A Scalable Architecture for the HTML5/ X3D Integration Model X3DOM

J. Behr* Y. Jung† J. Keil T. Drevensek M. Zoellner P. Eschler D. Fellner

Fraunhofer IGD / TU Darmstadt, Darmstadt, Germany

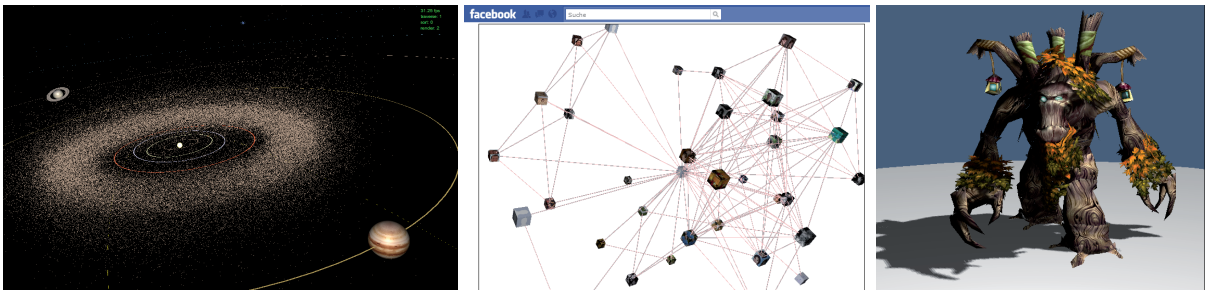

Figure 1: *Screenshots showing third party applications realized with X3DOM: Simulation of the planets and 100000 of the known 480000 asteroids of the Solar System (left), 3D visualization of social networks (middle), an animated WoW character with dynamic shadows (right).*

Abstract

We present a scalable architecture, which implements and further evolves the HTML/X3D integration model X3DOM introduced in [Behr et al. 2009]. The goal of this model is to integrate and update declarative X3D content directly in the HTML DOM tree. The model was previously presented in a very abstract and generic way by only suggesting implementation strategies. The available open-source *x3dom.js* architecture provides concrete solutions to the previously open points and extents the generic model if necessary. The outstanding feature of the architecture is to provide a single declarative interface to application developers and at the same time support of various backends through a powerful fallback-model. This fallback-model does not provide a single implementation strategy for the runtime and rendering module but supports different methods transparently. This includes native browser implementations and X3D-plugins as well as a WebGL-based scene-graph, which allows running the content without the need for installing additional plugins on all browsers that support WebGL. The paper furthermore discusses generic aspects of the architecture like encoding and introspection, but also provides details concerning two backends. It shows how the system interfaces with X3D-plugins and WebGL and also discusses implementation specific features and limitations.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality I.3.6 [Methodology and Techniques]: Standards—Languages

Keywords: X3D, HTML5, WebGL, DOM, Web integration

1 Introduction

There have been a wide number of approaches over the last 15 years to integrate 3D technologies in web browsers. Most of these systems and standards disappeared, some survived, but none has a prevalence rate which makes it relevant and known to most people who use the web daily. Real-time 3D is still an exception and only used in very specific domains. At last Web3D it was discussed [Behr et al. 2009], why these different approaches failed. The authors presented a solution, called X3DOM, to overcome this situation by making declarative 3D a first class citizen (like text, video and sound) of every user agent (UA, e.g. web browser). The proposed model allows to integrate X3D [Web3D 2008] content directly into the HTML DOM-tree. The authors proposed using X3D since the current HTML5 spec already references X3D for declarative 3D scenes with one line of text [W3C 2009a] in section 12.2:

> *Embedding 3D imagery into XHTML documents is the domain of X3D, or technologies based on X3D that are namespace-aware.*

This specification does not yet define how the integration actually should look like, how scene-updates and events should be managed and it also does not define any kind of media-integration aspect. Hence, in [Behr et al. 2009] various designs and possible solutions were discussed and an integration model called X3DOM was developed. This model was presented to the Web3D community and heavily discussed. It was accepted as one of the official solutions supported in the W3C/Web3D Working group. The X3D/HTML5 Working group [Consortium 2010] is contributing to the general HTML Working Group for the purpose of best integrating X3D with HTML. The goal is to make the native authoring and use of declarative XML-based X3D scenes as natural and well-supported for HTML5 authors as the support provided for SVG [W3C 2009e] or Mathematical Markup Language (MathML) [W3C 2009c].

The proposed framework was build to support the ongoing discussion in the Working Group and overall X3D/HTML community. It is an intermediate solution until X3D becomes an integral part of HTML. Hence, with this integration model we hope to trigger a process similar to how the SVG in HTML5 integration evolved:

- Provide a vision and runtime to experiment with and also develop an integration model for declarative 3D in HTML5.
- Get the discussion in the HTML5 and X3D communities going and evolve the system and integration model.
- Finally it would be part of the HTML5 standard and supported by every major browser natively.

In this paper we present the current state of the framework implementation, which allows studying and progressing the integration

*e-mail:johannes.behr@igd.fraunhofer.de

†e-mail:yvonne.jung@igd.fraunhofer.de

model today without modifying a web browser's actual code base.

2 Related Work

Having genuine 3D inside web pages still is limited today. However, there are lots of free or commercial third party browser tools or plugins available. A comprehensive overview of the current 3D web-technology can be found in [Behr et al. 2009]. This overview includes plugin technology (e.g. X3D, Java3D, Flash) and pluginless technologies like Canvas3D and 3D on 2D-pipeline systems using SVG and canvas rendering. The only really new technology is the WebGL API, which is a successor of Canvas3D [Vukicevic 2009] and embedded into JavaScript that originally was designed to create dynamic web pages. Hence, in the following we will especially focus on recent developments concerning WebGL.

WebGL describes an additional 3D rendering context for the HTML5 canvas element [W3C 2009b] by exposing the rendering API via new JavaScript objects and methods. A working draft of the specification is available online [Khronos 2010]. Furthermore, additional information, tutorials and programming examples e.g. can be found in the official WebGL Wiki [Khronos 2009] or in the "Learning WebGL" blog [Thomas 2010]. WebGL is based on the OpenGL ES 2.0 standard [Munshi et al. 2009], an OpenGL dialect that was developed for embedded and portable devices such as mobile phones with less powerful graphics chips. In contrast to standard desktop OpenGL [Shreiner et al. 2006] it has no support for the old fixed function pipeline but is completely shader-based. Thereby it is comparable to the newer OpenGL 3.x standard with the exception that advanced features like transform feedback or geometry shaders that require very recent GPUs are not yet supported.

By utilizing ES 2.0 as basis, it was possible to define the WebGL specification in a platform independent manner as on the one hand OpenGL 2.1 (the current standard for desktop machines) is a superset of ES 2.0 and on the other hand, most recent smartphones already have chips being conformant to that standard. Hence, at the moment there are ongoing efforts for porting the mobile versions of various web browsers, including WebGL, also to the common platforms for mobile phones like Maemo 5 (Nokia N900), Apple's iPhone or Google Android. Already in late September 2009 the first WebGL implementation was available with a Mozilla Firefox 3.7 pre-alpha build. Since then, most other browsers like Apple WebKit, Google Chrome and Opera except IE followed with WebGL-enabled developer builds, and the capabilities very soon converged.

Also, WebGL-based utility libraries like WebGLU [DeLillo 2009] emerged, which mimics the old OpenGL fixed function pipeline by providing concepts like the matrix stack, and rendering frameworks building on top of WebGL by providing a JavaScript-based API, but none connects the HTML DOM-tree to the 3D content. XML3D [Sons 2010] is another system, which integrates 3D graphics into the HTML DOM but until now there is unfortunately no published tag-set, system architecture or publication available.

For instance GLGE [Brunt 2010] is a JavaScript scene-graph system on top of WebGL, which masks the low-level graphics API calls of WebGL by providing a procedural programming interface. The library is comparable with typical graphics engines on the one hand and on the other hand to other JavaScript libraries like jQuery[1], which aims at simplifying HTML document traversing, event handling, and Ajax interactions. 3D models in GLGE are represented either with a special GLGE format or by using Collada [Arnaud and Barnes 2006], an XML-based file format for representing 3D assets, which in contrast to X3D is only used for data import and export. Likewise SpiderGL [Benedetto 2010] provides

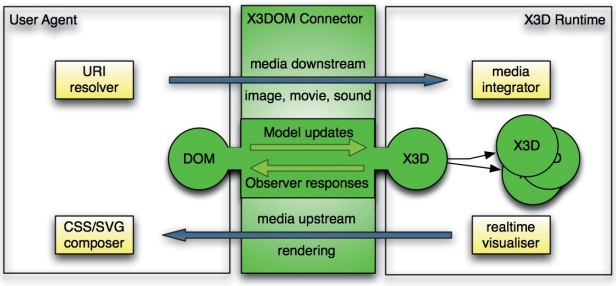

Figure 2: *Proposed X3DOM system architecture including the UA (i.e. web browser), the Web3D runtime and the X3DOM connector.*

algorithms for 3D graphics, but on a lower level of abstraction and without special structures like the scene-graph. Another JavaScript library that also facilitates Collada for loading is SceneJS [Kay 2010], which is based on functional programming that, according to [Crockford 2008], is a programming paradigm natural to JavaScript. CopperLicht [Ambiera 2010] provides a 3D world editor and promises to ease browser-based 3D game development.

To avoid embedding large models into the web page, most libs make use of an XMLHttpRequest (XHR), which is a DOM API "that provides scripted client functionality for transferring data between a client and a server"[2]. Because XHR allows asynchronous data requests without page reload, big meshes with lots of data can be loaded in the background without stalling the complete application, which is e.g. important for low-speed internet connections.

3 System Architecture

As mentioned, the presented open source framework and runtime was built to support the ongoing discussion in the Web3D and W3C communities how an integration of HTML5 and declarative 3D content could look like. It tries to fulfill the current HTML5 specification for declarative 3D content and allows including X3D elements as part of any HTML5 DOM tree.

The goal here is to have a *live* X3D scene in the HTML DOM, which allows the application developer to manipulate the 3D content by only adding, removing or changing DOM elements. No specific plugins or plugin interfaces like the SAI [Web3DConsortium 2009] are needed. It also supports some of the HTML events, like "onclick" on 3D objects. The basic architecture extends the system proposed in [Behr et al. 2009]. Its extension building blocks are the UA, X3D runtime and the connector.

User Agent Holds the DOM tree, integrates and composes the final rendering and provides some form of uri-resolver to download image, movie, sound and scene content.

X3D runtime Provides services to build and update the X3D scene. It renders the scene on any change and handles user inputs on navigation and picking.

Connector Builds the inner core of the architecture. It connects the DOM tree with the X3D runtime. The most important functionality is to distribute relevant changes in both directions, i.e. all DOM tree updates (e.g. adding, removing nodes or changing attributes) and observer responses to trigger DHTML-style events from user inputs on the 3D content (e.g. picking to onclick events). It also has to handle any media up- and downstream: scene visualization back from the X3D runtime and image, movie and sound downstreams.

[1] http://jquery.com/

[2] http://www.w3.org/TR/XMLHttpRequest/

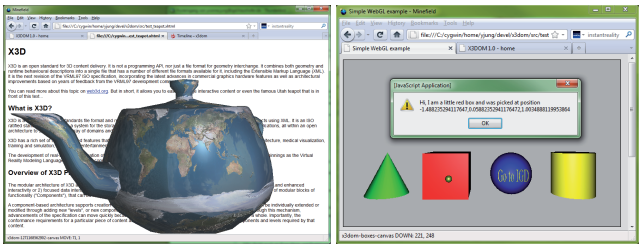

Figure 3: *Examples of tight HTML integration: a 3D object in front of HTML text via simple CSS styling of the X3D element (left), and mouse interaction (right); after clicking on an object a message box pops up and the small ball is repositioned to the last pick position.*

The general architecture (cp. Figure 2) could be implemented in various ways. The first idea was to extent an open-source UA codebase to demonstrate and evaluate the proposed HTML tags. Such an extension was drafted and proposed in [Behr et al. 2009]. The current implementation avoids any native extensions to UA codebases by providing a JavaScript-layer that automatically monitors the DOM-changes and updates an X3D runtime and rendering system in the background. The approach utilizes the fact that modern UA parse and append any tags from a (X)HTML stream to the DOM even thought they are not part of the current standard. The system supports thereby not a single X3D environment but uses the later on presented fallback-model to pick the best one based on the UA setup and content profile.

This intermediate framework allows developing the integration model further and supports a smooth transition from the current situation until all web browsers support X3D. Thereby, it even considers native browser support. The presented system allows the application developer to insert X3D sections in the HTML or XHTML page (which will be shown later) and only needs to add one extra line of script code that initiates the runtime automatically:

```
<script type="text/javascript" src="x3dom.js"></script>
```

There is therefore no installation process and setup method required. Similarly to e.g. jQuery, the general architecture is deployed as part of the web application, which utilizes the framework for rendering. The architecture tries to run with the current UA-setup and with the help of 3D APIs, e.g. WebGL, this works without having any plugin installed. The system gives the application developer the freedom to deliver one of the released versions with fix url addresses or content specific variations. This framework thereby allows web developers to use X3D today as part of their application. They can utilize this flexible layer on the one hand but also build tight integrated content currently not available with standard plugins on the other hand. An example of the tight HTML integration is shown in Figure 3, where the famous Utah teapot appears in front of HTML text, which is achieved via simple CSS styling of our proposed `<X3D>` element. Furthermore, the image to the right shows HTML-like interaction with 3D objects via the ”onclick” event.

3.1 Profile

The system can handle any X3D profile as long as there is a backend available supporting the requested profile. There is not a single profile which must be supported by every backend but we strongly suggest that every backend support the HTML-Profile proposed in [Behr et al. 2009]. The profile extends the X3D Interchange-Profile with ”Inline” and ”Shader” nodes but excludes ”Script” nodes and protoypes. We further extend the profile with two new nodes, which are the result from a discussion on the WebGL email list[3]. They are mainly included to support the WebSG proposal and further ease the integration of HTML/X3D.

First of all we propose a new generic `<Texture>` node derived from ”X3DTextureNode”, which can hold a single `<img>`, `<video>` or `<canvas>` as child. This allows using these HTML tags within the X3D content. The second node is a generic single-index `<Mesh>` class without any specific ’coord’, ’normal’, ’color’ or ’texCoord’ fields. This node, without any vertex attribute semantics, is especially useful for modern shader-based content and can be directly mapped to GPU APIs (e.g. WebGL). In the near future we may split the HTML-Profile, as result of the aforementioned WebSG discussion, to build a minimal shader-based sub-profile, which further eases the implementation effort.

3.2 Fallback Model

The architecture provides a single and stable DOM-based interface for application developers, which is based on the encoding and profile outlined in [Behr et al. 2009]. The proposed JavaScript layer, as connector and synchronizer, monitors the DOM-changes and updates the relative X3D structures and vice versa. The system supports not a single X3D runtime but is able to instantiate and synchronize different backend technologies following the fallback model illustrated in Figure 4. Every supported backend provides support for specific X3D profiles and has specific feature and performance criteria. The framework checks the requested profile, as part of the 3D-content, and tries to detect and initiate a matching backend following the fallback model. The performance and feature decreases with every alternative but it is more likely to find the supporting APIs in subsequent backends.

The proposed system supports native implementations, X3D/SAI-plugins [Web3DConsortium 2009] and WebGL [Khronos 2010] as backend. O3D [Google 2009] and Flash [Adobe 2010] should also be supported in the future but they have a lower priority. The original O3D plugin presumably should render faster, but now that O3D switched to WebGL there is probably no advantage any more. Flash has a very wide installation base but does not support hardware accelerated 3D and is therefore much worse performance wise.

3.2.1 Scalability with X3D

The proposed framework monitors the DOM and deploys for every detected `<X3D>` tag the fallback model. The user has no further control, besides the content profile, to pick a specific backend or X3D runtime. There is no backend-specific interface or functionality reflected to the DOM node. This allows the framework to incorporate additional backends in the future. The following features are supported for all backends:

- `<x3d>` tag as root
- full X3D runtime including routes
- declarative material system and shader
- generic Texture node but also explicit ImageTexture etc.
- single-/ multi-index IndexedFaceSet (may need extra copy)
- additional X3D nodes (TimeSensor, Interpolator, Inline,...)
- no scripting and prototypes

3.2.2 Context and Runtime Access with WebSG

The Khronos Group is eager to develop WebGL as 3D-API standard and every major browser, besides Internet Explorer, will support

[3] http://www.khronos.org/webgl/public-mailing-list/archives/1003/msg00134.html

it in the next major release. Therefore every browser, supporting WebGL, will be able to create and return a WebGL context to the application developer. The question emerged, if X3DOM, as next declarative integration model could also provide access to the WebGL context and API. As result of this discussion we introduced the WebSceneGraph concept, called WebSG, which builds on the current X3DOM architecture and fallback-model, but does provide interfaces to access the native WebGL context. This allows mixing the declarative nature of X3D with the API approach of WebGL.

Thereby the system supports a specific behavior when used with a `<WebSG>` root tag instead of the `<X3D>` tag. The fallback-model selects a backend, which provides the WebGL context and appends functions to the DOM element to access the context. Additional functions are provided to control the render update mechanism and event flow. The WebSG environment still supports the full HTML-Profile proposed, but as mentioned this may change in the future. We currently investigate how a minimal profile could look like. This discussion will probably result in the following setup, which is only a very first draft, reflects the current development state and shows a possible direction for further evaluation:

- `<websg>` tag as root
- minimal runtime, no routes
- only explicit shader based material
- `<texture>` uses `<img>`, `<video>` or `<canvas>` tags
- single-index `<mesh>` node, parses to WebGLArray/ VBO
- always WebGL based (*websg.context* holds the context and *websg.render()* renders the scene to the WebGL context)
- `<transform>` to build the hierarchy
- all additional X3D nodes optional
- only id/USE

3.3 Encoding

The X3D ISO standard [Web3D 2008] provides three different encodings. The classic encoding is directly derived from the original VRML 2.0 syntax. The X3D XML-encoding allows encoding content with all XML validation and modification benefits, and the binary-encoding allows compressing and storing large data-sets very efficiently. 3D (X)HTML-ized graphics requires an XML-encoded format, and therefore XML is the only encoding supported by the framework to be inserted directly into the (X)HTML page. However, some backends (e.g. X3D-SAI plugins) will support classic and binary file references via X3D "Inline" nodes.

The goal was to support HTML and XHTML. Unfortunately, only the XHTML encoding is very close to the original X3D-encoding. XTHML supports XML namespaces [W3C 2009d], which are used to identify the X3D content. It also supports case-sensitive tag-names and therefore the original upper-case names are used. Since it is XML, it also works with self-closing tags (see example).

```
<X3D profile='HTML' xmlns=
  "http://www.web3d.org/specifications/x3d-namespace">
  <Scene>
     <Shape>
        <Appearance>
            <Material id='mat' diffuseColor='1 0 0' />
        </Appearance>
        <Box DEF='box' />
     </Shape>
  </Scene>
</X3D>
```

As opposed to that, the HTML encoding does (not yet) support namespaces – therefore only the x3d-tag name is used to identify the content. HTML-tags are also not case-sensitive and lower-case names should be used. HTML also only supports self-closing tags for language specific VOID-elements[4], which are elements that do not have any children. In X3D any node can have at least X3D "Metadata" nodes as children and therefore no X3D node complies the criteria for a VOID node. Hence, self-closing tags are not supported here.

```
<x3d profile='HTML' >
  <scene>
    <shape>
      <appearance>
        <material id='mat' diffuseColor='1 0 0'>
        </material>
      </appearance>
      <box DEF='box'></box>
    </shape>
  </scene>
</x3d>
```

3.4 X3D-Scope and Element Identification

X3DOM uses X3D data embedded in (X)HTML pages and optional X3D-XML sources referenced by the embedded part. The external parts are linked with "Inline" nodes and loaded asynchronous as soon as the node is appended to the DOM. The external files can reference further X3D-XML files and therefore build a hierarchy of asset container. X3D uses DEF-attributes to identify nodes in a given scope. All embedded nodes, which are defined inside of a X3D-tag, define a single name-scope. Every "Inline" content and Proto instance defines yet another scope that can be used to handle separate DEF/USE relations. Since the proposed HTML-Profile does not include prototypes we only have to handle separate DEF-scopes for every "Inline" node. DEF-names are only used to USE and ROUTE nodes. The X3D spec provides the IMPORT/EXPORT mechanism to create Routes between different scopes.

Unlike X3D, (X)HTML uses id- and name-attributes to identify nodes in a global scope per document. These attributes can be used to access any element in the X3D content, which either can be contained in the embedded scene or in the inlined tree. If there is a single id- or name-attribute in the inlined content the system will append the full tree to the DOM. Therefore, *getElementById()* and *getElementByName()* can be used to find and reference nodes in all content parts. The web browser does not recognize the types for the unknown X3D elements (e.g. "Box") but stores the nodes and attributes unchanged as part of the DOM tree. The tag names are preserved and therefore *getElementsByTagName('Box')* can be used to collect all "Box" elements. This works even for content which is loaded afterwards via "Inline" nodes, if this content contains an id- or name- attribute. This allows the content provider to control the access to nodes. If there is a single id-attribute all *getElementX()* will work even in the inlined content, otherwise it is considered as a black box with internal Routes and DEF/USE scope.

3.5 DOM Changes and SAI Field Interfaces

The DOM HTML specification defines the standard for accessing a HTML document and the DOM's tree structure relates all elements. There are three DOM changes that affect the X3D backend: inserts/ removals of element and attribute changes. The application developer can use JavaScript standard methods like *element.appendChild()*, *element.removeChild()* and *element.setAttribute()*. The *setAttribute()* function needs text encoded values but can handle e.g. HTML colors like 'blue' directly.

```
var materialElement = document.getElementById('myMat');
materialElement.setAttribute('diffuseColor','blue');
```

[4] http://www.w3.org/TR/html-markup/syntax.html#syntax-elements

These changes are automatically synced to the backend graph without any further notification mechanisms from the application developer. The JavaScript layer is able to recognize those updates using DOM Mutation Events[5] if the backend is not able to monitor the graph changes. Thereby, standard libraries like *jQuery* can be used to traverse the X3D graph or to update node attributes.

The standard *element.setAttribute()* methods to update attributes and therefore X3D-field values is quite inefficient for large updates (e.g. 10000 points) since the data has to be encoded and parsed as text-value. To accelerate accessing and updating large single- and especially multi-value fields we support a subset of the SAI standard interface to access the field values via *element.getField()*. These changes are again automatically synced to the backend graph. Here we use special ECMAScript 5 getter to catch and process the changes if they are not handled in the backend automatically.

```
// update single point
document.getElementById('coord').
         getField('point')[4711].x = 0.815;
```

3.6 CSS Integration

There are several ways to integrate CSS into X3DOM. In general, HTML tags can be styled with CSS arguments in three ways: by selecting a specific tag itself, by selecting its ID or by declaring a CSS class definition. The syntax for the CSS statement always is `selector {property:value;}`. Almost all CSS properties are valid for all major HTML tags. Thinking of a `<div>`, `<p>` or a `<span>` tag, all of them have a padding, margin, color or border attribute to style the appearance. This is different to X3D tags, where interesting attributes for CSS would be very tag specific.

In contrast to HTML4 or XHTML, X3D has also no strict separation of structure and style. Not every X3D tag describes the content inside (semantically), but rather styles it or handles both, structuring and styling. Best examples for this are `<Material>` and `<Transform>` nodes. And unlike to HTML, there is no default render direction or page flow, except that untransformed elements are always positioned inside the 3D world's origin. All these issues have to be taken into account, if we think of a CSS integration, where X3D tags could be styled like HTML tags inside a DOM that originally was designed for 2D elements.

Yet, CSS transformations are still valid for the X3D enclosing canvas or container. WebKit for Mac OS X supports CSS based 3D transformations and shows successful ways to enrich CSS rules with 3D capabilities, since the used elements remain to be basic HTML tags. 3D effects are achieved by simple perspective projections. The basic styling properties (e.g. for width, height, color or borders) remain nearly the same. There is no support for sophisticated material handling or for shaders. Future work has to answer the question of how to combine X3D DOM elements and CSS transformations without breaking the specifications of both sides.

3.7 HTML Events

Suitable JavaScript events for web pages are defined in the DOM events specification, and most of them like *onkeypress* are more or less self-explanatory. Basically four different types of events can be distinguished: key, mouse, HTML, and mutation events. HTML events comprise events such as *onload*, which is raised when a window or frame was loaded. Here, the fallback model as explained in Section 4 is executed. Mutation events like *DOMNodeInserted*, *DOMNodeRemoved*, and *DOMAttrModified* occur whenever the DOM tree was modified and are discussed in more detail in section 3.5. Most HTML tags can react to mouse events, if an event handler was registered. The latter is implemented either by adding a handler function via *element.addEventListener()* or by directly assigning it to the attribute that denotes the event type, e.g. *onclick*.

There exist several mouse events, which need to be considered for designing the interaction with the 3D scene analogously to the standard HTML use case: *onclick*, *ondblclick*, *onmousedown*, *onmouseup*, *onmouseover*, *onmouseout*, and *onmousemove*. As the names imply, they are triggered on click, double click, on mouse press and release, when the mouse pointer enters or leaves the element, and during mouse movements respectively. Whereas in the 2D case handling of enter, leave, and move events can be reduced to a simple inside test of the current pointer position with respect to the rectangular region of the corresponding HTML element, for the 3D case this gets more intricate as on the one hand complex meshes with lots of animated vertices must be considered, and on the other hand more expensive matrix math is involved like e.g. the transformation hierarchy and projective mapping of the objects.

4 Implementation

The framework implementation consists of a very small JavaScript layer, which searches initially for all X3D elements and monitors the DOM-graph (using DOM Mutation Events) for any X3D element creation/ deletion. For every new X3D element it determines the content profiles and tries to find a matching and available X3D runtime backend. If the JavaScript wrapper has found a suitable backend, it starts the corresponding X3D runtime to process and render the embedded content. Therefore every backend has to provide the following basic functionality:

Detectable The backend must be detectable from the JS-layer without instantiating the backend.

Profile matching The backend must provide a list of supported X3D-Profiles.

Lifetime control The JS-layer must be able to instantiate and destroy the backend on request.

DOM synchronization The backend or an additional JS-layer must be able to monitor changes in the DOM to update the X3D scene-graph.

4.1 Native Backend

This is by far the simplest case. The environment just checks the type of the X3D element using the *element.nodeType* property. If it is a generic *Node.ELEMENT_NODE* the UA does not handle X3D elements as HTML elements and the fallback-model will proceed to find a matching backend. Otherwise, the UA supports X3D natively and the framework returns without and further action. But this is a theoretical case, since currently no UA supports X3D natively.

4.2 SAI/ X3D Browser Backend

A simple mimeType check is used to test the SAI-plugin availability. Content transfer and further synchronization are plugin-feature-specific. There are two possibilities to integrate the X3D graph, namely via serialization or the SAI *importDocument* service [Web3DConsortium 2009]. The following code fragment shows an example where the browser gets triggered for its supported properties and profiles. It also takes care of a profile matching between the underlying plugin and the suggested scene. A proof-of-concept implementation was carried out using Instant Reality's X3D plugin.

```
if (navigator.mimeTypes["model/x3d"] &&
    navigator.mimeTypes["model/x3d"].enabledPlugin) {
```

[5] http://www.w3.org/TR/DOM-Level-2-Events/events.html#Events-eventgroupings-mutationevents

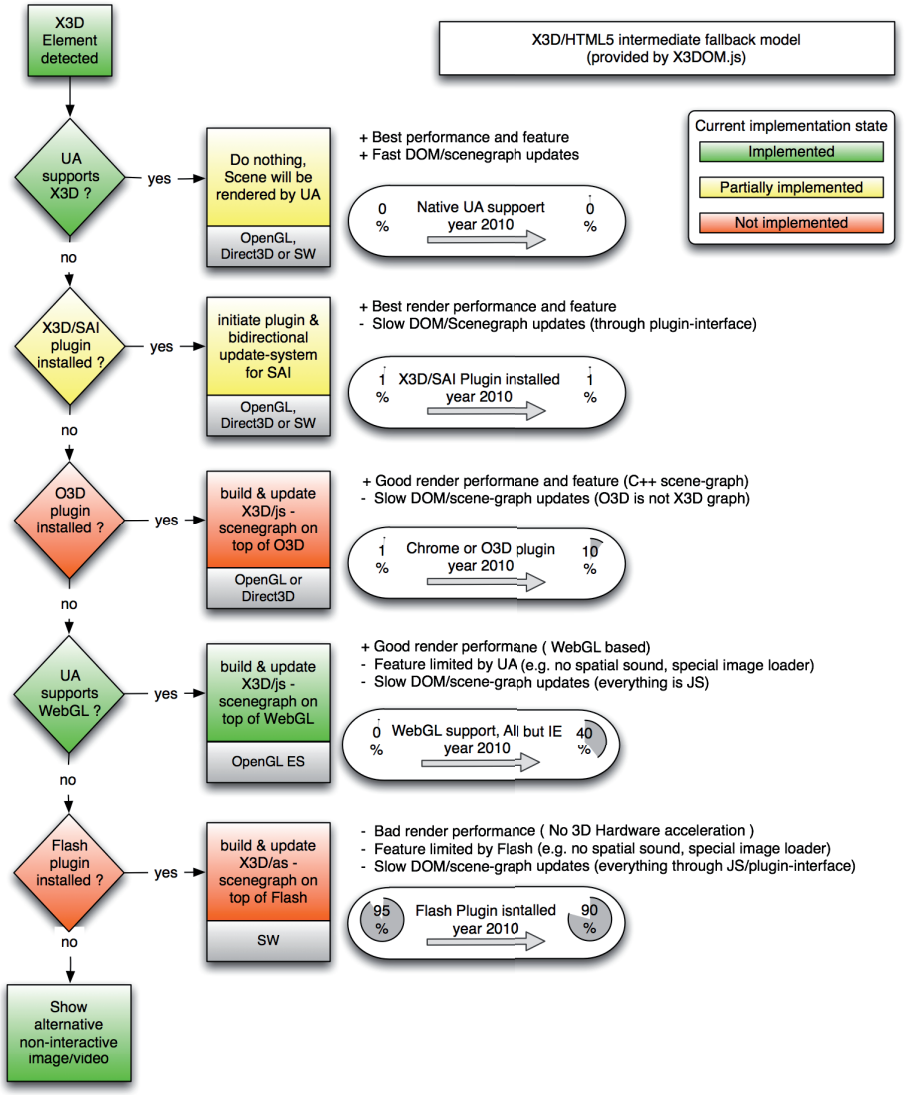

Figure 4: *Illustration of the X3DOM fallback model. The general goal is to build a flexible intermediate evaluation system, which allows evolving and testing the integration model. An important aspect of the system is the support of different rendering and synchronization backends. This embraces the support of native implementations, SAI-based X3D-plugins and WebGL, whereas Flash has a far lower priority.*

```
  var x3ds = document.getElementsByTagName("X3D");
  ...
  embed = document.createElement("embed");
  var profiles = embed.getSupportedProfiles();
  var properties = embed.getBrowserProperties();
  ...
  if (properties["DOM_IMPORT"]) {
    embed.importDocument(x3ds[i].childNodes[1]);
  }
  else {
    domString = (new XMLSerializer()).serializeToString(
                        x3ds[i].childNodes[1]);
    embed.createX3DFromString(domString);
  }
}
```

Depending on the properties, the fallback model has to decide, which procedure of the SAI interface to use, whereas the preferred method is to deliver the whole X3D subtree to the plugin. This approach reduces the complexity of handling changes from within script updates onto DOM elements and therefore the required transformation into SAI calls to the plugin. If the *importDocument* is not supported, the scene-graph first gets serialized as a string and then is used for creating a new scene within the browser plugin. Obviously, this brute force approach is not sufficient. The implementation has to ensure that every node is uniquely identifiable, which can be used to connect both representations, the DOM element and X3D node. The IDs are also necessary for handling updates of the X3D graph.

Modification of the X3D scene and therefore calls to the providing plugin are supported by a thin JS-layer from within the fallback model. Therefore all calls upon any DOM element get transferred to matching SAI calls. This also allows the registration of callbacks as well as controlling the lifetime of DOM and therefore X3D elements from within JavaScript with standard methods.

4.3 WebGL Backend

The WebGL [Khronos 2010] backend is based on JavaScript code designed for the Canvas 3D element, which originally was provided by P. Taylor (`http://philip.html5.org`). Currently it is

the only type of backend that does not require a plugin. As mentioned, WebGL is no additional set of HTML elements but extends the standard JavaScript interface. The 3D rendering context is instantiated within the HTML5 `<canvas>` tag. Thereto, detection is simple: in JavaScript the reference to the OpenGL context can be requested via *gl = canvas.getContext('webgl')*. If the returned `gl` object is defined and not null, the web browser supports WebGL. The backend, as X3DOM test environment, only supports a single X3D profile, namely the aforementioned HTML-Profile. Instantiating, destroying and synchronizing the X3D runtime is also part of the JS-layer. No additional plugin or functionality is needed since the scene-graph is constructed inside of the framework.

4.3.1 Scene-graph Construction and Synchronization

The backend does not traverse the DOM elements directly for rendering, but first builds and connects a scene-graph [Akenine-Möller et al. 2008] hierarchy of special X3D nodes. This is done for two reasons: performance and flexibility. Parsing the attributes of each element every frame would not lead to any interactive frame-rates. The nodes therefore parse the initial or updated attribute values and store the data in typed field container (e.g. an MFVec3f represents an array of SFVec3f objects). The extra scene-graph also helps to translate and handle the differences in the design of DOM- and X3D-graph. DOM is a pure single-parent/ multi-child graph. X3D allows to link parts of the scene using the DEF/USE construct and therefore is a multi-parent/ multi-child graph. The shape 'obj01' in the following example is rendered at two different positions.

```
<Transform id='trans' translation='3 0 0'>
  <Shape DEF='obj'> ... </Shape>
</Transform>
<Shape USE='obj' />
```

Every DOM and X3D node are linked with each other as illustrated in Figure 5. A single DOM node always links to a single X3D node, but an X3D node can link multiple DOM nodes, which allows deleting the 'trans' object in the previous example without effecting the second shape, which still references the single scene-graph node. And there is another imported issue that can be handled with an extra scene-graph: the X3D node tree can exist without DOM counterparts, e.g. for data reference via X3D "Inline" nodes [Web3D 2008]. If the "Inline" content does not contain any node with a given 'id' attribute, the DOM nodes will not be appended to the UA-DOM, only the X3D scene-graph will be constructed and rendered. This allows storing larger parts of the content more efficiently. Furthermore, efficiency and data access in "Inline" content is improved. The WebGL backend uses XHR to load the content, which allows to asynchronously download additional data. This implies that the whole application will start quicker but some of the id's will not be available during initialization.

As mentioned, the backend uses standard DOM mutation events for synchronizing DOM changes to the X3D scene-graph. If the web application adds or removes DOM elements (e.g. via *element.appendChild()* or *element.removeChild()*) the framework automatically constructs or destroys the corresponding scene-graph. No additional user notification mechanism is needed. Mutation events are also used to update all attribute changes. However, not all browsers support the DOMAttrModified mutation event and therefore we overwrite the DOM *setAttribute()* method in the JS-layer as workaround in WebKit-based browser implementations.

4.3.2 Rendering and Media Integration

The X3D material system is heavily inspired by the old OpenGL 1.x fixed function pipeline, and nodes such as "Material" or "DirectionalLight" usually directly map to the corresponding GL functions. As opposed to that, WebGL conforms closely to the OpenGL ES 2.0 API, which is purely shader-based. Thus, there is no matrix stack, no notion of build-in lights or fog etc. Instead, apart from fragment operations like blending and depth test, all former GL states need to be emulated with the help of setting appropriate GLSL shader programs and `uniform` variables. Likewise, all vertex attributes of a mesh, like positions, normals, and texture coordinates, need to be declared as special GLSL `attribute` variables before they can be used in the vertex shader.

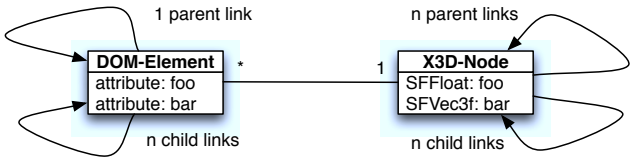

Figure 5: *HTML DOM-tree and X3D scene-graph relationship.*

After having assembled the internal mesh data structures, all geometries can be rendered. As the runtime resides in the browser, for saving resources the scene is only re-rendered on change. This can either happen within X3D due to standard "Interpolator" based animations or because of scripting as outlined in section 3.5. Thereto, first all "Shape" nodes, which hold both, geometry and appearance, are collected including their respective accumulated transformations. After that, all objects are sorted back-to-front along the viewing direction according to their bounding box center for alpha blending. If shadows are enabled, then the shadow pass is rendered. Eventually, for rendering the final image, all required render states and shader programs are activated and the corresponding program variables are enabled such as the model-view-projection matrix etc. Thereby a lot of unoptimized operations (with double precision) in JavaScript are involved and this gets even worse for deep and probably even animated transformation hierarchies.

Another issue are ancient VRML-style multi-index geometry nodes such as "IndexedFaceSet". These cannot directly be parsed into vertex buffer objects on the GPU, but first need to be converted into a single-index representation for the subsequent draw call. Whereas during initialization emerging delays are at best annoying, field changes during runtime can require yet another time consuming conversion. In this regard, newer X3D nodes like the "IndexedTriangleSet", which has only one 'index' field for all geometry properties like 'coord' etc., are better suited here. Another more reduced approach is outlined in Section 3.2.2 with the generic mesh node.

For integrating shadows into X3D we follow [Jung et al. 2007], where it is proposed to extend the light nodes with another field 'shadowIntensity'. For implementation (see Figure 1, right) we use a shadow mapping technique called PCF for obtaining soft shadows [Akenine-Möller et al. 2008]. Thereto, the scene is first rendered from light view. But because currently only 8-bit textures are supported in WebGL, in a special shader we need to encode the depth values into all four channels to avoid strong quantization artifacts.

Albeit JavaScript does not allow any direct access to the file system, textures (at least those whose file formats are supported by a web browser, i.e. mostly png, gif, jpg, and bmp) can easily be implemented by internally instantiating a JavaScript *Image* object, whose 'src' attribute is assigned the 'url' of an "ImageTexture" node. Image loading happens asynchronously, thus, in the *Image's* "onload" event handler the texture is then transferred to GPU memory via *gl.texImage2D(gl.TEXTURE_2D, 0, img)*.

Because most web browsers convert all images to 8-bit RGBA already when loading them (which makes sense for web applications) without providing any means for querying the original image for-

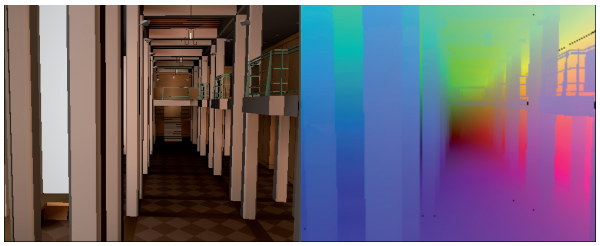

Figure 6: *Rendered scene consisting of one single mesh (left) and picking buffer containing the normalized world coordinates (right).*

mat, the X3D lighting model cannot be fully implemented. According to this lighting model the material's diffuse color modulates the color of the texture in case of a one- or two-channel texture (intensity or luminance/alpha). If instead the texture has three or four channels, the texture color is used. Moreover, because the original information, whether the image has an alpha channel or not, is lost, it can't be used as a selector for alpha sorting and blending, thereby forcing the runtime to first sort all objects and always enable blending. But in this regard, the X3D spec is outdated in that it neither provides a possibility for render state control nor for specifying the sorting order. Hence, to alleviate some of those problems, the extensions described in [Jung et al. 2007] could be used.

Due to the extended tag set of HTML5 the integration of other media like sounds and movies is comparatively simple to implement via the new `<audio>` and `<video>` elements. Similar to an image the latter can be directly captured to a texture object that resides on the GPU, with the exception that this has to happen every frame. Likewise, for implementing the X3D "AudioClip" node, internally an `<audio>` tag is created, appropriately parameterized and appended to the given DOM tree. But because the `<audio>` element does not support 3D sound, again we have the limitation that e.g. the spatial sound model of X3D cannot be implemented.

4.3.3 Picking and Interaction

Mouse events (cf. Section 3.7) require implementing a picking algorithm, which usually is done via ray intersect by traversing the scene-graph and coarsely checking bounding boxes, and testing all triangles if this test passes. But this heavily CPU-based process is not fast enough with JavaScript as lots of matrix operations etc. are involved. Skipping the triangle test leads to a speed-up, but this approximation is wrong for most geometries as shown in Figure 6: here, the whole 3D scene consists of only one single mesh. Hence, there is only one bbox to check, which leads to wrong picking results, not allowing any useful interaction.

Thus, we use a render-buffer-based approach (as shown in Figure 6) instead of standard intersection tests. Here, the picking buffer is implemented by first rendering the scene into a texture attached to a framebuffer object: the normalized world coordinates are encoded into the RGB channels, and the alpha channel contains the object ID that references the rendered "Shape". Occlusions are automatically handled by the depth buffer. By retrieving the values located under the mouse pointer via *gl.readPixels()* the picked 3D object is obtained. Similar to 2D elements the mouse event handlers attached to a picked object, or one of its parents, is called then.

In Figure 3 (right) a 3D object's onclick event is evaluated: the blue sphere represents an X3D "Anchor" node that allows browsing to another web page similar to default web links, and when clicking the red box, a message box with the pick position pops up. Moreover, if one of the 3D objects was clicked, the olive sphere is repositioned to the last pick-point. According to our performance tests, rendering the same scene into the picking buffer with half resolution is already slightly faster than a scene traversal with pure bounding box tests due to the hardware accelerated rendering. Furthermore, this method yields correct results for all types of 3D surfaces.

As floating-point-precision textures are not yet supported in WebGL and due to the 8-bit buffer only a maximum of 255 objects are supported per rendering-pass. Hence, especially for big scenes only a rather bad precision for the coordinates is achieved. Additionally, when solely relying on user-defined vertex attributes without any further semantics – as conceived with the aforementioned WebSG approach – the system neither can determine the bounding box nor calculate a precise geometry intersection. This requires to provide additional hints for specifying which vertex attributes (given as "FloatVertexAttribute") shall be treated as positions.

5 Application Prototypes

We have developed several prototypical applications for testing and to show X3DOM's capabilities. Beneath some feature demos that only concentrate on certain functionalities, our demos (see Figure 7) cover real-world scenarios. The first demo shows a line-up of 3D objects, as it is done with images or videos today. Here, 3D is used as just another medium alike. The second application depicts a car configurator and explains, how 3D could be used in the near future as not only a part of a website, but as an online tool. The last one is a small Augmented Reality (AR) application, which brings virtual and real content on a user's desktop closer together. In the following, we will take a closer look on these applications.

The **Coform 3D Gallery** shows a line-up of over 30 virtual objects. Historian vases were scanned with a 3D scanner. This allows not only a digital conservation of ancient artifacts but offers the possibility for convenient comparison, too. The results have been exported into the X3D file format. The application framework consists of a HTML page with a table grid with 36 cells, each filled with a thumbnail image of a virtual vase object. As soon as the user clicks on a thumbnail, a second layer pops up inside our HTML file showing the vase in 3D. The user can now scale or rotate it or he can close the layer to return to the grid again. Technically, we're opening a subpage with the declared X3D content which is rendered by X3DOM. The subpage is loaded inside an iFrame within each layer inside the main page. Figure 7 (left) shows a screenshot.

Our **Online Car Configurator** shows a 3D car with a minimalistic UI (see Figure 7, middle). A user can choose certain colors from a given color palette or change the rims in real-time. In addition, the car can be viewed in almost any position, since nothing is prerendered. Since the car is a declared 3D model, even small or complex animations of the doors or other parts are possible. Todays online configurators are using semi 3D presentation methods, where certain points of a car are prerendered or photographies. Hence, it is time intensive for developers to change models or other media data and real-time user interaction is limited. The communication between the GUI and the 3D model is implemented in JavaScript. The buttons send onclick events, which trigger functions in the script. A node's attribute is changed by first fetching the node by its unique ID via *document.getElementById()* and then calling the *setAttribute()* function as outlined in Section 3.5.

The **Augmented Reality Application** uses X3DOM for hardware accelerated rendering. It shows the earth globe, which hovers above a marker (Figure 7, right). A second textured sphere shows the actual clouds surrounding the whole planet – via live data loaded into the 3D scene. Like every AR application, we need a webcam and a video image to achieve the augmentation effect. Sine there is no standard or native HTML interface for hardware devices, we make use of Adobe Flash-based marker tracking, which is available for

Figure 7: *Application prototypes (from left to right): Coform3D – a line-up of multiple scanned 3D objects integrated with JavaScript into HTML; a simple car configurator; a desktop Augmented Reality scene using Adobe Flash for marker tracking and X3DOM for rendering.*

non commercial use [Spark 2009]. We adapted the marker tracker and implemented an interface for data exchange between the included Flash object and our HTML page. As soon as the marker is detected, a JavaScript interfaces receives the marker's position and orientation. The tracker's values are used to change and transform our 3D objects. Although the tracking still uses Flash, the demo scene is modular enough to change and switch tracking as soon as there are native standards for camera access available.

6 Conclusion and Future Work

We presented an intermediate architecture that supports the ongoing HTML/X3D integration effort, but also provides a solution to web application developers until X3D is a native part of HTML. In contrast to most other approaches, our framework integrates 3D content into the web without the need to forge new concepts, but utilizes today's standards. Our design and architecture brings X3D to mass market and promotes 3D in a declarative manner for everyday use. As a thin layer between HTML and X3D we deliver a connector that employs well-known standards on both sides. So far, we have presented first implementation results of a scalable architecture for HTML/X3D integration, which provides a single declarative developer interface but also supports various backends through a powerful fallback-model for runtime and rendering modules.

The X3D standard as web technology uses a plugin model for deployment. For desktop browsing it is a question of choice, whether to download a tool or not. But since more and more people are browsing on smartphones or other mobile devices, plugins are still an issue: not all platforms support them and vice versa. Thus, X3DOM may be a powerful solution for mobile devices, too. Moreover, all presented applications utilize our approach of declarative 3D integrated into HTML combined with web standards like CSS and JavaScript. We have also shown the range of possible applications from plain 3D content containers to more sophisticated 3D applications with potential to become an online application, where other web pages may be placed in or be parts of 3D itself.

Future work will cover cross-browser issues (e.g. Microsoft's IE does not support WebGL but may support other 3D APIs in the future) and address the seamless integration of CSS. We intend to finish our fallback model as we'd like to implement missing nodes and continue working on our shader framework, including fog and clip planes, to mention a few.

References

ADOBE, 2010. Flash. http://www.adobe.com/products/flashplayer/.

AKENINE-MÖLLER, T., HAINES, E., AND HOFFMANN, N. 2008. *Real-Time Rendering*, 3 ed. AK Peters, Wellesley, MA.

AMBIERA, 2010. Copperlicht. http://www.ambiera.com/.

ARNAUD, R., AND BARNES, M. 2006. *Collada*. AK Peters.

BEHR, J., ESCHLER, P., JUNG, Y., AND ZÖLLNER, M. 2009. X3DOM – a DOM-based HTML5/ X3D integration model. In *Proceedings Web3D '09*, ACM Press, New York, USA, 127–135.

BENEDETTO, M. D., 2010. Spidergl. http://spidergl.org/.

BRUNT, P., 2010. Glge. http://www.glge.org/.

CONSORTIUM, W., 2010. X3d and html5 working group. http://www.web3d.org/x3d/wiki/index.php/X3D_and_HTML5.

CROCKFORD, D. 2008. *JavaScript: The Good Parts*. O'Reilly, Sebastopol, CA.

DELILLO, B., 2009. Webglu. http://github.com/OneGeek/WebGLU.

GOOGLE, 2009. O3d; an javascript based scene-graph api. http://code.google.com/apis/o3d/.

JUNG, Y., FRANKE, T., DÄHNE, P., AND BEHR, J. 2007. Enhancing X3D for advanced MR appliances. In *Proceedings Web3D '07*, ACM Press, New York, USA, 27–36.

KAY, L., 2010. Scenejs. http://www.scenejs.org/.

KHRONOS, 2009. Webgl public wiki. http://www.khronos.org/webgl/wiki/Main_Page.

KHRONOS, 2010. Webgl specification. https://cvs.khronos.org/svn/repos/registry/trunk/public/webgl/doc/spec/WebGL-spec.html.

MUNSHI, A., GINSBURG, D., AND SHREINER, D. 2009. *OpenGL ES 2.0 Programming Guide*. Addison-Wesley, Boston.

SHREINER, D., WOO, M., NEIDER, J., AND DAVIS, T. 2006. *OpenGL Programming Guide*, 5 ed. Addison-Wesley, Boston.

SONS, K., 2010. Xml3d. http://www.xml3d.org/.

SPARK, 2009. Flartoolkit. http://www.libspark.org/wiki/saqoosha/FLARToolKit/en.

THOMAS, G., 2010. Learning webgl. http://learningwebgl.com/.

VUKICEVIC, V., 2009. Canvas 3d. http://blog.vlad1.com/2007/11/26/canvas-3d-gl-power-web-style/.

W3C, 2009. Declarative 3d scenes in html5. http://dev.w3.org/html5/spec/Overview.html#declarative-3d-scenes.

W3C, 2009. Html 5 specification, canvas section. http://dev.w3.org/html5/spec/Overview.html#the-canvas-element.

W3C, 2009. Mathml. http://www.w3.org/Math/.

W3C, 2009. Namespaces in xml. W3C Consortium. http://www.w3.org/TR/REC-xml-names/.

W3C, 2009. Svg. http://www.w3.org/Graphics/SVG/.

WEB3D. 2008. *X3D*. http://www.web3d.org/x3d/.

WEB3DCONSORTIUM, 2009. Scene access interface(sai), iso/iec 19775-2.2:2009. http://www.web3d.org/x3d/specifications/ISO-IEC-FDIS-19775-2.2-X3D-SceneAccessInterface/.