XML3D – Interactive 3D Graphics for the Web

Kristian Sons∗
DFKI Saarbrücken
Felix Klein†
Saarland University
Dmitri Rubinstein‡
DFKI Saarbrücken
Sergiy Byelozyorov§
Saarland University
Philipp Slusallek¶
DFKI Saarbrücken
Saarland University

Figure 1: A modified version of Chromium browser with XML3D support showing an extended Wikipedia page about Venice with realtime navigation and interaction with a 3D model of a famous Venetian palace (left) and a car configurator demonstrating the tight integration of 2D and 3D content within the same web page inside the modified Firefox browser (right).

Abstract

Web technologies provide the basis to distribute digital information worldwide and in realtime but they have also established the Web as a ubiquitous application platform. The Web evolved from simple text data to include advanced layout, images, audio, and recently streaming video. Today, as our digital environment becomes increasingly three-dimensional (e.g. 3D cinema, 3D video, consumer 3D displays, and high-performance 3D processing even in mobile devices) it becomes obvious that we must extend the core Web technologies to support interactive 3D content.

Instead of adapting existing graphics technologies to the Web, XML3D uses a more radical approach: We take today’s Web technology and try to find the minimum set of additions that fully support interactive 3D content as an integral part of mixed 2D/3D Web documents.

XML3D enables portable cross-platform authoring, distribution, and rendering of and interaction with 3D data. As a declarative approach XML3D fully leverages existing web technologies including HTML, Cascading Style Sheets (CSS), the Document Object Model (DOM), and AJAX for dynamic content. All 3D content is exposed in the DOM, fully supporting DOM scripting and events, thus allowing Web designers to easily apply their existing skills. The design of XML3D is based on modern programmable graphics hardware, e.g. supports efficient mapping to GPUs without maintaining copies. It also leverages a new approach to specify shaders independently of specific rendering techniques or graphics APIs. We demonstrated the feasibility of our approach by integrating XML3D support into two major open browser frameworks from Mozilla and WebKit as well as providing a portable implementation based on JavaScript and WebGL.

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

Keywords: XML3D, HTML5, DHTML, DOM, CSS, web integrated, real-time, ray tracing

1 Motivation

Originally designed as an information network, the World Wide Web has moved to an interactive multi-purpose platform for all kinds of applications, including web-mail, social networks, online shops, web mappings, and collaborative encyclopedias such as Wikipedia. The web consists of ubiquitous W3C standards and quick emergence of de facto standards such as

- HTML [W3C 2010] as a markup language for content and structure,
- CSS [W3C 2010] as a presentation definition language for the style,
- DOM [W3C 2009d] representing the hierarchical structure of the web document and

These technologies comprise what was termed Dynamic HTML (DHTML) and constitute - together with asynchronous server access through the XMLHttpRequest [W3C 2009f] API - the technology stack for nearly all interactive and animated web sites in the browser.
However, HTML itself only supports the description of text and simple, box-shaped 2D graphics including images and video. More flexible 2D vector graphics support has later been added through SVG [W3C 2009a]. However, support for interactive 3D graphics is not yet available in any browser. It can be added through plug-ins that manage the 3D content internally often using a game engine or an X3D renderer. However, the content managed by the plug-ins is completely separate from the rest of the web page. Using it in Web pages requires learning new APIs, new data models, and often unusual scripting languages. Furthermore, these plug-ins are largely incompatible with each other and can often not be installed in certain (business) environments. As a result of this, there is hardly any 3D content on the Web even though it could significantly enhance the capabilities and usability of many Web applications and the fact that it would readily be supported on essentially all the existing hardware platforms for the Web – even down to small and mobile devices.

We believe that a standard technology for 3D graphics on the web should reuse as much as possible of what web-technologies already provide. It should thus extend the stack of Web technologies as well as expand the capabilities of all Web documents and applications. An integrated 3D technology would be familiar to web developers and compatible with existing libraries and tools used in web development. Our approach is therefore to design a 3D technology compatible to DHTML that reuses a maximum of its concepts, including Cascading Style Sheets [W3C 2009a] and the Document Object Model [W3C 2009d].

On the Web most of the content developers will not be graphics experts, making it mandatory to provide an intuitive approach to 3D graphics that eliminates most low-level details anyway. Note that this is similar to text: HTML/CSS also do not provide all the options of a professional text layout engine but are “good enough” for the majority of use cases on the Web. Over time these capabilities can and should be increased in a controlled manner, though.

## 2 Related Work

In the following, we focus on and discuss existing technologies that add sophisticated 2D and 3D graphic capabilities to the web. We also briefly discuss other, non-declarative, API-driven approaches, that do not define a markup language but provide the functionality to create content procedurally. While some of these approaches show great results, the use of a 3D system with an API is orthogonal to declarative web documents and therefore difficult to compare to our approach.

### 2.1 X3D

X3D is an ISO Standard [Web3DConsortium 2008] file format to represent 3D scenes. X3D has XML encoding, an encoding that is backwards-compatible to VRML97, as well as a FastInfoSet based binary encoding. Additionally to the 3D content description, it also defines a full runtime environment, including scripting and an event system.

Besides several stand-alone browsers that implement the whole or parts of the X3D specification, there is a plug-in based web-browser integration model, where the scene is stored and managed by the plug-in and can be accessed and modified using the Scene Access Interface (SAI), which is also part of the standard. SAI bindings are defined for EcmaScript/JavaScript and Java.

Apart from the optional XML encoding, the X3D standard is not referencing or leveraging other W3C standards like CSS or DOM Events [W3C 2000]. The style of an X3D document in terms of shading and transformations is not separable from the geometry description using CSS. X3D defines its own event model, that is incompatible with DOM Events. While the SAI defines an integration model for DOM nodes, it is limited to a one-time import of DOM nodes with no bidirectional update or synchronization mechanisms.

The most frequently used node for storing geometry in X3D is the IndexedFaceSet node. While it is a very flexible geometry representation, for many configurations the data must be pre-processed before it can be passed to low-level graphic APIs, like OpenGL or DirectX, which require VertexArrays for good performance. However, the original data must stay available for possible changes via routing or SAI access, requiring the need to store multiple copies of the data.

The X3D standard defines other geometry nodes that support unindexed or single-indexed vertex data, but these nodes are seldomly used. For most of the existing content and for output from common X3D exporters an interim conversion step is necessary to achieve hardware-friendly data structures.

### 2.2 X3DOM

X3DOM [Behr et al. 2009] is an approach to embed the X3D scene graph into the DOM of a web page. Just like SVG, the 3D content should be displayed in place of the declaration without the need to install a plug-in. To seamlessly integrate X3D into the DOM, its functionality is stripped down to visualization components while dynamics, distribution, security, and scripting are managed through the web technologies provided by the browser.

The proposed architecture of X3DOM consists of a connector component between the browser (front end) and an X3D runtime (back end). The connector transforms the X3DOM scene graph, declared inside the DOM, into the scene graph of the X3D backend. Afterwards, any change of the scene is synchronized between these two representations. While the X3D-backend is responsible for rendering the image, media linked in the X3D scene graph can be resolved using the browsers URI/URL streaming mechanism. The current implementation of X3DOM is based on WebGL [Khronos 2009], which includes an simple X3D runtime implemented in JavaScript.

While the approach of X3DOM is similar to XML3D, it tries to add an existing 3D graphics format into the web, rather than consequentially extending the current technology where necessary. This way, existing X3D content can be reused in the browser - as long as it does not exceeds the proposed DOM profile.

But although X3D is stripped down to visualization and interaction components, it still contains features that do not fit well to the ubiquitous use of DHTML in recent Web 2.0 applications. Thus X3D does not separate the style and layout from the content as it’s done in HTML with CSS and authors can realize interaction with sensor nodes and routes, a mechanism that the web community is unfamiliar with. The common way to add interaction to a web page is via DOM Events and DOM Scripting. Adding DOM Events and CSS to X3DOM would result in two opposed mechanisms to achieve the same behaviour and it is necessary to define which mechanism precedes the other. The same applies for the non-trivial task to synchronize the tree-based DOM structure with graph-based X3D structure. Additionally, this approach shares all the limitations that come with X3D as discussed above.

Although X3DOM is an obvious approach to integrate X3D into the DOM, it is not an intuitive way for web authors to handle 3D objects in the DOM.
2.3 SVG

Scalable Vector Graphics (SVG) [W3C 2009e] was introduced in 2001 by W3C as an XML-based file format for describing two-dimensional vector graphics. SVG integrates and leverages other W3C standards, like CSS2 and DOM2. A lot of features are modeled after HTML. All modern browsers have or intend to have native support for SVG embedded into XHTML [W3C 2002]. HTML5 [W3C 2010] allows inline SVG even in text-encoded HTML. SVG also uses DOM events, event handlers such as onclick and onmousemove can be assigned to any SVG graphical object, just as in HTML.

SVG has capabilities for declarative 2D animations via SMIL [W3C 2008] animation elements. But since SMIL does not fit well into DOM concepts and these elements are not very well supported by authoring tools, they are rarely used. Instead most web applications use DOM scripting capabilities for dynamic SVG content.

The CSS3 Animations [W3C 2009c] proposal could be a good alternative for declarative animations.

SVG has some other design issues. Though SVG leverages CSS, most properties can also be expressed as attributes, which can lead to ambiguous behavior. Providing a <use> element, SVG allows the reuse of complete sub-scene graphs. This makes the CSS inheritance down the referenced tree even more complex and hard to implement. Also SVG could leverage existing HTML elements rather than defining its own, i.e <audio>, <video>, <script>, <style>, <a>, and <img>.

SVG and XML3D have a similar approach to extend the web by graphic capabilities. This is a good reason to take SVG into account when designing a web format for 3D graphics. Anyway, we try to remain even closer to HTML, with respect to its much larger user base.

2.4 Imperative Approaches

There are several imperative approaches that allow rendering of complex 2D and 3D graphics. In contrast to declarative document-based approaches, those technologies usually provide an API that can be accessed via a scripting language in order to create the content procedurally.

One example in this category is WebGL [Khronos 2009], which is managed by the Khronos Group. It is basically a JavaScript binding to native OpenGL ES 2.0. Using this low-level API, one can render hardware-accelerated 3D graphics inside the HTML <canvas> element. WebGL is currently in the process of being natively implemented in several browsers, including Firefox, Firefox for mobile, and WebKit-based Browsers.

Other approaches are plug-in based technologies, which connect a separate rendering system to the browser and provide an API to be accessible from the web document. An example is the current version of Google’s O3D [Google 2009], which allows the user to create a scene graph with JavaScript. O3D focuses on games for the Web and supports skinning, programmable shaders, and other features.

Even though those imperative approaches allow the display of complex graphics in the browser just as well as declarative approaches, they have to be used in a way that is orthogonal to web technologies. On one hand, we have the declarative content of the web page which may be generated from the Web server. It is designed to easily be processed by other applications using common XML-based tools, including Web indexing, transformations, annotations, and many more.

On the other hand, we have a program or script, included in the web document, that creates the 3D content, but where the content itself is never explicitly visible. As a result, the representation of the 3D content is inherently different and separate from the rest of the web page’s content and incompatible with a vast amount of programs, scripts and tools used in web development. Thus, for a 3D graphics technology that should be used as an extension of web documents, the imperative approach is inherently unfit.

3 Properties of HTML

In order to integrate support for 3D graphics into web documents, we need to understand their properties. Therefore, we have a closer look at how HTML documents are structured and used by web applications.

The content of complex web pages and web applications is often generated on the server dynamically. There are many document management systems that are used to generate the HTML output. The output is sent to the client and rendered by different web browser on different platform using a variety of different layout and rendering engines. Because of the common Web standards Web applications are portable across platforms, browsers, and renderers.

Another important concept of modern web design is the separation of content and style. The content is described by the HTML document, containing text, links, and references to images and other resources within a hierarchical structure of this content. The way this content should be displayed is separated into the “style” described by cascading style sheets (CSS) that allows to specify a fixed set of formatting properties that are applied to the elements of the HTML document.

A style sheet is referred to by the HTML document and can be reused for multiple documents. This separation allows web designers to easily modify the layout across a large number of web pages by modifying the style sheet alone. Since CSS allows a generic assignment of formatting properties, using element classes and identifiers for referencing, the combined size of the HTML document and style sheets tend to be significantly smaller compared to an HTML document that has formatting attributes assigned to each element individually. Also, while HTML documents might be reloaded on each access, style sheets can be better cached and reused since they tend to change less often, reducing the downloading overhead even further. Due to all these advantages, there is a general trend in web design where the content of the HTML document is reduced to the minimum while style sheets are getting more complex.

Finally, the content and layout described by HTML and CSS respectively are mostly static. Declarations of dynamic behavior and interaction are limited to links, forms, and style changes depending on mouse position, element focus, and other properties. The general approach to tackle this limitation is to provide an interface to access and modify the content of the web page via scripting. This has been standardized 1998 with the introduction of the Document Object Model (DOM), a cross-platform and language-independent convention for representing objects in HTML, XHTML, and XML documents. Today, the DOM is used together with JavaScript to access and dynamically modify the content of a web page. DOM events allow scripts to react to user interactions. This way even complex dynamic behavior and interaction mechanisms can be realized.

In summary, web applications consists of a declarative part for content and layout and an imperative part for dynamics and interaction.
4 Requirements for Realtime 3D Graphics

Hardware technology has reached a state where specialized graphics hardware can be found in any modern computer as well as in most smaller devices like mobile phones. We aim at maximizing the functionality of the GPUs exposed for declarative scene descriptions while minimizing the necessary overhead caused by the browser environment.

4.1 Performance and Quality

We assume that within the browser XML3D is used for fully interactive web applications with high realtime requirements and not for off-line rendering. We also assume that it is sufficient to initially expose triangle meshes only as commonly supported by OpenGL. Because of the way GPUs work we assume that the general concept of vertex arrays from OpenGL should be used as for organizing the basic data structures. Additionally, since mesh data can be quite large, 3D graphics relies on instantiation of meshes to display complex scenes without using too much memory.

Since 3D graphics can be used in very different contexts, the quality of the results is hard to measure on a global scale. However, a frequent goal is to render photo-realistic images which commonly includes effects like procedural shading, shadows, reflections, and refraction. A forward-looking specification for the Web should be able to specify these effects in a portable way even if they may not be rendered fully correctly by all renderers. A consistent and portable specification is of key importance. The photo-realism is enhanced even further by adding detail to the scene, either through complex geometry or detailed surface shader. All this should be supported for static as well as dynamic content.

4.2 Renderer Independence

Most of today's real-time graphics are rendered using the rasterization algorithm with the aid of fully programmable GPUs that use shaders to specify different parts of the image synthesis process such as a surface appearance, light emission, geometry detail, and others. While shaders can be used as parts of a declarative scene description, they are often also used to procedurally specify the rendering process itself (e.g. for multi-pass rendering). Such a use would be largely incompatible with the general declarative approach we are aiming for.

While the rasterization approach of GPUs is procedural and describes how an image is generated, ray tracing starts with a declarative scene description and is used to simulate the light transport (to various degrees) within a scene, making it more amenable to the declarative approach taken here. Since ray tracing has global access to the scene at each point of the rendering process, it is easier to simulate effects such as shadows, reflections and refractions (see Figure 2). Furthermore, this approach is much more intuitive and provides "correct" images automatically and with little knowledge about the rendering process itself. This should greatly help lay users in designing their scenes and getting the results they expect.

Continuous advancements in hardware technology and algorithmic improvements have increased the performance of ray tracing to the point where interactive frame rates can be achieved on commodity hardware [Shirley et al. 2006]. Additionally, major hardware vendors have started to consider ray tracing when designing new hardware [NVIDIA 2010] providing the option to use ray tracing as a possible alternative to rasterization for real-time 3D graphics in the future. However, for most scenes its performance is still not on-par with rasterization, suggesting that 3D scene descriptions for the Web should support both options.
encoding. This ensures that we can directly transfer them onto the GPU (e.g. through OpenGL VertexBufferObjects). We encode these values via arrays elements based on the data types they describe: `<float>, <float2>, <float3>, <int>`. The content of these elements can then be parsed very efficiently and possibly be reused at various places throughout the scene. An example is given in Figure 3.

To accomplish reused the binding of the arrays to specific named parameters of a mesh or its associated shaders we enclose these arrays in `<bind>` elements. Besides the parameter name, the bind element can also provide the semantics (e.g. position, normal, vertex index, etc.) which is required for some operations within the renderer. Using this structure, we can declare the data of the `<mesh>` in a generic way that is easy to convert to from other file formats and graphics APIs.

The geometry of the scene can be structured by being placed into `<group>` elements. Each `<group>` defines a coordinate system that can be manipulated via CSS using the `transform` property as described in the next section. The transformation is relative to its parent. This way, a tree of `<group>` elements represents the transformation hierarchy of the scene. Additionally to geometry, this hierarchy may include other elements, including `<light>` elements for light sources and `<view>` for viewpoints.

As discussed in Section 4.1, instantiation is an important feature of 3D graphics to display complex scenes with a limited amount of memory. Unfortunately, being a tree-based structure, XML documents do not allow instantiation of declarative content. We work around this fact (similar to SVG) by introducing a `<use>` element, that instantiates a mesh in place of its declaration. The reference is formed by using the mesh’s document identifier. The `<mesh>` element to be instantiated can be referred from the same or an external XML3D document.

In contrast to SVG, we do not support the instantiation of whole sub-scene graphs by referencing a `<group>` element, therefore avoiding the problems with CSS inheritance as described in Section 2.3. As this would be a valuable addition and can potentially save memory, we hope future research eventually finds meaningful ways to support this feature without the current issues.

With this set of elements, XML3D allows a generic declaration of the geometric content in a way comparable to the pure text content of HTML. Figure 3 shows a simple XML3D example using the above XML3D elements. Obviously missing so far is the possibility to define the appearance of objects, which we consider to be part of the “style” of a 3D scene.

5.2 Style

Applied to HTML documents, the style defines the size, position, and appearance of box-shaped layout element. We translate this concept also to 3D content, where it describes the transformations and appearance properties of mesh elements.

Transformations can be assigned to groups using CSS, in which case all children of the group are transformed. For the description of 3D transformations with CSS, the CSS 3D Transforms specification [W3C 2009b] is suitable. It is a working draft of the CSS working group and is already implemented in some versions of WebKit, i.e. for the iPhone. For browsers that don’t provide CSS 3D Transforms, there is also the possibility to refer to an XML3D `<transform>` element using the CSS URI syntax. Figure 3 shows how a transformation described by a `transform` element is assigned to a `<group>` group.

Regarding appearance properties, it would be nice to express all kinds of shaders and their parameters via CSS only. But since surface shading in 3D graphics can be highly complex with many different shaders each having its individual parameters, this requires to allow an arbitrary number of arbitrary named shader attributes. CSS cannot currently provide this as it only allows for a fixed set of predefined properties.

Instead, we provide two options to describe surface shading. One is based on a set of default shader models that can be configured using a fixed number of CSS properties. These properties can be applied on `<mesh>` and `<group>` elements. They can be inherited similarly to other CSS properties. Here we use the typical standard shaders and shader parameters as available in OpenGL, X3D, and other formats and APIs.

Alternatively, the user can declare a shader inside the XML3D document, using the `<shader>` element. This element contains a combination of `<bind>` and value elements, just as `<mesh>`, to allow a generic description of arbitrary shader attributes. The `<shader>` element also refers to a `<script>` element containing the code for the surface shader. A shader declared this way can be assigned to a `<group>` element via CSS using the newly introduced `shader` property.

Using CSS for shader assignment is a new approach in computer graphics. It is a powerful technique as it allows to build up external shader libraries that can be reused for several scenes. Other common CSS mechanism such as selectors and pseudo-classes can be directly applied to the 3D objects. For example, the support of media dependent style sheets allows to vary the shader complexity tailored to specific output devices.

Similar to surface appearance we could specify the emission properties of surfaces through an additional “light-shader” style param-
eter. This allows to turn any surface into a light source. Additionally, for simplicity we currently still support specialized `<light>` elements that allow to define the common light models as defined in OpenGL and other APIs. However, their functionality is fully redundant.

The surface shader code has to be independent from the rendering algorithm, thus we cannot rely on a single, traditional shading languages such as GLSL, HLSL, or Cg designed for the rasterization pipeline. Instead we use AnySL, a portable, renderer-independent shader representation that can be used in combination with multiple shader languages. We discuss AnySL in more detail in Section 6.2.1.

Methods to animate CSS properties are proposed in the CSS Animations Module [W3C 2009c] working draft. These techniques can directly be applied on XML3D as well, for instance to animate transformations of objects or shader attributes.

This again shows the advantages of reusing the Web technology stack as much as possible as we can leverage existing work and upcoming improvements automatically. In reverse, we may be able to provide new concepts that may turn out to be useful for other parts of the technology stack.

### 5.3 Scripting

As XML3D is designed to be a thin additional layer on top of existing Web technology its functionality is designed to be generic, orthogonal, and is kept to the minimum needed to display and interact with 3D content. Since XML3D is a part of the DOM, which already provides scripting and event services, most additional functionality can usually be implemented through scripts instead of new special-purpose elements.

The DOM API includes interfaces to create, connect, and remove elements as well as to modify their attributes. Also, element type may define additional functions to access or modify attributes or get supplemental information. Those functions may return specialized data types, such as three dimensional vectors, that provide additional functions such as math operations and are easier to handle compared to the simple strings used in the generic DOM interfaces.

Harnessing JavaScript’s built-in timer functions, it is easy to create animations, by performing sequential changes of the DOM. For instance, the translation properties of a group can be changed every few milliseconds to create a movement of its content.

Generic DOM event listeners reacting to mouse movement and clicks can be connected directly to individual scene elements, including `<mesh>`, `<group>`, and `<use>` elements. Those events are triggered whenever the mouse pointer is placed above the rendered representation of the element, which is exactly the same as for HTML elements. With these simple constructs, complex interaction mechanism can be created, including drag and drop of 3D objects.

While the event system described by the DOM events specification of the W3C is also applicable to 3D graphics, the predefined event types need to be extended to provide additional informations that might be needed for certain 3D applications. For instance, specialized 3D mouse events should provide informations, like depth, parametric coordinates, and global-world coordinates.

With the support of the DOM and DOM events, XML3D scenes can be extended with dynamic content, interaction mechanisms, and additional functionality, exactly like HTML-based web pages. Therefore, many JavaScript- and Ajax-based technologies originally used for web pages can be easily extended to work with XML3D as well.

### 6 Implementation

We started several implementations for XML3D in order to support a wide range of browsers. Therefore we created a formal and machine-readable description of the XML3D specification. Also all DOM IDLs, some source code, the documentation and the XSD are generated from this description. Thus it is possible to propagate changes or additions in the specification very quickly and consistently to all implementations while reducing human errors in this repetitive process.

In this section, we describe important aspects of our implementation strategy as well as the individual implementations. An schematic overview of our implementations can be seen in Figure 4.

![Figure 4: Overview of all current implementations of XML3D. The native implementations are based on RTSG for scene graph management, RTfact for ray tracing, and OpenGL for rasterization (currently work in progress). Any implementation may use AnySL for shader descriptions.](image)

#### 6.1 WebGL-based Implementation

One implementation of XML3D is based on JavaScript and WebGL, similar to the current release of X3DOM. It supports most features, including the embedded XML3D scene inside an XHTML document that is part of the DOM and can be modified and attached with event listeners correspondingly. However, CSS based features are not implemented as DOM scripting does not currently provide access to all CSS properties of an element or the possibility to attach event listeners to style changes. The 3D scene is displayed inside a `<canvas>` element, that is created for each `<xml3d>` element present in the document. Rendering is performed with WebGL and thereby fast, but currently limited to hardware-accelerated rasterization. Since only a subset of the features can be implemented and the JavaScript DOM scene graph traversal lacks performance, this implementation is currently used only as a prototype.

#### 6.2 Native Implementation

In addition to the WebGL-based implementation, we have integrated XML3D support natively into the Mozilla browser framework (the basis for the Firefox Web browser) and into WebKit (the basis for browsers like Google Chrome, Apple Safari, and many open-source browsers). These implementations target to eventually support all features of XML3D, different rendering algorithms, and large 3D scenes – a lot of which will be difficult or impossible to support efficiently with a JavaScript and WebGL-based implementation, due to language specific limitations (see Section 6.3 ) and performance issues (see Section 6.1 ).

The native implementations of XML3D are composed of several components. Each browser is modified to support the new ele-
ments as well as their DOM interfaces. In our implementation the XML3D subgraphs of the DOM is implemented by an extended version of the RTSG [Rubinstein et al. 2009] scene graph management library, that supports an efficient and renderer-independent scene graph representation. In contrast to the original RTSG, the extended version supports the additional flexibility of XML3D in addition to the traditional X3D structures.

In order to avoid data duplication and synchronization we try to hold all data in the RTSG structure only. Generic access through DOM functionality and specialized API script access is implemented by small DOM wrapper objects that access the data from the optimized representation in RTSG, wherever possible. Only for access through the generic, string-based DOM interfaces do we have to convert to and from this representation on-the-fly. As these interfaces are inherently not suitable for 3D data and can easily be replaced by accessing the elements with their specific API, this is not a drawback.

Furthermore, when we can go even one step further in reducing the footprint of the implementation by storing the (often large) 3D data arrays not in the browser but instead entirely on the GPU via OpenGL VertexBufferObjects. As extensive operations on the data sets through JavaScript are unlikely in the first place and will require obtaining an array object first, we can implement this access through mapping the VertexBufferObjects into the browsers address space on demand.

6.2.1 AnySL

In order to be independent of the rendering algorithm we use AnySL [Programming and Computer Graphics Groups of Saarland University 2010], a portable multi-language shading description, when the predefined shading functionality (of the common single-textured Phong shading) is not sufficient. The key idea of AnySL is to compile a shader from different languages into an pre-processed, intermediate, renderer-independent, and thereby portable representation, that can also be referenced directly in XML3D documents. This would improve loading and compilation efficiency and eliminate the need to develop and support front-ends for all relevant shading languages.

Upon loading into the browser, this common representation is then merged with renderer-specific interface code – describing the implementation of basic types and renderer interfaces – and compiled using an embedded JIT compiler (in our case LLVM). In the process, the shaders code is optimized together with the renderers interface code completely eliminating any overhead between the two (often caused by the otherwise required separate compilation of the two codes, which freezes APIs into static ABIs). Furthermore, the embedded compiler can be used to automatically parallelize scalar shader code to match the (SIMD) parallelism of a given renderer. The compiler can easily even take data/scene dependent information into account when compiling the shader interfaces, such as the set of used shaders or light sources. For further information about AnySL we refer to [Programming and Computer Graphics Groups of Saarland University 2010].

The AnySL framework solves the key technical problems of efficiently and portably supporting a variety of different shading languages on all the renderers that may be used in Web browsers on different platforms. As a result, it is simple to achieve high performance and effectively integrate support for shader languages into a renderer without having to implement a complete new compiler and code generator for each one. For embedding AnySL-shaders we use <script> elements, as shown in Figure 5. A rendering of this example combined with an HTML text overlay is shown in Figure 6.

![Figure 5: An XML3D scene with embedded AnySL-shader.](xml3d_metal.png)
Figure 6: A rendering of an XML3D scene from Figure 5 with AnySL/RenderMan texturing shader in a Firefox browser. The shader example is combined with HTML text overlay.

Figure 7: A product configurator for a car based on XML3D. The user can click on the 2D HTML elements links surrounding the rendered image to change viewing angle, switch on and off light sources, modify the cars color, or open and close its doors, trunk and hood.

supported by either X3DOM WebGL or our XML3D implementation. A possible solution would be to propagate this limitation into the XML3D specification in the first place. However, this may complicate the handling of large geometric objects that depend on the unique representation of shared vertices.

We also encountered a general problem, when embedding 3D data into the DOM. Since the representation of 3D geometry consumes a lot of data, especially compared to the usual content of HTML web pages, the document size can increases significantly to many megabytes for non-trivial scenes. The DOM implementations of the browsers, on the other hand, are not currently designed to parse and handle such large amounts of data. Some further optimizations are required here, which will likely benefit the browser as a whole.

Another current limitation concerns the handling for vertex data in the browser. Both WebKit and Mozilla always represent the text within an HTML element as a native “text nodes” within the DOM data and do not allow to override this depending on the containing element. As a result both currently store the full text representations of all vertex data in the DOM. Furthermore, WebKit also does not allow overriding the attribute access functions of DOM elements, always storing their values in text form as well. Not only does this increases the memory footprint significantly and eliminates the benefits of not having to store extra copies of them, it also increases the computational overheads for serialization and de-serialization of this data when synchronizing with RTSG. However, non of these issues poses a permanent issue and should be straight forward to eliminate.

7 Results

We have implemented XML3D based on WebGL as well as natively in the browsers Firefox and Google Chrome on Linux and Windows. In all versions, the scene can be declared inside an XHTML document as described in Section 5. The DOM functionality is fully supported in all implementations, allowing a scripting application to arbitrarily access and modify the 3D content of the scene. DOM events are fully supported in the native implementations, providing the ability to implement almost arbitrary interaction metaphors as scripts, e.g. by connecting mouse event listeners to XML3D elements. With DOM lacking ways to access CSS states and to monitor CSS modifications it is hard to implement CSS support for the WebGL/JavaScript implementation of XML3D. Thus CSS is not yet fully implemented. While the WebGL-based implementation renders the scene with hardware-accelerated rasterization through WebGL in the browser, the native implementations additionally provide software-based, Whitted-style realtime ray tracing for rendering the scene.

As a thin extension layer, some functionality that is commonly included in other 3D scene graphs, like a camera controller, are not available by default in XML3D. However, as discussed in Subsection 5.3, those features can be easily implemented with the use of scripting languages. We expect that common libraries of such interaction techniques will become available, some of which may eventually become new XML3D elements.

In our examples, we used JavaScript to implement (a) common camera controllers for interactive navigation of the scene, (b) a scene inspector, providing information about the object the user clicked on, a (c) transformation interpolation framework that can be used to animate content by modifying their transformation properties, (d) various sensor nodes that return information about visible elements in the scene, and many more.

As a result most of the basic X3D functionality is already available, with just a few new HTML elements that perfectly fit into and reuse functionality of the existing stack of Web technologies. Any current Web programmer should be able to start using this new functionality in his applications immediately. A number of converters and exporters have also been implemented to provide the initial 3D content.

8 Future Work

As seen in Section 6.3, there are still a number of limitations in the current browser implementations, that needs to be addressed quickly. Furthermore, we continue to scan traditional scene graphs and graphics applications for functionality that may be useful in the Web context either through a JavaScript library implementation or additional XML3D elements, where necessary or useful.

Due to the large size of web documents with embedded geometry data, we plan to also support external mesh formats that can be referred to from within the XML3D document, similar to the way image content are stored external to HTML documents.
Additionally, XML3D still lacks mesh processing capabilities comparable to vertex shaders, geometry animations. Even though it is possible to modify mesh data from JavaScript, it not as efficient for large geometries. We plan to address this limitation by introducing XFlow, a new data flow based technology to describe processing of arbitrary large data sets in the form of the XML3D arrays.

Finally, we plan to add the option to support server-based rendering with XML3D for clients that lack the computational power for complex 3D scenes or where issues of intellectual property prohibits the distribution of the 3D content to a client. With server-based rendering the content of the scene is stored on the server, which receives the URL of the scene as a AJAX query. The server then sends the rendered images as low-latency streaming video to the browser on the client, while user interface events are sent to the server.

9 Conclusion

In this paper a minimal extension to the existing stack of web technologies that adds interactive 3D graphics as native data type for Web browsers and browser-based applications. The approach uses a declarative description that is fully integrated with the DOM and leverages CSS, DOM scripting, and DOM events to create full-scale 3D scene graph fully integrated with the other Web technologies. Because it reuses large part of this technology stack, it benefits from any improvements and extensions.

The approach is as close and familiar to modern web programming as possible and shows how tight 3D graphics can be integrated into HTML and the DOM. We demonstrated, that at least for the basic functionalities it is not necessary to avoid declarative approaches and fall back to APIs, such as in O3D. The proposed format – combined with fast rasterization and high-quality realtime ray tracing technology – enables millions of web authors to create highly interactive web sites with high quality 3D content without the need to learn new and complex graphics API. We hope that thereby XML3D will make interactive 3D graphics available on any platform and for everyone.

Finally, our implementations into Mozilla and WebKit browsers demonstrate, that XML3D is not just an interesting proposal, but can be integrated into recent browser frameworks with manageable effort. Modified versions of the browsers will be available on http://www.xml3d.org for free download as open-source technology.

References


