs-contact. 2

[C*10] COUMANS E., ET AL.: Bullet Constraints, 2010. http://www.bulletphysics.org/mediawiki-1.5.8/index.php/Constraints. 5

[C*11] COUMANS E., ET AL.: Bullet Physics Library, 2011. BulletPhysicsLibrary.http://bullet.sourceforge.net/. 2, 5

[CV07] COUMANS E., VICTOR K.: Collada physics. In *Proceedings of the twelfth international conference on 3D web technology* (New York, NY, USA, 2007), Web3D '07, ACM, pp. 101–104. 2

[Fra11] FRAUNHOFER IGD: instantreality, 2011. http://www.instantreality.org/. 2

[Hav11] HAVOK: Havok Physics, 2011. http://www.havok.com/index.php?page=havok-physics. 2

[Khr08] KHRONOS: COLLADA - Digital Asset Exchange Schema for Interactive 3D, 2008. http://www.khronos.org/collada/. 2

[Khr11] KHRONOS: WebGL Specification Version 1.0, 2011. https://www.khronos.org/registry/webgl/specs/1.0/. 1

[KLN*10] KAPAHNKE P., LIEDTKE P., NESBIGALL S., WARWAS S., KLUSCH M.: ISReal: An Open Platform for Semantic-Based 3D Simulations in the 3D Internet. In *The Semantic Web Ű- ISWC 2010: 9th International Semantic Web Conference, Shanghai, China, Revised Selected Papers, Part II*. 2010, pp. 161–176. 3

[Moz11] MOZILLA: Gecko Plugin API Reference, 2011. https://developer.mozilla.org/en/Gecko_Plugin_API_Reference. 6

[NVI11] NVIDIA: NVIDIA Physx, 2011. http://www.nvidia.com/object/physx_new.html. 2

[RGSS09] RUBINSTEIN D., GEORGIEV I., SCHUG B., SLUSALLEK P.: RTSG: Ray Tracing for X3D via a Flexible Rendering Framework. In *Proceedings of the 14th International Conference on Web3D Technology 2009 (Web3D Symposium '09)* (New York, NY, USA, 2009), ACM, pp. 43–50. 5

[S*11] SMITH R., ET AL.: Open Dynamics Engine, 2011. http://www.ode.org. 2

[SKR*10] SONS K., KLEIN F., RUBINSTEIN D., BYELOZYOROV S., SLUSALLEK P.: XML3D: Interactive 3D Graphics for the Web. In *Proceedings of the 15th International Conference on Web 3D Technology* (New York, NY, USA, 2010), Web3D '10, ACM, pp. 175–184. 1, 3

[W3C00] W3C: Document object model level 2 events specification, 2000. http://www.w3.org/TR/DOM-Level-2-Events/. 3

[W3C04] W3C: RDF/XML Syntax Specification, 2004. http://www.w3.org/TR/REC-rdf-syntax/. 3

[W3C09a] W3C: Cascading Style Sheets, 2009. http://www.w3.org/Style/CSS/. 4

[W3C09b] W3C: Scalable Vector Graphics, 2009. http://www.w3.org/Graphics/SVG/. 3

[W3C10] W3C: Media Queries, 2010. http://www.w3.org/TR/css3-mediaqueries/. 4

[Web08] WEB3D CONSORTIUM: ISO/IEC 19775-1.2:2008, 37 Rigid body physics component, 2008. http://web3d.org/x3d/specifications/ISO-IEC-19775-1.2-X3D-AbstractSpecification/Part01/components/rigid_physics.html. 2

[Web11a] WEB3D CONSORTIUM: Rigid Body Physics Example Files, 2011. http://www.web3d.org/x3d/content/examples/Basic/RigidBodyPhysics/. 2

[Web11b] WEB3D CONSORTIUM: Xj3D, 2011. http://www.xj3d.org/. 2

[Zak11] ZAKAI A.: Emscripten 1.4, 2011. https://github.com/kripken/emscripten/wiki. 7