Dissertation zur Erlangung des Grades des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik der Universität des Saarlandes

XML3D: Interactive 3D Graphics for the Web

Kristian Sons
Saarbrücken, 30. März 2017

UNIVERSITÄT
DES
SAARLANDES

Datum des Kolloquiums	30. März 2017
Dekan der Fakultät MI	Prof. Dr. Frank-Olaf Schreyer
Betreuer der Promotion	Prof. Dr.-Ing. Philipp Slusallek
Prüfungsausschuss	Prof. Dr. Antonio Krüger (Vorsitzender) Prof. Dr.-Ing. Philipp Slusallek Prof. Dr. Sebastian Hack Prof. Dr. Anthony Steed Dr.-Ing. Richard Membarth (akademischer Beisitzer)

Abstract

The web provides the basis for worldwide distribution of digital information but also established itself as a ubiquitous application platform. The idea of integrating interactive 3D graphics into the web has a long history, but eventually both fields developed largely independent of each other.

In this thesis we develop the XML3D architecture, our approach to integrate a declarative 3D scene description into exiting web technologies without sacrificing required flexibility or tying the description to a specific rendering algorithm. XML3D extends HTML5 and leverages related web technologies including Cascading Style Sheets (CSS) and the Document Object Model (DOM). On top of this seamless integration, we present novel concepts for *a*) a lean abstract model that provides the essential functionality to describe 3D scenes, *b*) a more general description of dynamic effects based on declarative dataflow graphs, *c*) more general yet programmable material descriptions, *d*) assets which are still configurable during instantiation from a 3D scene, and *e*) a container format for efficient delivery of binary 3D content to the client.

Additionally, we present three implementations of XML3D. The architecture and implementations have been successfully evaluated for designing 3D web applications from various domains, including commercial use for visualization of PDM data from the automotive industry. Finally, we discuss how our approach contributes to existing 3D application models.

Zusammenfassung

Das Internet hat sich nicht nur zu der Basis für den weltweiten Austausch von digitalen Informationen entwickelt, es ist auch eine universelle Anwendungsplattform. Die Idee, diese überall verfügbare Plattform um 3D-Inhalte zu erweitern, existiert schon seit langem. Trotzdem haben sich 3D-Grafik und das Internet weitestgehend unabhängig voneinander entwickelt.

In dieser Arbeit präsentieren wir die XML3D Architektur, unseren Ansatz für die renderer-unabhängige Beschreibung von 3D-Szenen mit Hilfe von existierenden Webtechnologien. XML3D ist eine Erweiterung von HTML5 und nutzt damit verknüpfte Webtechnologien wie Cascading Style Sheets (CSS) und das Document Object Model (DOM). Darüber hinaus entwickeln wir neuartige Konzepte wie *a*) ein minimales abstraktes Modell für die 3D-Szenenbeschreibung, *b*) eine allgemeinere Beschreibung von dynamischen Effekten auf der Basis eines deklarativen Datenflussgraphen, *c*) einer plattformunabhängigen programmierbaren Materialbeschreibungssprache, *d*) der Kapselung von 3D-Inhalten, die sich bei der Instanziierung konfigurieren lassen, und *e*) einem Containerformat für die effiziente Übertragung von binärkodierten 3D-Inhalten zum Browser.

XML3D und die XML3D Implementierungen wurden erfolgreich in unterschiedlichsten 3D-Webanwendungen evaluiert, u.a. in einer Anwendung für die Visualisierung von Produktdaten aus dem Automobilbereich. Schließlich diskutieren wir die generelle Auswirkung unseres Ansatzes auf die Strukturierung von 3D-Webanwendungen.

Acknowledgments

This thesis is the result of a long journey and could not have been completed without so many people that have contributed and given their support.

First of all I would like to thank my supervisor Philipp Slusallek. Only his trust and courage made me start this large research project. His support made this work and thesis possible. And finally, his excitement for research and contagious passion kept me going.

Thanks to the members of the XML3D team for their collaboration and support: To Felix Klein, a trusted companion on the way and man behind the Xflow approach, to Jan Sutter, who was a great sparring partner, implemented Blast and was a great help with shade.js. To Christian Schlinkmann who helped transitioning the WebGL implementation from pure research code to a solid piece of software, to Sergiy Byelozyorov, who developed the Chromium implementation, and to Dmitri Rubinstein, who implemented RTSG. Many thanks also go to the many students who helped implementing XML3D or related topics: Nina Istomina, Boris Brönner, Johannes Ebersold, Aref Ariyapour, Patrick Keller, Stefan John, Benjamin Friedrich, and many others.

To Hilko Hoffmann, thanks for your encouraging and fatherly guidance at critical times.

Thanks to the public and private funders that enabled me to conduct research in this exciting area. This is a privilege! Also I would like to thank the DFKI, Saarland University, and Intel Visual Computing Institute for providing such a great environment for my work.

Special thanks to the colleagues participating in the EU-project VERVE. The results achieved in this project were breathtaking. In particular the XML3D-based training application for Parkinson patients with its wide user acceptance, the controlled triggering of freezes in the application, and (subjective) improvement of the patients' conditions with respect of freezing was a rewarding and unique experience that I would not want to have missed.

Not at least I want to thank my family. This work would not have been possible without their patience and support. In particular thanks to my wife Kirsten: I love you!

Contents

1 Introduction

1.1 3D Graphics and the Web

Interactive 3D graphics technology has become a commodity: Graphics hardware support has been integrated into all new desktop CPU, mobile devices, and even cars. Most new TV-sets are expected to be 3D-stereo capable and VR hardware such as head-mounted displays have entered the consumer market. At the same time, Internet connections come with bandwidths able to support large content and realtime remote interaction.

The possibilities that come with ubiquitous 3D graphics technologies, however, remain unavailable to non-experts: A deeper understanding of the underlying hardware including the rasterization pipeline and its programmable stages is required in order to use recent application programming interfaces (APIs) for graphics. The limited amount of graphics experts (and the related costs) makes 3D graphics remaining a broadcast medium where applications and content is produced by only a few specialized companies – mostly for games.

This is in stark contrast to web content. In the course of the Web 2.0 [O'R05] phenomenon, the web transformed from yet another distribution channel with content produced and packaged by professional journalists to a platform where user generated content (wikis, blogs and microblogging, social networks, etc.) plays a major role. This transformation applied to content such as text, images, videos, and open source software. However, it did not yet apply to 3D content.

One reason for this is the challenge creating appealing 3D content and authoring interactive 3D scenes. Another important reason is that there is no suitable and widely accepted format for the distribution of interactive 3D content on the web. The web, on the other hand, consists of ubiquitous standards, including HTML5 [W3C14c] as a markup language for content and structure, CSS [W3C09a] as a definition language for the style, the DOM [W3C09c] as a representation of the hierarchical structure of the web document, and JavaScript [ECM11] for client-side DOM scripting. Together with asynchronous access to remote content and services through the XMLHttpRequest API [W3C09f] and Representational State Transfer (REST) [Fie00] APIs, these technologies comprise a new application platform for the Internet. This web technology stack is well-known to millions of developers who created a large and lively ecosystem with thousands of libraries and frameworks that make the development of web applications even more convenient and productive. This, in turn, makes web development available to almost everyone. Even small businesses turn to web-based applications and cloud computing. In addition, the commodity of web technologies allows rapid uptake of new technologies.

On the other hand, no such environment exists for 3D graphics. Despite many efforts the use of graphics technologies remains fragmented: a wide range of different platforms, APIs, applications, and data formats exists. This makes it impossible to create, share, and

experience 3D content in the same way as e.g. ubiquitous video, which exploded due to its integration into the web by YouTube since around 2005.

With the advent of WebGL [Khr09], hardware accelerated 3D rendering has finally arrived in web browsers. WebGL is a wrapper to the low-level graphics API OpenGL ES 2.0 [Khr07] that allows drawing into the HTML <canvas> element from JavaScript. Providing access to the graphics hardware is a big step forward towards 3D web applications. However, expert knowledge is still required in order to develop web-based 3D applications. Additionally, WebGL is not integrated with other web technologies such as CSS or DOM scripting. Finally, WebGL is neglecting recent developments in alternative rendering technologies such as ray tracing and global illumination approaches that achieve interactive frame rates at a much higher image quality. A suitable high-level representation for interactive 3D content that is accessible to web developers and tightly integrated into the web environment is still missing.

1.2 Research Questions

This thesis is motivated by the following focus question: Is it feasible to integrate interactive 3D graphics as first class objects into the web technology stack?

Given the situation outlined above, the background to this question is the assumption that if we are able to positively answer this technical question, we will make a large step towards democratization of 3D graphics technology by *a*) enabling millions of web developers to use 3D graphics with minimum effort, *b*) making 3D graphics part of the powerful web environment with thousands of libraries and tools available, and *c*) providing a format to publish and share interactive 3D content easily.

The superordinate research question can be broken down to a number of partial questions that we will address in this thesis:

How does a web-integrated 3D scene description look like? In order to integrate 3D graphics tightly into declarative web technologies such as HTML5 and CSS, we require a declarative scene description for 3D graphics. Since existing approaches are not originally designed for the integration into HTML, they do not support important web concepts. Additionally, existing approaches include some concepts that are unknown to web developers, but which could be replaced by concepts that already exist in the web context.

The question that arises is how to design a declarative 3D scene description that *a*) is compatible to existing web technologies, *b*) reuses existing web technologies wherever this is possible and appropriate, and *c*) avoids the introduction of concepts that are unknown in the web community unless absolutely necessary.

How to design a lean yet flexible 3D scene description? A common approach in 3D scene descriptions (but also in scene graph libraries) is to "bake" specific data structures for important use cases into the format (e.g. for animations or virtual humans). However, mostly there is no "on size fits all" solution appropriate for the majority of use cases. Thus, the user is forced to convert back and forth between the data suitable for the application and the data structure imposed by the scene description. Additionally, the number of newly introduced nodes for specific use cases increases constantly and thus challenges

implementations as well as users. For instance, the X3D specification [Web04a] contains 252 different nodes in version 3.3, many of which are very specific to a small set of use cases.

The question that arises is if we can find concepts for a generic and lean scene description that at the same time increases the flexibility and thus functionality compared to existing approaches.

How to describe highly dynamic effects in a declarative approach? Interactive websites use JavaScript to modify the DOM, the data structure of the web page. In general, DOM modifications are changes in the DOM's structure or changes of element attributes. With the 3D scene description being embedded in the DOM, users can approach dynamic 3D content in just the same manner.

However, many dynamic effects in 3D graphics require computationally expensive operations, for instance on vertex attributes of meshes (e.g. skinning of organic objects based on their skeleton) or pixels of images (e.g. modifying textures or extracting features from video streams). Traditional (sequential) JavaScript is not suitable when processing e.g. vertex or pixel data, especially because many operations may exploit the options for parallelism offered by today's CPUs and GPUs. Thus we require an approach to describe dynamic effects in a way it can be mapped to various hardware architectures offered by the heterogeneous devices that web applications run on today. Additionally, the questions arises if it is possible to not only accelerate predefined operations but to also expose data parallelism to users in order to exploit it for applications-specific dynamic effects.

How to describe the material of a 3D object? HTML elements are styled using Cascading Style Sheets (CSS) [W3C09a]. CSS comes with a predefined set of properties describing elements such as the appearance of 2D objects and the page's layout. In contrast, realtime 3D graphics technology made the transition from predefined shading based on a configurable fixed-functionality hardware pipeline to highly flexible shading with multiple programmable stages. On recent GPUs, the shading of individual pixels can involve virtually arbitrary parameters accompanied by a routine describing the shading process – a so called shader. However, currently abstract scene descriptions are caught in a dilemma: Either they limit their expressiveness by sticking to predefined material models neglecting the need for highly customized appearances, or they allow integrating shader code in a platform-specific language and thus sacrifice renderer and platform independence.

Additionally, today's shaders are highly specialized and therefore strongly tied to the used render engine as well as to 3D content specifically designed for the shader. As such, shaders are not suitable to be used as general material descriptions, which can be organized in libraries and shared across the Internet.

In order to be able to match the capabilities of imperative approaches we think it is essential to provide users the possibility to customize the shading of 3D objects to a larger extend than previous approaches. Thus the question is how to achieve highly customizable yet general material descriptions that are renderer independent.

How can we compose 3D scenes? Images and videos are resources that are usually not directly embedded within the web page. Instead, these binary objects are kept external and

referenced from within the web page. The content of such externally referenced resources is immutable. This mechanism is not suitable for 3D resources, which typically have a structure or parameters that need to be modified by the application during runtime. The required level of access depends on the application's needs, i.e. to what extend parts of the scene need to be modified. The levels can range from

a) no access at all (e.g. a static subscene), through

b) access to the transformation hierarchy (e.g. moving a rigid object around) or

c) access to a subset of parameters (e.g. an excavator that needs to be animated using an animation parameter), to

d) access to individual vertex attributes of a mesh (e.g. for authoring tools).

Another common use case is instancing slightly altered variations of the same 3D resource multiple times.

A naive approach would be giving the user full access to all data. However, this would require having all data inside the DOM, which results in unmanageable large documents with bad start render time. Additionally, it prevents any kind of optimization because nothing can be assumed to be static. The question arises how we can provide mechanisms to allow developers fine granular composition of 3D scenes on different levels and from multiple internal and external sources.

How to deliver 3D content to the client? If we integrate 3D graphics into client-side web technology we have to deliver – analogous to HTML – all content to the client. However, 3D content generally exceeds the size of classical web content such as text, images, or scripts. Moreover, it is not easily streamable like video, which can be streamed based on a linear arrangement of 2D images. However, a progressive loading of 3D scenes is essential in order to avoid long load times that would ruin the user experience.

In contrast to video, there are no established compression schemes and no established container formats for 3D resources – mainly due to the inhomogeneous structure of 3D content. Solely for textures a number of competing texture compression formats exist. Since the device's graphics processor will typically support only one specific texture compression format, the delivery process has to adapt to the capabilities of the device. HTTP [FGM*99] offers several mechanisms for content negotiation, the process of selecting the best suitable representation based on the client's capabilities. Content negotiation is not limited to textures, but is also useful to adapt the representation format of other 3D resources, e.g. for choosing a suitable level of detail (LOD) or compression format for meshes and animations.

Research is required in order to find a delivery format that exploits the communication APIs (e.g. XMLHttpRequest (XHR) [W3C09f] and WebSocket [FM11]) available in the browser, provides means for flexible compression and can be streamed to the client. Additional, such a format has to compatible with content negotiation mechanisms available in HTTP.

1.2.1 Out of Scope

The research questions above define the conceptual framework of this thesis. Answering these questions are the main goal of this thesis. A general and sharable 3D scene description builds the foundation for democratizing 3D graphics. However, answering these questions obviously does not cover all aspects involved in making 3D content a natural component of the web. In the following, we present a selection of equally important research questions that, we consider out of scope for this thesis:

How to enable non-experts to create or acquire 3D content is an important research question that is not addressed in this thesis. We assume that 3D content can be created by artists using digital content creation (DCC) tools, exists in product databases, or can be acquired with professional 3D scanners. Additionally, there is a range of new hardware and promising research that we expect will shift 3D content creation from the professional to the consumer market.

In this thesis, we declare web developers to be the target group for the developed technologies. Our aim is to make 3D graphics accessible to this group by making them "feel at home". We assume that this objective can be accomplished by proposing a range of novel 3D technologies that integrate seamlessly into existing web technologies. We present a number of applications in order to evaluate our results in Chapter 9. However, we do not present a quantitative evaluation, which is left to future work.

1.3 Contributions and Thesis Structure

The main contributions, as presented in this thesis, are as follows:

1. We propose XML3D, a novel interactive 3D scene description that is designed as an extension to HTML5 and thus fully integrates into the web technology stack. XML3D reaches a new level of integration, in particular with CSS and DOM libraries such as jQuery[1]. Additionally, XML3D is the first scene description with a generic approach to data sources that fits very well to recent graphics APIs. Another novelty in XML3D is the consistent approach to internal and external resources. It allows for a very fine-granular scene composition on various levels. These aspects of the thesis were published in [SKR*10, SS11a, KSJ*12, SSK*13, KSSS14, SSS15].

2. We present Xflow, a novel declarative approach to dataflow graphs. Xflow provides composing functionality from small, generic building blocks. Xflow describes what should be computed rather than how to compute it in terms of language primitives. The latter is left up to the implementation that can merge the operations and map subgraphs to be computed exploiting available APIs. This way, Xflow exposes data parallelism to the user. Using dataflow analysis, we can map Xflow data flow graphs to the vertex shader stage of the GPU, to Parallel JavaScript [HHSS12], and other platforms and APIs. This part of the thesis has been published in more detail in [KSJ*12, KRS*13, KSRS13].

3. We present shade.js, a novel high-level material description language. Its interface to the lighting subsystem is based on radiance closures, which makes the material description agnostic to the rendering algorithm used. shade.js is based on JavaScript and exploits its polymorphism and introspection into the current execution environment to support adaptable and thus more general material descriptions. The accompanied compiler resolves the polymorphic types based on the input data and generates native shader code specifically typed and optimized for the target rendering system and algorithm. This part of the thesis has been published in [SKSS14].

4. We developed requirements for a more general 3D delivery format [SS13]. Based on these requirements and findings we got from the evaluation of existing approaches we propose Blast, a novel container format for the transmission of structured binary encoded data. It is particularly designed for (but not necessarily limited to) the web. Blast uses a code-on-demand approach to support arbitrary compression schemes. Additionally, it allows receiving collections of 3D resources in a single HTTP request. This reduces the number of requests necessary to load a scene composed from multiple resources, which can be the bottleneck for the transmission time. Blast consists of self-contained chunks. As a result, it can be streamed and decoded in parallel. This part of the thesis has also been published in [DSR*13, SS13, SSS14].

The combination of these four contributions comprises the XML3D architecture, which largely answers the research questions posed in Section 1.2. Obviously, the design and implementation of the XML3D architecture was collaborative work with colleagues and

[1] `https://jquery.com/`, last accessed 20 August 2016.

students from the German Research Center for Artificial Intelligence (DFKI), the Computer Graphics Lab at Saarland University, the Intel Visual Computing Institute (Intel VCI), and the University College London (UCL).

In particular the ongoing work on Xflow is research conducted by Felix Klein (Intel VCI) for his PhD thesis. Hence, we will discuss Xflow on the conceptual level only, which we elaborated jointly. For more details on Xflow, please refer to the corresponding publications. The work on Blast is research conducted in collaboration with Jan Sutter, PhD student at Saarland University and DFKI.

This thesis is organized as follows: Chapter 2 introduces the necessary foundations for our work and conceptualizes the used technologies. In Chapter 3, we develop design criteria that we then use to evaluate existing approaches in Chapter 4. We describe the XML3D architecture in Chapter 5 that was developed based on these design criteria. The novel concepts of the XML3D architecture are the core of this thesis. In Chapters 6 to 8 we discuss the related technologies in more detail. We show some applications developed with XML3D in Chapter 9 and evaluate our proposed architecture in Chapter 10. Chapter 11 concludes this thesis.

2 Fundamentals

Naturally, we will reference a lot of standard web and graphics technologies in the remainder of the thesis. However, since we address two virtually distinct disciplines that each developed rather independently of the other, we cannot take it for granted that the reader is familiar with both worlds. Hence, we will briefly recall some of the fundamental technologies that drive the web on the one side and 3D graphics and rendering on the other side.

2.1 Web Technologies

The Web has evolved from a unidirectional mainly text-based information source to a fully interactive and ubiquitous runtime environment providing a rich-media user experience. At the same time, also pressured by the rise of mobile apps, it has also evolved to a fully-fledged development platform.

Numerous technologies exist for the development of the server-sided functionality of a web application (back end). In contrast, the application protocol for communication over HTTP [FGM*99] and the client-side technologies are standardized so that they can run in any compliant web browsers. HTML5 [W3C14c], the DOM [W3C09c], CSS [W3C09a], JavaScript [ECM11] and related markup languages and APIs comprise the foundation of the so called web technology stack (see Figure 2.1).

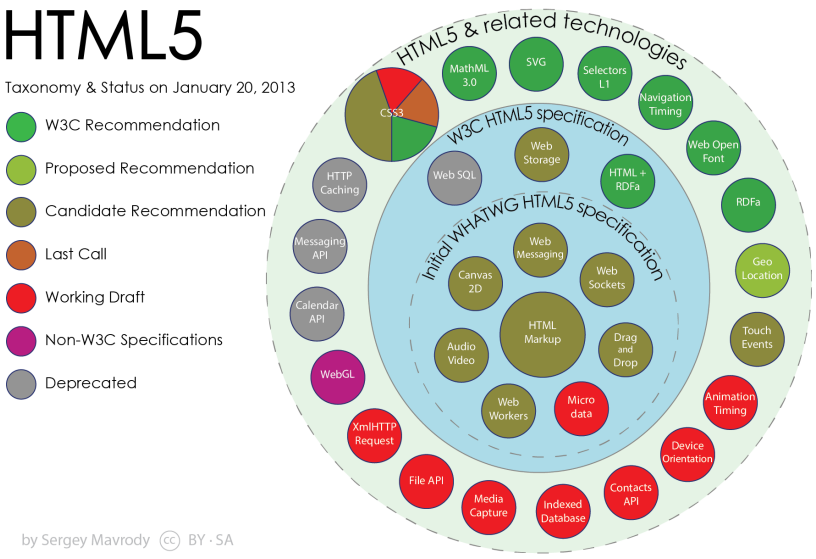


Figure 2.1: Taxonomy and status of HTML5 and related APIs and technologies.

2.1.1 HTML and the WWW

The *Hypertext Markup Language* (HTML) is a declarative language to describe web documents. The *Hypertext Transfer Protocol* (HTTP) is an application protocol to communicate documents and other data between clients and servers. An HTML document may reference other HTTP resources via links, which are identified and located on the network by *Uniform Resource Identifiers* (URIs) [BLFM98].

These three technologies (HTTP, HTML, and URIs) paved the way for a distributed information system of interlinked hypertext documents. Today, this system is commonly known as the *World Wide Web* or simply as *the web*.[1] It was first proposed by Tim Berners-Lee et al. [BL89, BLC90] and Berners-Lee also developed the initial versions of all three technologies. Thus, he is known as the inventor of the World Wide Web.

An HTML document consists of text, marked up with *elements*. Elements structure the text or add multimedia content by referencing external resources such as images and videos. The HTML specification defines an abstract language with two concrete syntaxes: The HTML syntax bears a close resemblance to SGML [ISO68], but is a separate language with its own error-tolerant parsing rules. The XHTML syntax, in turn, is based on XML [W3C08] and uses the more strict XML parsing rules.[2]

Web browsers are software applications that retrieve and present, i.e. render, HTML documents and their related content. Since most of the client-side web standards are declarative, it is up to the browser *how* the web content is actually rendered, leaving room for innovations and technological development.

Since its very beginnings, the web has undergone multiple evolutions that changed the web experience significantly. For instance, the web made a transition from a mainly text-based medium to an interactive multimedia experience including images, 2D vector graphics, audio and video. Many of the web's evolutions were strongly linked with newly introduced technologies. We discuss the most important ones in the following sections.

2.1.2 CSS

The initial versions of HTML contained elements to describe the style of content, e.g. <font> and . Additionally, there was an increasing trend to (ab)use tables to influence the layout of a web page [Lie05].

The introduction of the style sheet language *Cascading Style Sheets* (CSS) [W3C09a] allowed strict separation of content and style: The HTML document describes the content of a web page including its hierarchical structure. The presentation of this content can be described separately using CSS. CSS provides a fixed set of properties to configure layout, colors, fonts, etc. These properties can be applied to the elements of the HTML document. CSS style rules can be defined within an HTML document or within external style sheet documents that can be referenced from an HTML document. Using external style sheets

[1] We follow the recommendation of many publishers and the Chicago Manual of Style to use the term *web* (decapitalized) in the remainder of this thesis.

[2] Initially we defined XML3D as a language that is embedded in its own XML namespace within an XHTML document. Then we proceeded to design XML3D as extension to HTML's abstract language and independent of the used syntax. Hence, the name **XML**3D is not very meaningful anymore. However, we kept it for consistency across publications.

allows web designers to modify the layout consistently across a large number of web pages by modifying the style sheet alone.

With CSS, the style of an HTML document is orthogonal to its content. The applied styles and applicable style sheets can be exchanged at generation time on the server, at loading time, and at runtime. CSS allows adapting the style not only to the wishes of the readers and authors of a web page, but also to the capabilities of the display device and the browser [LB99]. For instance, the presentation of the content can adapt to different screen-sizes and different rendering methods (e.g. screen, print, and tactile devices).

2.1.3 DOM and Scripting

The content and layout described by HTML and CSS respectively are mostly static. Therefore, Brendan Eich developed JavaScript, which was first shipped in Netscape Navigator 2.0 in 1995 [Eic05]. JavaScript adds client-side interactivity to a web site and therefor allows building "rich" web applications, i.e. web applications with many of the characteristics of desktop application software.

Microsoft was the first to provide a general interface to the web page's structure, exposing virtually every HTML element as a scriptable object. Each modification would trigger a reflow of the page [Goo02]. These two characteristics are the basis of interactive web pages as we know them today.

Due to the "browser wars" in the late 1990s, there were significant differences in the scripting languages and document object models of the existing browsers at the time. Today, JavaScript is standardized as ECMAScript [ECM11] and the W3C standardized the Document Object Model (DOM) [W3C09c], a platform- and language-neutral interface that allows scripts to dynamically access and update the content, structure, and style of HTML, XHTML, and XML documents. It also includes a standardized event model and common events, known as *DOM events* [W3C00]. All recent browsers support one of the latest versions of JavaScript, DOM, and DOM events.

Note that such a common data structure and event model is unique in computer science: The DOM is *the* data structure of the web and thousands of available libraries operate on the DOM and implement certain aspects of an application. Due to the common data model such libraries can be combined and can interoperate easily. In contrast, other programming environments typically require converting data structures for the interoperability between different libraries.

2.1.4 SVG

Scalable Vector Graphics (SVG) [W3C09d] is a declarative language to describe interactive two-dimensional vector graphics. It is an open standard developed by W3C since 1999.

Initially developed as a standalone and purely XML-based file format, recent versions of SVG integrate other W3C standards such as DOM, DOM events, and CSS. HTML5 [W3C14c] allows SVG to be be freely mixed with other content in a single document. Today, the majority of browsers support SVG. The tight integration with HTML and the declining support of Adobe Flash[3] on mobile platforms helped SVG to gain popularity. SVG is for instance used for interactive data visualizations with D3 [BOH11].

[3] `https://www.adobe.com/products/flash.html`, last accessed 20 August 2016.

2.1.5 Ajax

Asynchronous JavaScript and XML (Ajax) is an umbrella term for technologies used to create asynchronous web applications. The *XMLHttpRequest API* (XHR) [W3C09f] is the major new technology behind Ajax: Instead of the traditional model of providing a link to another resource, a resource can be requested via the XHR API without navigating to a new web document. This request runs asynchronously, i.e. the application execution does not wait for the response. The server returns the requested resource and the content can be used to modify the web page, e.g. to add new content to the DOM. Other than the name suggests the returned resource can be of arbitrary format, including plain text, XML, JSON, and binary data.

XHR allows for a new communication model between server and clients and therefore a new web application model that closes the gap between web applications and traditional desktop applications in terms of their responsiveness [Gar05]. XHR is therefor considered as one of the technologies that paved the way for the web evolution know as "Web 2.0" [O'R05].

2.1.6 Web APIs

Browser vendors and the W3C consortium are continuously pushing the evolution of the web. This is also driven by the pressure to keep up with the capabilities of competing stacks – in particular those of mobile application platforms such as Android or iOS. This led to a series of new JavaScript Web APIs, some of them depicted in Figure 2.1.

Not all of these APIs are immediately available in all browsers. For those browsers that do not provide an expected functionality natively, so called *Polyfill* [LS11] implementations may emulate the missing functionality using JavaScript or any other technology (e.g. Flash). A Polyfill may emulate just a single API function but some Polyfills implement whole standards. For many new standards, a Polyfill implementation serves as reference implementation or for prototyping (e.g. [W3C14e]).

JavaScript is the *lingua franca* of the web. Despite its sometimes rather strange syntax, JavaScript has become one of the most widely used programming languages. JavaScript can be described as safe, portable, rapid prototyping friendly, and deterministic [HHSS12]. However, JavaScript has remained mostly sequential and provides few means to take advantage of today's parallel hardware.

Web Workers [W3C15b] is an API for concurrent actor-style threads that use message passing as the synchronization mechanism. Web Workers are widely adopted and allow offloading long running computations to background threads. However, they are "not suitable for the development of parallel scalable compute intense workloads due to their high cost of communication" [HHSS12].

Beside Web Workers for task parallelism, there are several approaches on different abstraction levels to expose data parallelism to web developers, including

1. WebGL [Khr09] for GPU access thorough JavaScript providing an API that conforms closely to the OpenGL ES 2.0 API,

2. WebCL [Khr14b], a JavaScript binding to OpenCL, which provides a compute language to harness GPU and CPU parallel processing,

3. Parallel JavaScript [HHSS12], an API for JavaScript that provides data parallel operations on arrays, e.g. *map*, *scatter* and *combine*, and

4. SIMD.js [MFM*14], an API that provides an abstraction over several SIMD instruction sets.

Note that WebGL has been widely adopted, whereas the other approaches are still in an early development stage.

WebGL and WebCL are the first Web APIs that require a good understanding of the hardware to be used efficiently. The introduction of hardware-related APIs comes at the cost of porting issues such as performance portability, up to the point where writing different kernels and shaders for different classes of devices becomes necessary [MRR12]. This is a very novel concept for the web. Similar, using vector instructions requires specific knowledge in suitable memory layout and masking and packing to allow control flow in SIMD-fashioned programming. As a result, we expect very few applications to directly use these new APIs. Instead, most applications will use libraries that provide data-parallel operations on a higher abstraction level.

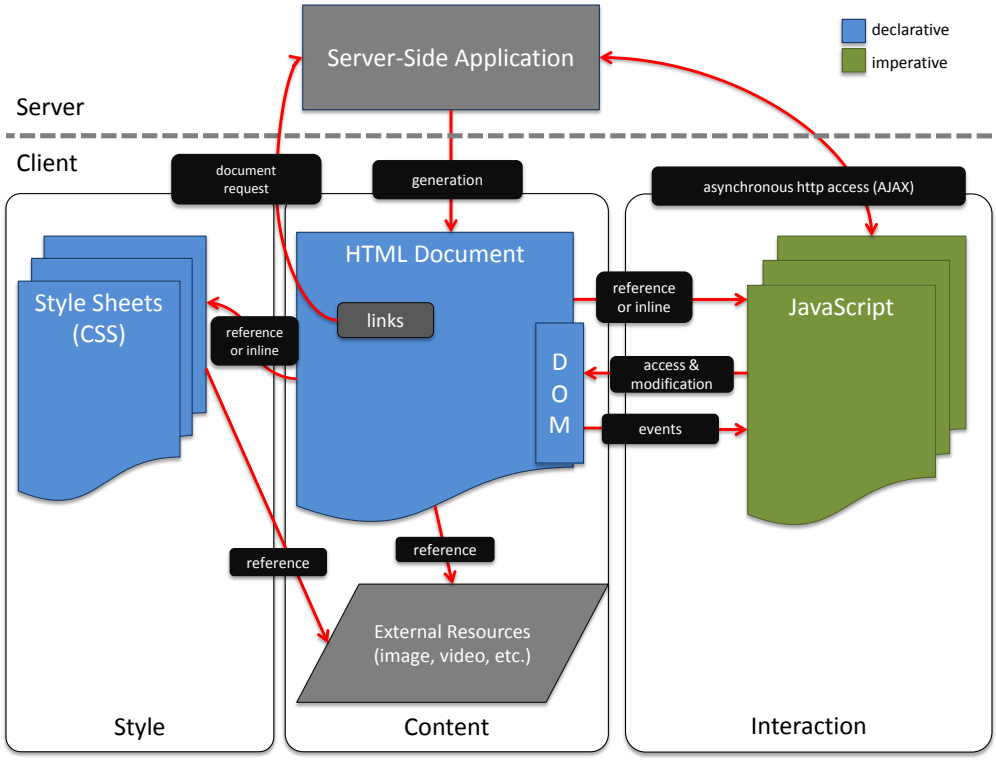


Figure 2.2: Structure of an HTML5 web application. The server generates the content HTML file which in turn references external resources with heavy payload (images, video, etc.), CSS documents for style definitions, and JavaScript source files for interaction. Using JavaScript, the application monitors user interaction, modifies the DOM, and loads additional content from the server asynchronously.

2.1.7 Web Components

Web Components[4] is an umbrella term for the set of four new web technologies that aim at providing a standard component model for HTML. This set includes *i*) Custom Elements [W3C14a], a method enabling definition of new DOM elements (including a custom API), *ii*) HTML Imports [W3C14b], a way to include and reuse HTML documents in other HTML documents, *iii*) HTML Templates, to declare reusable DOM subtrees in HTML (part of HTML5 [W3C14c]), and *iv*) Shadow DOM [W3C14d] definition of elements in a hidden DOM.

Web Components can be used to encapsulate functionality in custom elements and to hide the inner complexity of that element similar to the facade pattern in software engineering. The resulting components are composable and allow constructing applications from building blocks.

Web Components lend themselves perfectly to implement XML3D and convenience functionality on top of XML3D, but they were not yet available when we developed XML3D. However, we will discuss the use of Web Components at several places in this thesis including its relevance for future work (Section 11.2).

2.1.8 Summary

In summary, web applications consist of a declarative part for content and layout (HTML, SVG and CSS) and an imperative part for dynamics and interaction (JavaScript, DOM, XHR, and other APIs). Figure 2.2 depicts the relations of these technologies in a typically web application.

With SVG, the web technology stack has a declarative language for 2D vector graphics that is well integrated into HTML5. The HTML5 <canvas> provides an API for drawing 2D graphics into a canvas using JavaScript. The <canvas> element is also used by WebGL to enable hardware-accelerated 3D graphics using JavaScript. In contrast, as Figure 2.3 illustrates, a DOM-integrated language for 3D graphics is still missing.

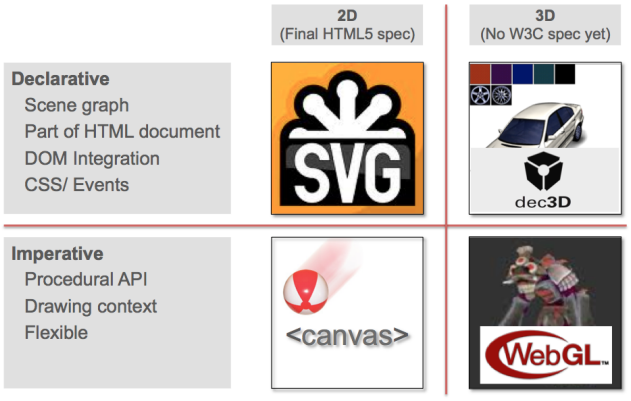


Figure 2.3: 2D graphics and 3D graphics web technologies: SVG and the Canvas API are W3C standards for 2D graphics, WebGL is an imperative API for 3D graphics standardized by the Khronos Group. In turn, a declarative 3D standard is missing. Image reprinted from [JRS13].

[4] `http://webcomponents.org/`, accessed 20 August 2016.

2.2 3D Graphics

For the 3D graphics part, we give a brief overview that has a strong focus on the aspects relevant to this thesis. If the reader is interested in more in-depth details, we recommend reading [FvDF13], [AMHH08], and [PH10].

2.2.1 Rendering

In computer graphics, rendering is the generation of a 2D image from a virtual 3D scene. Typically, a 3D scene consists of 3D models, light sources, and a virtual camera. 3D models are often described using a boundary representation approach, most commonly as a set of geometric primitives such as triangles. Alternatively, a 3D model can have a representation based on volumetric primitives, e.g. based on voxels.

Material properties can be considered as parameters of a 3D model. However, since *a material* is a self-contained real-life concept and since materials may be applied to more than just one 3D model, materials are usually described as independent objects inside the scene.

Numerous algorithmic strategies to render a 3D scene exist, the more popular of which include ray tracing [Whi79], path-tracing [LW93, Vea98], radiosity [GTGB84], photon mapping [Jen96], and various rendering algorithms that exploit hardware rasterization (see next section).

In general, these rendering algorithms vary in following characteristics:

Rendering Speed While some rendering algorithms focus on generating images in realtime, others focus on the generation of high quality images and therefore may not be suitable for realtime rendering (offline rendering). The rendering time decreases with increasing compute power, increased availability of parallel hardware, and improvements and optimizations of the algorithms. As a result, rendering algorithms that were exclusively used for offline rendering in the past, can now be used for interactive rendering or even for realtime rendering (e.g. realtime ray tracing [Wal04]).

Hardware Support In the past, rendering algorithms based on rasterization were considered as HW-accelerated rendering whereas all other algorithms were considered as software rendering. However, there is virtually no renderer which would not accelerate rendering by utilizing parallel computing in some way. For instance, renderers can exploit the options for parallelism on recent CPUs including vectorization units for data parallelism and multiple cores for task parallelism (e.g. [Ben06]).

In general, all rendering algorithms offer options for parallelism requiring logical separate computations for instance for each (sub-)pixel, tile, ray, vertex, primitive or light.

Supported Primitives Geometry-based rendering algorithms support some solid geometry representation scheme, most commonly surface meshes. In contrast, volume rendering algorithms are used to render volumetric models.

Realism For realistic rendering, rendering algorithms attempt to solve the rendering equation [Kaj86]. Different rendering algorithms have different strategies to approximate the integral equation. Therefore, they may take into account only the direct

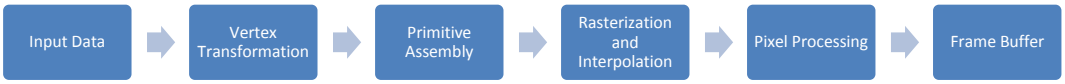


Figure 2.4: The pipeline stages of the rasterization rendering algorithm.

illumination of surfaces by light sources (local illumination) or consider light/surface interactions of the entire scene (global illumination). Additionally, rendering algorithms may or may not simulate optical effects such as reflection, refraction, scattering, and dispersion phenomena.

Generality Not all rendering algorithms are able to render all kinds of scenes equally well. In turn, some rendering algorithms need rendering parameters or even the content to be (manually) adapted in order to achieve good rendering results.

For instance, rendering algorithms designed for the GPU often suffer from restricted access to the entire scene due to insufficient GPU memory and restrictions in the APIs. As a result, global scene information such as shadows maps and reflection maps are typically created in separate rendering passes. However, these maps contain an originally continuous signal that needs to be sampled during the early rendering passes, and bandlimited and reconstructed during the main rendering pass. This process introduces errors that will lead to visible artifacts if the sampling parameters are not adapted to the scene, which is – despite many advances – often still a manual process.

2.2.2 The 3D Graphics Rendering Pipeline

In computer terminology, a pipeline refers to a series of stages where the output from one stage is fed as the input of the next stage. The stages of such a pipeline are often executed in parallel. Using functional decomposition, many rendering algorithms can be described based on a pipeline pattern, and many graphics pipelines have been proposed (e.g. [CCC87, PBMH02, Sch06]).

However, today the term *graphics pipeline* is frequently used as a synonym for pipelines on graphics processor units (GPUs) that support Z-buffer rasterization-style rendering. These pipelines can be controlled by APIs such as OpenGL [SA94] and Direct 3D [Bly06]. Figure 2.4 depicts the functional stages of the rasterization algorithm. Hardware acceleration started at the end and successively worked back up this pipeline [AMHH08]. For instance, the first 3D graphics cards available for consumer PC hardware (3dfx's Voodoo card, 1996) supported hardware rasterization including texture mapping and z-buffering. The second generation (Nvidia's GeForce256, 1998) also moved the vertex transformation stage to the GPU.

For many years, the graphics pipeline was an application-configurable *fixed-function pipeline*. With the increasing demand for more flexible shading, GPU vendors began to provide programmability at different stages of the graphics pipeline. The vertex processing stage and the pixel processing stage were the first two stages open to programmers. More programmable stages followed gradually (see Figure 2.5).

GPUs not only evolved from configurable to programmable graphics pipelines. At the same time, the APIs became more generic and the execution model changed. Prior the transformation to programmable multi-purpose processors (GPGPU), the input data for the data was very specific to the functionality of the pipeline. Nowadays, data is transferred to the GPU mainly in generic buffers and textures. Also the execution model changed dramatically. In the age of the fixed function pipeline, a rendering was the result of a single pass through the rendering pipeline. However, with the functionality to write into (multiple) buffers that can be immediately fed back as input of another pass, the graphics hardware pipeline became just a stage within a larger non-linear *rendering pipeline* (or render graph) where data-dependencies are handled by software, configuring and triggering multiple hardware passes. This allowed for dozens of multi-pass rendering techniques, including shadow mapping and post-processing.

Deferred shading [ST90] is a multi-pass rendering technique that "defers" shading to a stage after hidden surface determination, i.e. after z-buffering. Rendering techniques based on deferred shading store their shading parameters in buffers (G-buffers) and the shading is performed in a second pass in screen space. However, deferred shading is just a basic example for a variety of novel rendering techniques (e.g. [OA11, OBA12, HMY12]) that were made possible by the increased flexibility and generalization of the graphics rendering pipeline.

In general, modern programmable GPUs allow developing rendering systems tailored to the needs of a specific application or domain. The game industry carries this paradigm to the extreme. On the other hand, a highly customized rendering system requires very specific content often created by specialists, e.g. by game designers in the games industry.

In contrast, general-purpose rendering systems are often stuck in the fixed-function pipeline era. We will show in Chapter 4 that the data structures of many 3D file formats, scene graphs, and 3D scene descriptions are still modeled closely according to the functionality of the fixed-function pipeline and its configuration parameters.

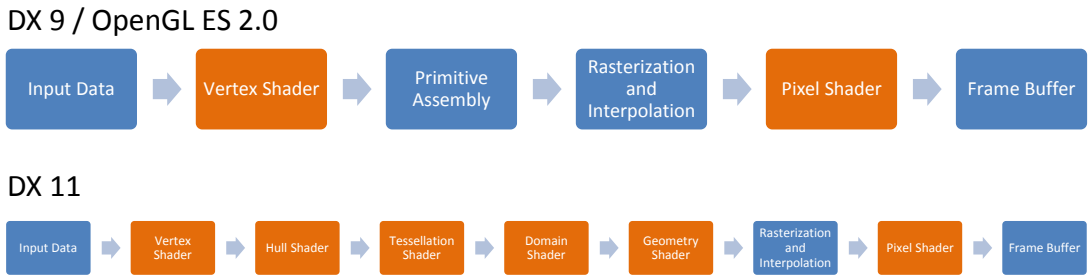


Figure 2.5: Schema of the DX9 / OpenGL ES 2.0 and DX11 pipelines. OpenGL ES 2.0 [Khr07], the OpenGL API for embedded systems, eliminates most of the fixed-function rendering pipeline and provides programmable vertex and pixel (fragment) shaders. OpenGL ES 2.0 is available on most Android and iOS devices and therefor also chosen as basis for WebGL. The DX11 pipeline provides four additional programmable stages.

2.2.3 Summary

While integration of 3D graphics into web technolgies is our primary goal, we still need to take into account the progress and trends in computer graphics. We can observe that rasterization hardware has become ubiquitous, but at the same time rendering strategies on this hardware are becoming increasingly heterogeneous. As a consequence, we think that XML3D should offer the ability to exploit the capabilities of common graphics hardware, but at the same time it should be abstract enough to allow various rasterization-based and global illumination rendering techniques.

On the other hand, similar to HTML, XML3D should be abstract enough to not only serve rasterization. Global illumination algorithms at interactive framerates are already available and become steadily faster. Compared to rasterization-based rendering, this family of algorithms is more general, i.e. they generate better predictable results because their algorithms model physical effects such as indirect illumination, reflections and shadows more reliably. Thus, we are convinced that these algorithms are better suitable for non-graphics experts.

Based on this and other requirements, we developed design criteria for XML3D which we discuss in the next chapter.

3 Design Criteria

In this chapter we discuss design criteria we think are important for a web-based 3D scene description. These criteria are partly inherent in the research questions we formulated in Section 1.2. However, we explicitly list them in this chapter for three reasons: Firstly we will use the criteria in order to evaluate existing scene descriptions and rendering system in the next section. We will also use these criteria as a guideline when developing the XML3D architecture in Chapter 5. Finally, we will review our design criteria when evaluating the XML3D architecture in Chapter 10.

3.1 Integration into the Web Technology Stack

One of the criteria directly derived from the research question is an in-depth integration of interactive 3D graphics into the web's technology stack. The integration should be as seamless as possible, i.e. 3D scene objects should become first-class objects on the web. Similar to HTML and SVG, the scene description should leverage technologies such as the DOM, the DOM event model, CSS, and JavaScript.

The integration criterion also implies to introduce concepts new to the web only if absolutely necessary. In particular, we do not want to introduce concepts if a similar concept already exists in the web technology stack. For instance, it would not make sense to introduce a 3D graphics specific event model if the well-known DOM event model – already being utilized in multiple web standards such as HTML and SVG – is sufficient.

A 3D technology that implements these two principles would be familiar to web developers and compatible with existing libraries (e.g. jQuery), server-side tools (e.g. CMS such as Drupal) and design patterns (e.g. REST, MVC) commonly used in web development.

3.2 Renderer Independence

Most of today's realtime 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 surface appearance, light emission, geometry detail, and others.

On the other hand, 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 [SSW06]. Additionally, major hardware vendors have started to consider ray tracing when designing new hardware [Nvi10] providing the option to use ray tracing as a possible alternative to rasterization for realtime 3D graphics in the future.

Consequently, and analogous to HTML5, the scene description should not be tied to a specific rendering algorithm. Instead, the abstraction level should hide all concepts specific for a set of rendering algorithms and should support GPU accelerated rendering

algorithms as well as upcoming algorithms, which may include for instance global illumination. On the other hand, for most scenes ray tracing performance is still not on-par with GPU-accelerated rasterization. Therefore, we require that the chosen abstraction levels do not compromise the flexibility, expressiveness, and performance (see criterion below) possible due to programmable hardware pipelines.

3.3 Usability

According to [ISO98], usability is the effectiveness, efficiency, and satisfaction with which specified users achieve specified goals in particular environments. In the context of this thesis, we distinguish between the 3D web application developer (author) and the user of the 3D web application (user). With respect to our research question, we focus on the authors of 3D web applications and hence on the effectiveness, efficiency, and satisfaction with which authors can create interactive 3D web applications.

Despite this focus on the author's needs, we must not lose sight on the user: Although it is in the responsibility of the author to create 3D web applications that meet the user's expectations, our approach should facilitate the creation of efficient and satisfying applications. This includes also the *user experience*. [ISO10a] defines user experience as "a person's perceptions and responses that result from the use or anticipated use of a product, system, or service".

Again, the quantitative evaluation of the usability of our approach (and the user experience gained from using our approach) is beyond the scope of this thesis. Instead, we discuss this aspect on a qualitative level and compare it with related approaches.

Learnability Learnability incorporates several principles such as familiarity, consistency, generalizability, predictability, and simplicity [Nie93]. With our approach, we try to address web developers. Hence, we assume that a seamless integration into HTML5 will increase the learnability of interactive 3D graphics compared to other approaches, in particular due to the familiarity of concepts. However, whenever *new* concepts are introduced, we require them to be consistent, general, and predictable.

Simplicity and readability are often mentioned design goals of VRML (cf. [CB97]). In this context, HTML was often referred to because it was simple enough to be authored by anyone using a text editor. However, if one studies the HTML source of a modest web application today such as the Google Gmail web mail client[1], it becomes apparent that applications tend to become hardly comprehensible with growing complexity.

Trying to push simplicity to far often leads to high-level abstractions that are easy to instantiate but hard to customize. In general, trying to make complex things too simple may lead to monolithic and therefor inflexible designs. Hence, a paradigm we want to follow during the design of XML3D is summarized by a quote attributed to Alan Kay: "Simple things should be simple; complex things should be possible".

Scene Organization Imagine a 3D web application that allows visitors exploring a virtual city or engineers planning a factory of the future. Such an application typically consists of hundreds or even thousands of objects. Each of these objects may in turn consist

[1] `https://mail.google.com/`, last accessed 20 August 2016.

of millions of polygons. It would be impractical and inefficient to have such a scene description in a single document.

Instead, we need to provide means to reuse existing 3D objects, functionality to hide the complexity of subscenes, and means to compose scenes from existing building blocks and from multiple documents. Efficient scene organization and software engineering have common demands: In software engineering, the reuse of components and hiding complexity behind facades are well-known patterns. Consequently, we want to provide authors means to apply these patterns also to 3D scene descriptions.

Performance We assume that XML3D is used for fully interactive web applications within the browser with high realtime requirements and not for offline rendering or editing of 3D geometry. Consequently, a key requirement for our approach is the possibility to implement it efficiently in terms of runtime performance, memory performance, and power consumption. Today, virtually all computers are parallel computers providing parallelism in various ways, including multi-core CPUs, many-core, GPUs, and vector instructions. For our approach it is essential to address all these forms of hardware parallelism in a manner that is abstract enough to avoid limiting the implementation to a specific platform or rendering algorithm. At the same time we want to leverage optimally the hardware capabilities in order to achieve realtime rendering performance and thus a fast and fluent user experience.

Algorithms in 3D graphics offer various options for parallelization. We can exloit data-parallism for per vertex processing which is required for instance for skinning of virtual humans. Also processing images can be parallelized and is required for instance for Augmented Reality applications. However, we not only want to exploit parallel hardware internally in the implementation of XML3D. To a certain extent, we require XML3D to also expose options for parallelism and in particular data parallelism to the user.

Web applications may run on computing devices from embedded systems and smart phones to high-end super computers. Given this heterogeneity of web clients and our requirement on platform independence, it is a challenging task to develop appropriate abstractions in order to describe data processing in way general enough to serve the majority of use cases. On the other hand we need to keep those abstractions still mappable to very specific APIs that expose different options for parallelism.

In addition to runtime and memory performance, another performance factor is the page loading time. The 3D content, which is typically larger than pure 2D content, needs to be delivered to the client as efficiently as possible. Additionally, to provide good user experience, it is required to give the user visual feedback that the web server is responsive, i.e. our approach should allow for fast start render times. This can be achieved for instance by asynchronous loading of content, prioritized loading of the most important content, streaming of content and showing progress on partial data already, and by level-of-detail (LOD) concepts, which present a coarse representation of the content first until the whole content has been loaded.

3.4 Expressiveness

Providing more and more means to highly customize the stages of the graphics pipeline using shaders, programmable GPUs have enabled graphics experts to achieve functionality, flexibility, and visual quality not comparable with those in the days of fixed-function pipelines.

As we will show in the next chapter, existing scene descriptions as well as scene graph APIs are either tied to a specific rendering algorithm or – if they endeavor to be abstract enough to be renderer-independent – lag behind in their expressiveness in terms of functionality, flexibility, and visual quality.

In order to remain competitive with domain-specific solutions and solutions tied to a specific rendering algorithm, we aim to push the expressiveness of XML3D to a level beyond what previous approaches achieve.

Functionality We require functionality comparable with similar scene descriptions and scene graph APIs. This includes features such as lighting using different lighting models, various texturing possibilities, shadows, rigid body and skeletal animations, object identification, callbacks for user events, and means for scene management.[2]

We aim for concepts general enough to be useful for the majority of use cases. We do not want to introduce domain-specific functionality into XML3D. For instance, game engines typically offer non-graphics features often required for the development of games, e.g. artificial intelligence (AI), physics simulation, and multi-user environments. However, we do not consider these features to be within the scope of XML3D.

However, we require XML3D to be designed in a way that the functionality can be extended. For instance, it should be possible to add and handle domain-specific functionality. It should be possible to add important aspects such as AI and physics in an orthogonal way. Orthogonality allows aspects to be extended and implementations to be replaced without affecting other aspects.

Flexibility Though a declarative abstract scene description can never compete with an imperative approach in terms of flexibility, we want to provide increased flexibility compared to previous declarative approaches. Therefore, we require an appropriate abstraction level. For instance, we do not want to introduce abstractions for specific effects such as skinning or morphing. Instead we want the system to be flexible enough to achieve these effects with its base functionality. Then, authors can collect such effects in a library and reuse and adapt such effects for their applications.

For the development of applications it is important that the scene description is flexible enough to adapt to the application's needs (and not the other way around). This includes the definition and composition of data structures from different sources, application-specific interfaces to 3D models, and unrestricted access to all parts of the scene that needs to be manipulated by the application.

There is a wide range of different ways to describe dynamic meshes, animations, and materials. Hence, our approach should not offer a single monolithic abstraction for these

[2] We discuss the functionality of related 3D toolkit libraries in more detail in Section 4.2.

aspects. Instead it should provide means to describe the aspect in a way that best suits a specific application.

Visual Quality 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 generate photo-realistic images. This includes effects such as 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.

3.5 Conclusion

Based on the criteria described in this chapter we developed XML3D. As for most architectures, we have to handle conflicting design criteria. For instance, the higher abstraction level implied by the renderer-independence and HTML integration may conflict with high performance and flexibility. A common approach is to go for a trade-off between conflicting requirements. In this thesis, we will apply techniques (e.g. using compiler construction and dataflow analysis) for achieving great flexibility and performance without sacrificing renderer independence.

Similar, there may be conflicting requirements between simplicity and providing features and flexibility. As already mentioned, we want to resolve such conflicts by making simple things simple but complex things still possible.

4 Related Work

We aim for an abstract, declarative, and interactive description of 3D content. Therefore, our work is loosely related to 3D graphics file formats which are not interactive, but may aim for an abstract representation of 3D content. Hence, we give a short overview on common 3D graphics formats and their characteristics in Section 4.1.

Due to its interactivity, XML3D is also related to 3D toolkit libraries which likewise aim on easing the implementation of interactive 3D applications providing higher abstraction levels compared to direct use of a specific graphics API. Thus, we will discuss 3D toolkit libraries including scene graph concepts and the more specific game engines in Section 4.2.

X3D and its better known precursor VRML are the only previous approaches for an declarative and interactive scene description. Additionally, X3D and VRML have been designed with web integration in mind. Hence we discuss them in more detail in Section 4.3.

With the advent of WebGL in all common web browsers, many 3D libraries for this specific context emerged. In Section 4.4 we discuss approaches of several WebGL-based 3D libraries including X3DOM, an integration model for X3D into HTML.

More specific related work such as shading languages (Section 7.1), data delivery formats (8.5), and integration of physics simulations into scene descriptions (6.3.1) are discussed within the corresponding chapters.

4.1 3D File Formats

There are hundreds of 3D file formats and it would go beyond the scope of this thesis to discuss all of them. Therefore, we restrict ourselves to the most common formats and those that are most relevant for our approach.

In general, one can distinguish between 3D file formats designed for

1. restoring a runtime state of a 3D tool or library (serialization format),
2. exchanging 3D content between different tools retaining as much information as possible (exchange format),
3. delivering 3D content to a client application in order to display the content within that application (representation format).

With respect to this classification, we consider XML3D as a representation format.

Serialization formats are in general application-specific and cannot be used beyond their main purpose. Hence we will not elaborate further on these kinds of formats. Instead, we discuss a selection of exchange and representation formats in the following:

OBJ *Object File* (OBJ) is a 3D file format originally developed by Wavefront Technologies (now Autodesk). It is restricted to geometry and a set of predefined material

models. Nevertheless, the format is still widely used to exchange assets between 3D digital content creation (DCC) tools. OBJ supports the definition of a fixed set of vertex attributes including coordinates, normals, and texture coordinates. These attributes can be used to construct static polygons, lines, curves, and free-form surfaces. OBJ can reference external material libraries (MTL). The MTL material format [RRT95] supports a set of material parameters to configure one of the ten predefined material models (e.g. based on Lambertian shading, Blinn's specular illumination model [Bli77], Whitted's illumination model [Whi79]).

PLY The *PLY Polygon File Format* is a format developed at Stanford University for storing static polygons represented by vertices, faces, and edges. A noteworthy feature of PLY is the possibility to store arbitrary vertex attributes represented in one of eight built-in data types. This feature allows storing additional per-vertex information such as colors, ambient occlusion, or quality measures. Therefore, the format is still very common in the 3D scanning community.

STL STL is a simple 3D file format created by 3D Systems in 1987 and has become a de facto standard in computer-aided manufacturing and rapid prototyping [CNW99]. A STL file describes a static 3D surface represented by triangles. However, the triangle representation is restricted to (positive only) coordinates and a face normal. There is no support for colors, materials, etc.

FBX Originally the native file format for Kaydara's Filmbox, FBX has evolved to the main format to exchange assets between DCC applications. In particular, it is the standard import/export format between Autodesk products. The FBX format is proprietary, but owner Autodesk provides an SDK that can read and write FBX files.[1] FBX files store data about cameras, lights, meshes, NURBS, etc. It supports animation of object transformations and skeleton based animations including skinning. Two material models are provided: *FbxSurfaceLambert* describes a surface with ideal diffuse reflection based on the Lambertian model [Lam60]. *FbxSurfacePhong* extends this model by adding the specular component of the Phong reflection model [Pho75]. Additionally, FBX can store the specific implementation of a material by attaching hardware shaders for a specific platform (e.g. HLSL [Mic02] and GLSL [KBR03]).

COLLADA Similar to FBX, COLLADA [Khr08] is designed as an exchange format for 3D assets. In contrast to FBX, COLLADA is an open and vendor-neutral standard format, originally initiated by Sony Computer Entertainment in 2003, but now managed by the Khronos Group consortium (Khronos). Among others, it supports geometry (mesh and boundary representation), materials, animations, lights, morphing, skeleton-based skinning, and physics. However, since COLLADA aims for lossless exchange between DCC tools it has a vast amount of features including various alternatives for representing geometry. As a result, most software systems support only a subset of COLLADA.

Similar to FBX, COLLADA materials may reference one of four simple material models: A constantly shaded surface that is independent of lighting (constant), an ideal

[1] `www.autodesk.com/fbx-sdkdoc-2014-enu`, last accessed 20 August 2016.

diffuse material model (lambert), and specular materials based on the Phong reflection model (phong) or on the Blinn-Phong reflection model (blinn) [Bli77]. Alternatively, it can configure the rendering pipeline based on one of three predefined platform profiles (CG, GLSL, GLES) that include specific shaders, passes (defining render states), etc. Obviously, specific shaders and the definition of passes tie the asset not only to a platform, but also to a specific rendering system that provides the rendering algorithm (see discussion below). Hence, most DCC tools support is commonly restricted to the common material models and, hence, exchanging advanced materials is not possible.

glTF With the *GL Transmission Format* (glTF) [Khr14a], Khronos addresses the issue that formats designed for exchange of 3D assets are not suitable for transmission and representation at runtime. For instance, exchange formats such as COLLADA need a "significant amount of processing to transform the data structures designed to retain information into a representation appropriate for graphics APIs" [Khr14a].

glTF represents the structure of a scene including a scene hierarchy, materials, and lights in a JSON file. Data defining geometry and animations, which make up the bulk of the payload, are stored in a second binary file, referenced from the JSON file using views (i.e. using byte offsets and lengths). For vertex data, for instance, these views can be used as vertex buffer objects in GL without any additional processing required.

glTF has many designs borrowed from COLLADA. For instance it uses a similar model for describing materials based on techniques that contain passes and shaders. In contrast to COLLADA, glTF allows for binding shader parameters to light parameters. However, the referenced shaders are still tied to a specific rendering algorithm and a change in the amount of lights or lighting models will not be considered in the shader.

Version 1.0 of glTF is in draft. It does not define compression for geometry and other rich data. However, geometry compression is planned: At the time being "the 3D Formats Working Group is developing partnerships to define the codec options for geometry compression" [Khr14a]. Although designed as a network-based representation format, glTF 1.0 has no concepts for streaming 3D content.

JT JT [ISO12] is a format designed for the exchange of 3D product information typically exported from CAD systems or product data management systems. The JT data format contains amongst other information the product structure and geometry at different levels of detail. Geometry can be presented using polygons, NURBS, or primitives. Additionally, the format may contain metadata, in particular product manufacturing information.

Similar to COLLADA, JT has a primitive material model based on the Phong reflection model or can alternatively use platform-specific shaders, which again ties the content to a specific application. Texturing is only available in combination with shaders.

When we look at the above list, it becomes apparent that with the existing formats the exchange of structure, static geometry, and keyframe-based rigid body animations between

applications works reasonably well, but that the exchange of other dynamic effects and materials remains an unsolved issue.

Dynamic Effects Dynamic meshes are supported by FBX and COLLADA using processing algorithms that interpolate between two or more meshes (morphing) or interpolate the mesh based on an (animatable) skeleton, where each vertex has a weighted association to the joints of that skeleton (skinning).

However, due to the variations in data structures that exist for these two approaches, the exchange without loosing information remains challenging. Dynamic effects that go beyond morphing and skeleton-based interpolation, e.g. enveloping, corrective shapes, volume-preserving simulations, and cloth and flesh simulations, are not supported by these data formats.

All of these procedural effects require data processing. It is however neither practical to prescribe fixed-function procedures and data structures nor to include the required operations as source of a specific imperative language in the file format. Hence, a different approach is required.

The open graphics interchange format *Alembic*[2] approaches this issue by storing samples of a dynamic scene at specific times and therefor baking these kinds of dynamic effects into the file format. Alembic is mainly used in the film industry, where these sampled results can be used in various systems for lighting and rendering. Obviously, the sampled scene includes the *results* of a dynamic scene, but is not dynamic by itself. Hence, it contradicts our requirements on interactivity.

Material Dilemma In the context of this thesis, we define a *material* as a visual concept that defines the lighting of (sub-)pixels of a surface, i.e. the material configures the lighting. Most 3D data formats support different sets of predefined lighting models based on variations of the Phong lighting model: Typically, it is possible to configure the specular lighting (e.g. Phong or Blinn-Phong) or to omit specular lighting entirely (Lambert materials). Additionally, some formats offer to configure transparency, reflectance, and refraction. However, the configuration of the lighting models is typically restricted to a single value for each coefficient, to a single vertex attribute, or to a single texture look-up.

However, software rendering, programmable hardware shaders, and shade trees allow the lighting to be configured virtually arbitrarily. Common use cases require flexible configuration of the lighting such as:

- Arbitrary linear combination of lighting models.[3]
- Variation of lighting *parameters* based on vertex attributes and/or textures. For instance, a scalar temperature value stored per-vertex could be mapped to a color to visualize the temperature distribution along a surface.
- Variation of lighting *models* based on vertex attributes and/or textures. A common approach in 3D modeling is to decode surface properties such as rustiness or worn

[2] `http://www.alembic.io`, last accessed 20 August 2016.

[3] For physical correctness, the combination has to be linear. However, for many applications, artistic freedom is more important than physical accuracy.

edges in vertex attributes. In some of these cases, not only the parameters but even the lighting model may vary within the material.

- Procedural variation of parameters. For instance some natural materials such as wood or marble can be achieved by varying lighting parameters procedurally.
- Blending of textures. All lighting parameters may depend on one or more texture map look-ups. The look-up may depend on multiple uv-coordinate sets and the results of the look-ups may be mixed in various ways (texture blending).

The sheer diversity of useful materials cannot be expressed with a small set of material models.[4]

Consequently, some file formats such as COLLADA and FBX support a hybrid approach: It is possible to store the configuration of one of the predefined material models and additionally (or alternatively) it is possible to include custom GPU shaders as an implementation of a material.

Obviously, using GPU shaders ties the content of the file not only to rasterization but also to a specific rendering API and version (e.g. to DirectX 9.0c). But that is not the only reason why custom GPU shaders are not suitable to describe materials. In the following we discuss three key principles we require for general material descriptions and that are violated if custom GPU shaders are used as a replacement for a more abstract material concept: [5]

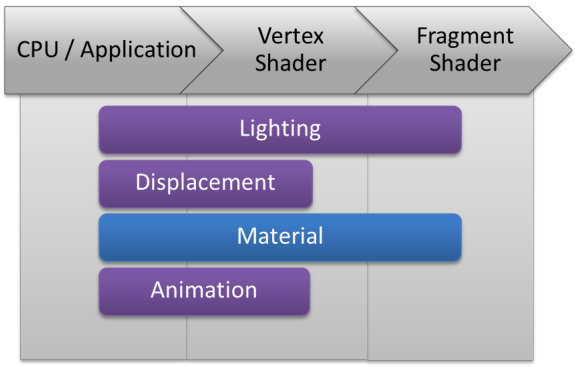


Figure 4.1: Logical concerns cross-cut a DX9 rendering pipeline being implemented in the rendering application on CPU, in the vertex shader, and in the fragment shader. For systems that offer at least one of these concerns based on generated shaders, allowing the user to replace one of the stages by a custom shader will likely break the system's implementation of the concern.

Separation of concerns We require that a material is a self-contained concept that is independent of other concerns and thus can be shared, e.g. in a library. However, a set

[4] This was also one of the reasons why GPUs made the transition from more and more configurable fixed-function pipelines to pipelines with programmable stages (Section 2.2).

[5] In Sections 4.2 and 4.3 we will show that also most 3D toolkit libraries as well as the scene description X3D allow using custom GPU shaders in order to implement material aspects. Hence we discuss the issues with this approach in some more detail here.

of GPU shaders (for instance a vertex and a fragment shader for a DX9-like pipeline) implements both, a superset and a subset of the material: Using GPU shaders enforce users to decompose logically distinct features into per-pipeline-stage procedures (c.f. [FH11]). For instance, the implementation of a material logic typically involves transformation of per-vertex parameters in the vertex shader stage, routing the results through possible other stages and doing per-fragment calculations in the fragment shader. However, uniform expressions, i.e. expression that do not change per geometry batch, are typically implemented within the application and executed on CPU. Additionally, a material might require a specific configuration of the rendering pipeline, for instance activating alpha-blending for semi-transparent objects.

As a result, each shader stage implements not just the material concern, but also multiple other concerns. For instance, the fragment shader implements parts of the material and lighting, the vertex shader implements animation, and parts of material and lighting (see Figure 4.1). However, data formats and 3D toolkit libraries typically provide abstract models for lights and animations, and rendering systems implement these models by generating shaders for a specific execution environment (shader compositing). As a result, custom shaders (provided by the format or by the user) put at risk that the rendering system's concepts will not work for objects that have those shaders assigned.

Generality Materials should not only work for a specific asset, but should be general enough to work with multiple assets. However, GPU shaders have fixed input signatures, i.e. they expect a fixed set of parameters with a specific type and name. This includes per-geometry (uniform) and per-vertex parameters, textures, and other input buffers. Hence, today's GPU shaders are closely interwoven with assets that provide the expected parameters and is not adaptive enough to handle assets with other parameters.

Typically, rendering systems adapt the shader's input signature and logic based on preprocessor directives or by composing the shader from small code snippets. However, this adaptation mechanism is not available for custom shaders or ties the GPU shader to a specific application that knows how to set the directives based on the actual available input parameters.

Renderer-agnostic Materials should not depend on a specific implementation nor on a specific rendering algorithm. However, GPU shaders expect transformations, light and animation parameters in a specific format and with a specific name. An application has no means for reasoning about the semantic of the parameters. In fact, the content is tied to an implementation that provides scene parameters exactly in the way the GPU shader expects them.

The same applies for the output of the shader. Executing shaders in a render pass results in one or more modified buffers. The semantic of the content of the buffer remains "the secret" of the shader author. For instance, a classical forward shading algorithm may write the result of the shading process as a color into the framebuffer. Another shader, designed for a deferred shading approach [ST90] might write intermediate results into multiple buffers that serve as input of subsequent rendering

passes. Consequently, the shader is not only tied to a specific implementation, but also tied to a specific algorithm to render the scene. Also, the application needs to know the intended algorithm in order to setup the render pipeline the right way (e.g. connecting output buffers with inputs of subsequent render passes).

Considering these aspects, embedding GPU shaders in file formats may be useful if the content is used in a single fixed setup only (for instance in a production pipeline within a game company), but are not suitable as an approach for describing general materials that can be used in multiple different setups.

4.2 3D Toolkit Libraries

With the advent of 3D graphics APIs also 3D graphics toolkits libraries emerged in order to provide users a higher level of 3D graphics programming. Similar to 3D file formats, it would go beyond the scope of this thesis to discuss all available 3D toolkit libraries. Also, there is a rich design space for these kinds of libraries. For instance, some libraries just aim to ease the integration of C-based API calls into object-oriented programming languages. Effect frameworks such as CgFX[6] provide functionality to configure the rendering pipeline including passes and render states. Scene graph libraries in turn try solving multiple aspects of 3D application development.

Scene Graph Concepts

Open Inventor [Str93] and Iris Performer [RH94] are the two major realtime interactive rendering systems that have decisively influenced the design of later scene graph libraries. They also introduced a number of concepts that can generally be found in scene graph libraries:

Multi-purpose In contrast to domain-specific graphics libraries such as game engines, scene graph libraries try to address a larger set of use cases. Scene graph libraries are often used as graphics back end for simulation engines, medical applications, virtual heritage applications, etc. Also, scene graph libraries are often designed to run on various platforms, including desktop applications, touch-screen installations, and virtual and augmented reality environments.

Abstract Model Instead of drawing collections of primitives, scene graph libraries define a conceptual model that provides a simplified and abstract view of a complex reality. The abstract model provides virtual objects for real-world objects and effects, i.e. for shapes, materials, lights, and cameras. These models are often idealized or designed to fit well to the underlying rendering algorithm.

Scene Graph Scene graph libraries provide a data structure that describes a hierarchical relationship of the objects. This hierarchy is typically also a transformation hierarchy, i.e. the placement of an object in the scene is relative to its parent placement. In most libraries, the objects of the scene are represented as nodes in a directed acyclic graph (DAG), where the edges describe the hierarchy. This data structure is commonly known as *scene graph*.

[6] `http://www.nvidia.com`, last accessed 20 August 2016.

API Scene graph libraries provide an API to manipulate the scene graph at runtime and to perform specific operations that require traversing the scene, e.g. searching for objects or serializing the scene, but also for rendering.

Event-model Scene graph libraries provide means to capture occurring events, e.g. user events (e.g. mouse motion and keyboard keys) or system events (e.g. resizing the window and timer-based events). The event model defines how the application can subscribe to these events, e.g. using some kind of observer pattern [GHJV95].

Object Identification Scene graph libraries provide means to compute the rendered object that is on a specific position on the screen (picking), e.g. below the cursor of the mouse. Additionally, the library provides a data structure to identify this object unambiguously. For instance, in systems where visible objects are organized in a DAG it requires a path to identify the picked object.

File Format Scene graph libraries allow (de-)serializing the scene graph, i.e. storing and restoring the initial state or any subsequent state during runtime.

Scene Organization Scene graphs and related objects tend to become very large and complex. For this reason, scene graph libraries provide mechanisms to compose scenes from smaller resources. The provided mechanisms range from simple inclusion of local file resources that contain a part of the scene to fully fledged component models that encapsulate a reusable set of related functionality (e.g. a 3D widget that contains geometry and functionality to rotate a target via events).

Semantics The scene graph hierarchy represents the parent-child relationship between scene objects. However, this is only one of multiple relations scene objects may have. Scene graphs may provide means to annotate object semantically to be able to build secondary data structures (sets, graphs) based on these annotations. A simple concept that can be found in many CAD applications are *Layers*, that allow to group logically related objects in a secondary structure. The user can then for instance hide or high-light the objects assigned to a specific layer.

Extensibility Scene graph libraries offer adding new features for the case that new functionality cannot be composed from existing functionality. For instance a library can provide a plug-in system to define new scene graph nodes.

Parallelism Most scene graph libraries provide means to distribute work over multiple threads or multiple nodes of a cluster. This requires mechanisms for sharing and synchronization of resources.

Optimizations Tuning the performance should not be a concern of the application developer. Thus a scene graph library internally organizes the scene to achieve the best performance. Common optimizations scene graph libraries offer include sorting the objects to be rendered in order to reduce state changes in the API, lazy evaluation of scene graph changes, concurrent evaluation of sub-graphs, hidden surface determination in order to avoid unnecessary draw calls, and level of detail (LOD) handling.

Not all scene graph libraries offer necessarily all these concepts. However, to support *interactive* applications which accept and respond to user input, at least an event model and a way to modify the scene objects is necessary. If an application requires direct manipulation of scene graph objects, the scene graph library additionally needs functionality for object identification (cf. Section 10.2).

After discussion of the main common characteristics, we will now discuss the main differences and contributions of related scene graph approaches.

Handling States

The evaluation of states in Open Inventor [Str93] is order-dependent because states are inherited from previous states from other branches. Since changing of states is expensive on graphics hardware, the performance of Inventor scenes largely depend on the order of nodes within the scene graph. In IRIS Performer [RH94] (as well as in later proposals) the evaluation is order-independent. This allows Performer to reorder the rendered objects to minimize state changes. It also allows for designing the scene with respect to the application's needs rather than to some arcane performance criteria. Performer was also the first scene graph to introduce multi-threading. Therefor it separates the application logic from the rendering task and further splits the rendering task into several stages (e.g. calculation of intersections, culling, and drawing). These tasks can be evaluated in partially in parallel in pipeline style.

Scene Graph Access

OpenSceneGraph [BO04] designs the rendering as an operation with limited write operations. As a rsult, it allows multiple culling and drawing operations to run in parallel on multiple CPU's, which in turn can handle graphics contexts[7] from multiple graphics subsystems (for instance for stereo and VR setups). OpenSG [Rei02] generalizes multi-threaded scene graph access. Instead of just synchronizing scene graph structure changes, it replicates the scene graph structure and scalar data for each participating thread and implements a copy-on-write strategy for all data arrays. The system keeps a *changelist*; synchronization of the changed data happens when requested by the application. This way, multiple simulations can change data of the scene graph at different frequencies. The changelist mechanism is also used to distribute the scene graph over multiple nodes of a cluster.

Generic Data Concept

While in previous approaches each entity of the abstract model has a predefined set of parameters (modeled closely to the predefined parameters of the fixed-function pipeline), more recent rendering systems adapt to the functionality of recent graphics APIs, including definition of generic vertex attributes and shader parameters, and render passes which can receive input from previous render passes. OGRE (Object-Oriented Graphics Rendering Engine) is such a 3D software system designed to "make it easier and more intuitive for developers to produce applications utilizing hardware-accelerated 3D graphics" [JS06]. In

[7] A graphics context receives draw calls in order to render to a canvas. In stateful graphics APIs such as OpenGL it manages also a stack of state objects.

OGRE, *submeshes* define sets of vertex buffers which can be shared between multiple submeshes. Similarly, OGRE materials are composed from *techniques* which in turn can define multiple rendering passes. Developers can use *Materials Scripts*, a declarative language to define OGRE materials. Material Scripts provide an abstraction level that allows rendering OGRE scenes on various rasterization platforms including Direct3D and multiple versions of OpenGL.

Scene Abstraction Level

All rendering system presented so far are designed for exclusive use with GPU-based rasterization. While Open Inventor still provides an abstraction level that makes it hard yet possible to render the scene for instance using ray tracing [SSBM03], more recent libraries such as OpenSG, OpenSceneGraph, Ogre, and several Open Inventor derivatives[8] introduce rasterization-specific abstractions for the configuration of the rendering pipeline (e.g. shaders, passes, and blending modes) as well as abstractions that make no sense for other rendering algorithms (e.g. shadows).

In contrast, Doellner et al. [DH02] propose a generic rendering system – VRS – which decouples the scene graphs structure from the underlying rendering API. VRS provides a generic attribute management that handles rendering-system dependent and rendering algorithm dependent attributes. VRS strictly separates the declaration of the scene from its rendering: To interface to different low-level rendering systems, specific *handlers* adapt to the scene description and *techniques* implement different algorithms. The rendering back ends implemented in VRS include OpenGL, Radiance [War89], and RenderMan [Ups89]. However, VRS has predefined surface shading attributes and every renderer needs "to map the attributes reasonably" [DH02]. Alternatively, specific shaders for specific low-level render systems can be assigned using the generic attribute system. Also, Doellner et al. do not provide concepts for dynamic scenes other than rigid body animations.

Data Level Parallelism

In general, the discussed rendering systems provide task parallelism on different data sharing levels. However, they do not provide abstractions for data level parallelism. As a result, data level parallelism is not exposed to the user but only used to implement the predefined functionality of the scene graph.

For GPU-based systems, users may provide custom shaders to exploit options for parallelism in their application. However, this comes with the same issues already discussed for providing GPU shaders to describe advanced materials. In particular, users run at risk breaking functionality of the rendering system that is usually implemented in system-generated shaders.

There are few approaches trying to expose data level parallelism in scene graph libraries. Giden et al. [GMLP08] suggest the configuration of CUDA pipelines using a new set of Open Inventor nodes. A more integrated approach is proposed in [OKK09]. It offers abstraction and sharing of buffer resources between compute nodes and rendering nodes in an OpenSG scene graph. However, the approach is not abstracting over compute platforms but restricted to CUDA. Moreover, the output of the computation can be used as an input

[8] e.g. Coin3D (`www.coin3d.com`) and FEI Open Inventor (`www.openinventor.com`), both last accessed 20 January 2016.

buffer of a custom rendering pipeline, but is not otherwise connected with the higher-level abstractions of OpenSG.

Dataflow Approaches

Dataflow is a term in computing, that is used for hardware architectures (e.g. [DM75, KDK*01]). Derived from these architectures, *dataflow programming* describes a program as a network of operators that forward data to each other, where each operator is run once all of its input data from other operators is available. This paradigm fits well for modern multi- and many-core hardware that offers various options for task- and data-level parallelism.[9] Dataflow software architectures can be found in various domains, and in particular for multimedia processing [BHK*02, Mic12].

The combination of dataflow algorithms with computer graphics has a long history. The central architecture of systems such as AVS Express [UFK*89], Data Explorer [LAC*92], and VTK [SML96] are built around user configurable dataflow graphs (DFG), often editable via some user interface. These dataflow based visualization systems are mainly used for scientific and information visualization where the application data consists of large amount of generic input data from a discrete simulation or from sensor data (source). The source data is sent through a series of transformations that select or filter parts of the input data (filter), until finally mapped to a graphical representation (map) that is presented to the user (render). Visualization is usually an iterative process, e.g. changing the time step in the input data triggers recalculation of the whole graph.

In scene graph applications, the transformation of application data is typically done in the application code. Dataflow systems allow this transformation to be (partly) handled by the dataflow graph that models the required transformation based on building blocks.[10]

In high-level dataflow graphs, the input of each graph node is typically an abstract structured type. The various specializations of the abstract type require the node (which typically represents a specific operation) to branch inside the node depending on the actual data. In VTK for instance, the *vtkPolyData* class can represents arbitrary combinations of points, lines, polygons, and triangle strips. Most vtkPolyDataAlgorithm nodes accept only a subset of the available representations or need to diverge the control flow depending on the available representations. Although this high abstraction level makes it easier for the user to configure a dataflow graph, it reduces the options for global analysis and optimization of the data flow. Also, it becomes hard to merge multiple operations to eliminate overhead due to intermediate results. Thus these systems typically leverage data-parallelism only locally within a node.

CashFlow [KS06] is a proposal for a visualization framework that combines scene graph paradigms with dataflow paradigms. In contrast to VTK, CashFlow does not require explicitly connecting nodes to construct the stages of the visualization pipeline (source->filter->map->render). Instead, it uses Open Inventor's traversal and state propagation concepts to collect the stages and to construct the pipeline: Nodes that represent the data, filtering, and mapping can be placed anywhere in the scene graph and become part of the

[9] In fact, the dataflow graph pattern is a producer-consumer pattern very similar to a non-linear pipeline pattern as discussed for rendering pipelines in Section 2.2.

[10] For a comparison of different application models including our approach, refer to the discussion in Section 10.2.

traversal state. When during execution a specific render node is found, the visualization pipeline gets constructed from the states and rendered using the scene graph functionality. However, since CashFlow is based on Open Inventor and the state propagation of Open Inventor depends on the traversal strategy, the same issues apply as described above.

Game Engines Game Engines are domain-specific 3D toolkits with the obvious aim of supporting the creation of games.

In terms of concept and functionality, they are typically both, a superset and a subset of scene graph APIs: There is typically no strong focus on extending the game engine and most commercial game engines are closed-source software systems that permit extending the core functionality. On the other hand, they offer additional functionality commonly required for the development of games. This includes collision detection, physics simulation, sound, and artificial intelligence.

The game market is a billion dollar market[11] and still growing. As a consequence, the effort spent into development of game engines has increased as well. Hence, there are easily understandable arguments for using game engines for development of non-game 3D applications and thus leverage the tremendous effort spent in the development in these toolkits:

Visual Quality A main differentiating factor in competition between game engines is the visual quality. Hence, the techniques used for achieving visual effects are at the cutting edge and game engines outpaced "classical" general-purpose scene graphs in this field.

Performance In order to dedicate as much computing time for the game logic as possible, all built-in functionalities of a game engine are highly optimized. This includes extensive use of parallel computing techniques.

Authoring Tools Recently, game engine vendors not only provide the game engine as such, but also highly sophisticated authoring tools. Additionally, most vendors license their engines for a fraction of the former costs. Both is in particular true for the Unity Game Engine[12] and the Unreal Engine[13]. As a result, game engines are not only used for high quality games, but also for casual games and in novel fields such as virtual heritage, architectural visualization, and serious games. The success of game engines and their penetration into new application domains shows the importance of good authoring tools. Authoring tools are not addressed in this thesis, however, we are convinced that it would be possible to build web-based authoring tools on top of our work and initial work in this direction has been done [ZSC*13].

Cross-Platform Game engines such as Unity and Unreal are cross-platform game engines that support delivery to mobile devices, desktops, consoles, and recently also web browsers (we discuss this in more detail below). The rendering algorithms get specifically adapted to achieve the maximum performance on each target platform.

[11] 91.5 billion US dollar revenue in 2015 according to Newzoo market research. Source: `http://www.newzoo.com/insights/us-and-china-take-half-of-113bn-games-market-in-2018/`, accessed 06 January 2016.

[12] `http://unity3d.com/`, accessed 06 January 2016.

[13] `https://unrealengine.com/`, accessed 06 January 2016.

Besides these advantages, using a game engine beyond games also comes with some disadvantages:

Content Creation As discussed earlier, a key argument for using game engines is their visual quality. Game engine provide a set of visual effects, many of which rely on faked physical phenomena. Many of these effects are not universally applicable, game designers need to prepare the content following specific rules. In this case, specialists create specialized content for specialized game engines.

Dynamic Content Both, Unreal and Unity have a strict workflow for the creation of games: The content is prepared and configured with game logic in the game engine's editor. In a subsequent step, the content gets compiled for a specific platform in order to start the game. In this approach, all content needs to be determined at compile time. Functionality provided in the editor (including loading and optimization of geometry, editor widgets, etc.) are not necessarily available during runtime of the game as this is not a typically use case for games. However, for instance for a collaborative factory configuration, users may want to load factory parts from a database dynamically and use 3D widgets to place them in the factory. For these kinds of use cases the sophisticated tools from the game authoring environment cannot easily be reused.

System Integration The main disadvantage of using a native game engine as development platform is its monolithic design compared to the web development environment. For instance, consider the factory configuration scenario again: The visualization of and the interaction with the virtual factory are just small parts of the overall application. Other parts may include the connection to an Enterprise-Resource-Planning (ERP) system and other databases, the integration of machine components from third-party providers, and the connection to simulations and systems that verify the configuration as well as domain-specific constraints during runtime.[14] In a Dual Reality [LP10] setup, an application may integrate sensor-data from an existing factory.

In game engines, all this functionality needs to be implemented in the game engine's environment, i.e. using the engine's native language or in a supported scripting language. In contrast, networking, the integration of databases, coupling with multiple third-party services (mashups), collaborative set-ups, and 2D GUIs are an integral part of every web application and the web development environment provides a stack of technologies and software systems to achieve these tasks.

Also, many web applications already exist that would benefit from adding a 3D visualization component. It is not likely that these existing applications will be ported into a game engine environment merely for the 3D component.

Deployment The accessibility of web applications due to the ubiquity of web browsers cannot be reached with native game-engine based applications. Obviously, this not only applies to game engines but equally to all the other non-browser based 3D toolkits discussed above.

[14] We will discuss such an application in Section 9.4.

Figure 4.2: Compilation process for publishing Unity-based applications on the Web[15]. User code written in C# or other .NET based languages gets compiled to the Intermediate Language (IL). Then, the IL2CPP compiler translates IL to C++. The result and the C++-based game engine code is compiled to LLVM's intermediate representation (IR) using Clang. The emscripten compiler [Zak11] translates LLVM IR into asm.js, a subset of JavaScript, which can finally be executed in a JavaScript engine. The process for the Unreal Engine 4 is nearly identical but does not require compilation from C# to C++ as user code is written in C++ already.

As a result, game engine are often used in mainly content-driven application, e.g. in virtual heritage applications, architecture visualization, and training applications. In business domains that require the integration into a larger system landscape, for instance in the automotive industry, building information management (BIM), or fashion design, game engines could not establish yet.

Game Engine Web Deployment Both, Unreal and Unity offer deployment of games to the Web. Therefore, the game's code and assets get compiled to asm.js [Moz13] (see Figure 4.2), an "extremely restricted subset of JavaScript that provides only strictly-typed integers, floats, arithmetic, function calls, and heap accesses".[16] The performance of JavaScript within the valid asm.js subset can be improved by ahead-of-time optimization (e.g. Firefox) and other performance improvements. Some techniques such as multi-threading and SIMD are not yet supported by JavaScript and are consequently also not available when deploying game engine code to the web.

Performance measurements show that game-engine features which make intensive usage of SIMD and multi-threading (such as 3D physics for instance) are considerable slower in asm.js compared to native code. In contrast, user code written in C# is often considerable slower than its trans-compiled asm.js counterpart.[17] WebAssembly [Web16] will most probably replace asm.js as a more size- and load-time-efficient compile target for publishing native applications to the web. At the same time, browser and hardware vendors work on SIMD (e.g. SIMD.js [MFM*14]) and shared memory[18] support. Hence we expect that in the future performance will not be the bottleneck for 3D games web deployment.

On the other hand, the approach deploying 3D applications to the web using game engines comes with drawbacks similar to deployment to native platforms: For instance, assets are compiled into the asm.js and are always in (compressed) memory representation as soon as the code has been parsed. [19] Again, assets need to be defined upfront or application-specific user code is required in order to load dynamic assets. Also, the issue integrating a web deployed game into a larger system landscape is not resolved by this approach: The various transformations during the compilation process obscure function

[15]Source: `http://www.davevoyles.com/asm-js-and-webgl-for-unity-and-unreal-engine/`, last accessed 09 July 2016.
[16]Source: `http://asmjs.org/faq.html`, last accessed 09 July 2016.
[17]Source: `http://blogs.unity3d.com/2015/12/15/updated-webgl-benchmark-results/`
[18]See `https://github.com/tc39/ecmascript_sharedmem`, last accessed 09 July 2016.
[19]See `http://blogs.unity3d.com/2015/12/07/unity-5-3-webgl-updates/`, accessed 09 July 2016.

and module names and, hence, it requires special means to let the browser communicate with the game through application-specific interfaces and scripts that have been defined upfront.[20] Hence, a once compiled game can be considered as a grey box: The application including all assets are compiled into asm.js code, which is deployed to and runs in the browser's JavaScript engine, but is not otherwise integrated with web technologies.

Conclusion We conclude that creating 3D web applications using 3D game development environments such as Unity or Unreal becomes increasingly interesting even for applications outside their original domain: They come with sophisticated authoring tools, latest rendering techniques, rich feature sets including third-party libraries, and large development teams. On the downside, the focus of these game engines is obviously still on games, i.e. on applications which are content-wise self-contained. As a result, game engines are less suitable for applications which need dynamic editing functionality and loading of assets not determined at compile time. With respect to this, game engines are not flexible enough for general-purpose applications and do not meet our requirements on web integration. We expect that a standardized, natively supported and widely adopted HTML extension for describing 3D graphics would yield in authoring tools and rich sets of convenience libraries similar to what is available for HTML.

Still we aim at functionality similar to games engines. Hence, we will show in Section 6.3 how we can extend XML3D to integrate functionality that can also be found in game engines, e.g. rigid body physics and artificial intelligence. Eventually, we will take game engine functionality into account when we evaluate XML3D in Chapter 10.

[20] See `https://docs.unity3d.com/Manual/webgl-interactingwithbrowserscripting.html`, last accessed 09 July 2016.

4.3 X3D & VRML

The activities around VRML and X3D emerged from the idea of incorporating virtual reality environments into the web. Dave Ragget was one of the first to propose a platform independent 3D scene description based on web technologies as "mechanisms for people to share VR models on a global basis" [Rag94]. Ragget proposes a *Virtual Reality markup language* (VRML) for describing the structure of virtual worlds on a high abstraction level (e.g. buildings, floors, and rooms) using the experiences from SGML and HTML as basis. Elements in the scene should reference detailed external models via URL [BLMM94]. Ragget also proposes that "annotations define whether a door opens inwards or outwards". Ragget anticipates some of the XML3D concepts including tight integration into web technologies, external references, and annotation of semantics. On the other hand, Ragget ideas remain extremely vague and finally did not find their way into the VRML standard.

At the same time, Pesce et al. [PKP94] proposed extending "HTML to describe both geometry and space" and presented the first prototype for a 3D browser (which was not based on HTML however).

The developers of VRML decided against designing VRML as an extension to HTML because

> "... VRML requires even more finely tuned network optimizations than HTML; it is expected that a typical VRML world will be composed of many more 'inline' objects and served up by many more servers than a typical HTML document. Moreover, HTML is an accepted standard, with existing implementations that depend on it. To impede the HTML design process with VRML issues and constrain the VRML design process with HTML compatibility concerns would be to do both languages a disservice. As a network language, VRML will succeed or fail independent of HTML." (quoted from the VRML 1.0 Specification [BPP94])

At latest since the introduction of asynchronous server requests (XHR), the first part of this statement is very much outdated: Today, requesting for instance the mainly static front page of Amazon[21] leads to more than 250 additional server requests for external HTML and JSON documents, style sheets, and scripts from various servers.

From that point in time, VRML and its successors developed independently from other web technologies at a much slower pace. As a result, there is hardly any overlap with W3C technologies and – to this date – the technologies did not converge.[22]

Instead of extending HTML, VRML adapts a subset of Open Inventor's concepts including its nodes, fields, and file format syntax. VRML adds some networking capabilities; in particular objects can be hyperlinked to other worlds or HTML documents [HW96]. Whereas VRML 1.0 can only describe static scenes, VRML 2.0 [CB97] (latter dubbed VRML 97 [ISO10b]) adds events and sensors in order to respond to user- or time-initiated

[21] `http://www.amazon.com/`, measured 20 August 2016.

[22] At time of writing, Web3D Consortium standards incorporate no W3C technologies other than XML. X3DOM, discussed in Section 4.4.1, is the first proposal towards an integration into the W3C technology stack.

events, interpolators for animations, sound, scripting, and *Prototypes*, a component system for encapsulating the functionality of a subscene graph.

X3D [Web04a] is the successor of VRML 97. The major differences to VRML 97 include two new serialization formats: XML encoding and a binary encoding based on FastInfoSet [Tel05]. With the steady growth of VRML and X3D by more and more nodes, it became difficult for X3D browser vendors to implement the standard as a whole. Consequently, X3D added the concept of *profiles*: "A profile is a named collection of functionality and requirements that shall be supported in order for an implementation to conform to that profile" [Web04a]. As a result, not every X3D browser supports every X3D scene.

Apart from the encoding and profiles, the concepts are virtually the same as in VRML 97, i.e. they have not changed significantly over the last 18 years. Similar, the abstract model has been extended but has never been adapted or otherwise modified. As a result, some content from 1997 can still be rendered in a recent X3D browser.

On the other hand, in the same period of time graphics hardware has evolved enormously from fixed function 3D accelerators to massively parallel programmable general-purpose processors (see Section 2.2). However, X3D has no concepts to reflect the advances in graphics hardware and to profit from the new flexibility gained from programmable stages in the graphics pipeline. Consequently, X3D never gained the same attraction as VRML did when the scene model fitted very well to the fixed functionality of GPUs at that time.[23]

Nevertheless, we will discuss the concepts of X3D in the context of our design criteria in more detail. This is for two reasons: First, X3D is the only previous interactive abstract scene description that aims at renderer-independence. Therefor we often compared and assessed concepts in X3D in order to see which concepts we may adopt, adapt, or discard. Secondly, X3DOM [BEJZ09] is the only other approach proposing an integration of 3D graphics into HTML. X3DOM does not start from HTML, but tries to integrate X3D concepts into HTML and the DOM. Consequently, we will discuss the issues of X3D concepts with respect to recent 3D graphics and 3D application development in this section. In Section 4.4.1, we will discuss the additional issues that arise from transferring these concepts into the web.

Web Integration Although explicitly designed as a web format, the development of VRML and X3D was largely independent from the development of W3C web technologies. Hence, apart from the optional XML encoding, X3D is not incorporating any other W3C standard [Web04a].

The only integration model it offers for the integration of X3D content into web browsers is based on plug-ins. Here, the scene is stored and managed by the plug-in and can be accessed and modified by the web application using the Scene Access Interface (SAI) [Web05]. Explicit bindings are defined for JavaScript. However, the API is rather limited compared to APIs available in the web context (e.g. the DOM API).

The integration model based on plug-ins comes with two major drawbacks. First, plug-ins have limited acceptance due to the serious security issues of cross-platform plug-in APIs and due to their low penetration rate. Second, the application run within the plug-in

[23] After 12 years of X3D, VRML still tops X3D in number of search results on `www.google.com` (06 January 2016).

environment is decoupled from the DOM content and there are no bidirectional update or synchronization mechanisms. If 2D content from the web site and 3D content within the plug-in should interact, it is in the author's responsibility to connect and synchronize the events and application models. Therefore developers "have to deal with the small plugin-specific interface and its synchronization capabilities" [BEJZ09]. This approach conflicts with our requirement on a seamless integration into the web technology stack.

Finally, web developers need to learn both event models (DOM Events vs. X3D Routes) and both access interfaces (DOM API vs. X3D SAI). This conflicts with our approach to not introduce new concepts if not necessary: It is contrary to principles such as familiarity and consistency and therefore impairs learnability.

Renderer Independence The declarative scene description and the high abstraction level makes X3D independent from the used rendering algorithm. There are several approaches that show X3D scenes can be rendered using e.g. ray tracing [DWWS04, RGSS09] and path tracing [SVB*13].

Despite many proposals and vendor-specific extensions that violate the renderer independence of X3D including explicit shadow mapping [Kam10], explicit render passes [dC03], and handling of draw states [Gra03], the Web3D Consortium managed to keep the specification renderer-agnostic for most of its parts. A woeful exception is the *Programmable shaders component* [dCGP04], which was introduced in X3D V3.1 [Web06] and adds the concept of hybrid material models to X3D. This approach comes with all the issues that were already discussed in detail in Section 4.1.

Learnability Due to its higher abstraction level, creating simple scenes is simpler in X3D than using for instance a graphics API. However, X3D also violates some of the principles of learnability. For instance, the number of nodes grew from 54 nodes in VRML 1.0 to 246 nodes in X3D 3.3. These nodes describe graphics features on various levels of abstraction. For instance shading nodes vary from low-level shader nodes for programmable hardware (<ProgramShader>), over simple material model nodes (<Material>) to very specific high-level nodes, for instance a node that renders volumetric data in cartoon-style (<CartoonVolumeStyle>). The inhomogeneous abstraction levels within X3D's abstract model contradict the principle of consistency.

In addition, X3D has no uniform and consistent way to define data for its nodes. Instead, it offers four different ways to define unstructured data depending on *a*) whether or not the data is often shared between multiple nodes, and *b*) whether or not the data belongs to a node field with a predefined semantic.

In general, unstructured data is defined in *fields* with a predefined semantic, name, and type. However, field values can only be shared between nodes adding Routes into the scene to explicitly connect those fields that share data.

For this reason, some often shared geometric properties are *not* defined in data fields. Instead, X3D defines specific nodes for these properties in order to simplify sharing them using the DEF/USE concept: "Several geometry nodes contain Coordinate, Color or ColorRGBA, Normal, and TextureCoordinate as geometric property node types. The geometric property node types are defined as individual node types so that instancing and sharing is possible between different geometry nodes." [Web04a].

With the advent of platform-dependent programmable shaders in X3D 3.1, the abstract model was extended to include nodes that define vertex attributes with an arbitrary name (e.g. <FloatVertexAttribute> and <Matrix3VertexAttribute>). The semantics of these generic vertex attributes depends on their use in the attached shader.

Finally, X3D defines a concept for dynamic fields as required for instance for Prototypes and <Script> nodes. Dynamic fields are used for less frequently shared data that is required to have arbitrary names and types.

To sum up, X3D has *four* concepts to define and *two* concepts to share unstructured data. This is just one example that demonstrates where the principle of generalizability is violated in X3D. In contrast, the XML3D architecture has a single, generalized approach for the definition of unstructured data, which also allows for fine-granular sharing and compositing (Section 5.4).

Scene Organization X3D offers three concepts for subscene abstraction that help to keep the scene description concise:

Subscene instancing Using the USE/DEF mechanism, authors can instance a subscene that has been previously defined in another part of the scene. The instances may differ in their placement due to their different position in the transformation hierarchy. Otherwise, the instances are identical. Hence, this mechanism is very limited in its applicability.

Prototypes X3D Prototypes can be considered as *facade pattern* [GHJV95] because they hide the functionality of a subscene behind a facade. A <ProtoDeclare> node defines such a Prototype, which is not rendered on its own, but only if instanced elsewhere in the scene. Additionally, it defines the interface of the Prototype that is exposed to the user. The fields from the interface can be connected to fields in the subscene. This way, aspects of the subscene can be configured per instance.

Prototypes may contain <Script> nodes. As a result, Prototypes allow not only content to be encapsulated but also behavior. Thus, Prototypes are a powerful concept that help organizing a scene by the definition of reusable, self-contained modules. In fact, X3D Prototypes allow defining new nodes that are built from existing functionality.

Inline The *Inline* node embeds an external scene into the current scene. The imported scene has its own runtime and namespace: The nodes of the external scene are rendered and participate in interaction; however the nodes of the external scene are not exposed to the current scene unless their names have been explicitly imported into the scene using the IMPORT statement.

X3D provides subscene instancing which is limited in its application, and two concepts that are powerful, but – due to their possible internal complexity including Routes and scripting nodes that allow modifying the encapsulated subscene arbitrarily – too complex for a system to exploit the coherence of multiple instances. On the practical side, prototypes are hard to implement. Behr testifies that "it is almost impossible to get the behaviour in a consistent way. The concept is powerful but underspecified and a beast by combining sub-classing and aggregation in a single construct" [BH15].

As depicted in Figure 4.3, there is no instancing technique in X3D that would allow instancing of encapsulated scenes on a level simple enough to enable exploiting the coherence of instances and the instancing facilities of recent graphics APIs, but on the other hand still allows configuring individual parts of the scene including appearances and animations.

In Section 5.6, we will present *Asset Instancing*, our approach for the XML3D architecture to fill this important gap.

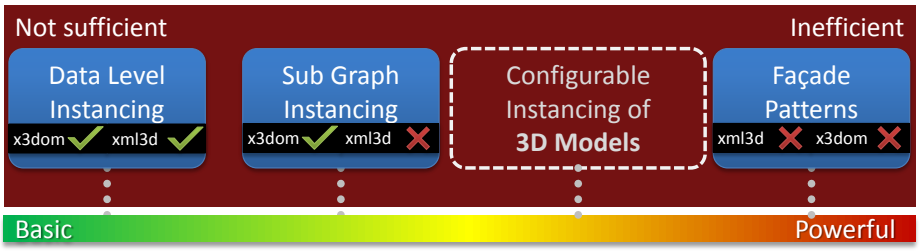


Figure 4.3: Different levels of subscene abstractions from simple data sharing to complex component models based on the facade patter. X3D lacks an instancing technique that allows reusing a subscene that is still configurable in a way that the coherence of instances is still exploitable (depicted here as "Configurable Instancing").

Performance X3D is declarative and has a high abstraction level. Thus it offers implementations various options for performing optimizations in the background. Although originally based on Open Inventor, the evaluation of an X3D scene graph is not order dependent. With the exception of transformations, which are affected by the graph hierarchy, all other states are defined locally to the affected *Shape* node. Hence, the graph can be easily flattened to a list of shapes with a list of evaluated (global) transformations that can then be ordered by states. A common implementation strategy is to expose X3D as application layer and to map parts of the functionality to a scene graph library. For instance, the X3D browser *Instant Reality* uses OpenSG for rendering [BDR04].

On the downside, X3D has data structures that require intensive pre-processing before they can be sent to the GPU.[24] For instance, most X3D geometry nodes provide multiple indices, i.e. for separately indexing positions, colors, normals, and texture coordinates. This allows for a more compact representation of the geometry. However, since recent graphics APIs provide only a single shared index for vertices, the vertices need to be "split" if they differ in any attribute. This requires pre-processing of all vertices in case multiple indices are present.

In addition, nodes with primitives of varying size (IndexedFaceSet, IndexedLineSet, IndexedFaceSet, etc.) contain occurrences of "-1" in index fields in order to mark the restarts of primitives. This representation is not suitable for stream-processing: To be compatible with recent graphics APIs (e.g. OpenGL's *glMultiDraw** calls) the information needs to be split into two separate buffers containing the indices and the sizes of each primitive.

[24]This is very similar to COLLADA and the motivation for the development of glTF (see Section 4.1).

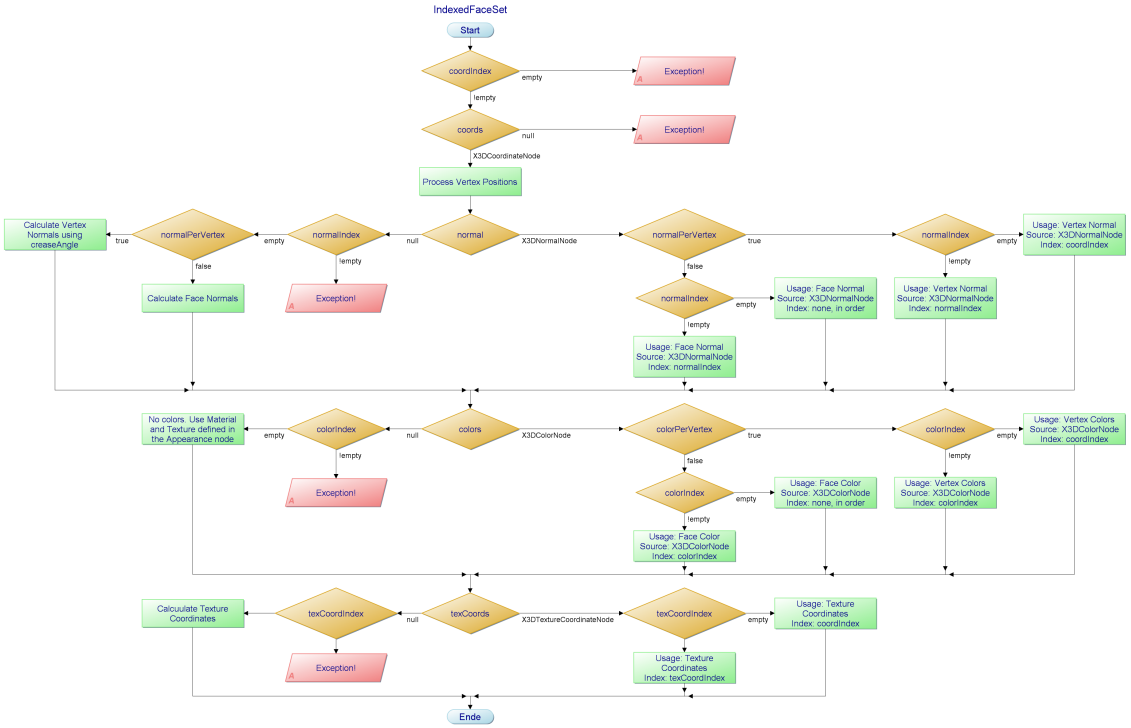


Figure 4.4: Flowchart for the evaluation of X3D's prevalent geometry node <IndexFaceSet>. The existence and values of the parameters determines the processing path (yellow diamonds). The red parallelograms represent invalid states. Note that 108 unique paths exist for the evaluation.

The ambivalent <IndexedFaceSet> node is striking example for the complex processing required to convert X3D data structures into a GPU-friendly buffer representation. It may define faces with arbitrary number of vertices which need to be triangulated, the index buffer needs to be filtered, and the multiple indices (position, normal, color, texture coordinates) require splitting the vertices. Colors and normals can be defined per face, which requires additional splitting of vertices. Additionally, the browser needs to generate normals and texture coordinates automatically if not defined. Figure 4.4 shows the variations in the processing logic that is necessary to transform an IndexedFaceSet node into a homogeneous data structure as understood by today's GPUs. The figure does not even include the splitting operations required for conversion into a GPU-friendly data structure.

Even worse, the browser needs to keep the original representation in memory (doubling the memory requirements) and the processing needs to be re-triggered as soon as the application changes any value in this representation. As a result, the IndexedFaceSet node is not suitable for instance for dynamic meshes.

As we will show in Section 5.4, we designed XML3D to avoid any *implicit* processing of geometry nodes. As a result, vertex attributes and element indices can be stored in GPU buffers *as is*. Nevertheless, it is still possible to have multi-indexed data that gets pre-processed *explicitly* using our declarative data flow approach.

Dynamics Another concept that X3D derived from Open Inventor is its event propagation model: *Routes*. To modify the fields of an object, X3D allows declaring directed

connections between fields (routes). The Routes concept is an event propagation model that connects output fields with input fields of the same type, e.g. single colors (SFColor), but also nodes (SFNode) and node collections (MFNode). The input field of a Route automatically derives the value from the connected output field if its value changes and the receiving node can process a response to that change, for instance by changing another field or changing the structure of the scene graph.

X3D provides a list of *sensor* nodes that trigger on system, network, or user events (<TimeSensor>, <TouchSensor>, <LoadSensor>, etc.). Additionally, it provides a set of nodes useful for typically use cases, e.g. interpolator nodes that allow defining key frame animations. For cases not covered by these nodes, X3D provides a <Script> node that allows defining arbitrary input and output fields that can be connected to the routing event model.

The route-based event mechanism of X3D (and Open Inventor) is similar to an Observer pattern [GHJV95], introducing functional dependencies between nodes. This "dependency graph exists independently of the structural graph of the scene, and represents a form of dataflow" [KS06]. X3D's Routes are a powerful concept to describe dataflow dependencies, however, the concept has also disadvantages:

1. Routes are X3D's only event model. This means, scene authors need to declare a dataflow graph even for very simple operations, for instance for modifying a single field. X3D provides a number of nodes that "gives authors the capability to gate, convert, or sequence numerous event-types for to convert and filter events for common interactive applications without the use of a Script node" [Web04a]. However, the number of available nodes is not sufficient even for small use cases and renders necessary to leave the declarative part of the scene and use scripting instead. For instance, changing an object's color in response to when the mouse pointer hovers above its position on the screen requires declaring a sensor node, a script node that holds the two colors for both hover states, and three ROUTE statements in the scene description. For comparison, the same functionality can be achieved in HTML with two lines of JavaScript triggered from *mouseover* and *mouseout* events.

2. Routes connect fields of node instances, i.e. data dependencies are defined between concrete occurrences of objects. As a result, Routes cannot be used to define general data dependencies in the sense of a template. The only way to reuse Routing is declaring them in Prototypes, whose drawbacks have already been discussed.

3. With Routes, processing is limited to single fields or to a collection of (structured) nodes. Hence, it is – similar to 3D toolkits based on a dataflow approach – hard if not impossible to exploit data parallelism beyond the scope of a single node in the dataflow graph. To our knowledge, there is no X3D browser capable of mapping route-based dataflows to the GPU. As a result, there is no hardware acceleration for animations and such.

In summary, X3D requires declaring complex and verbose route-based dataflows even for simple tasks. On the other hand, the Routes event mechanism is not ideal to describe general data processing and to detect operations that can be merged and mapped to the GPU. In Section 5.5, we will propose a novel declarative approach for describing

general data processing in a way operations can be easily merged and mapped to data parallel hardware. All other processing – in particular defining operations that modify the structure or attributes of the scene description – is done in the imperative part of the application, i.e. using the simple, well-known, and powerful combination of DOM events and JavaScript.

Functionality and Flexibility Like for many scene graph libraries, X3D's approach to provide new functionality is based on introducing new node types. This approach comes with two disadvantages: The semantic of newly introduced node types are often very specific for a certain domain and hence not of interest for many applications. Additionally, the "thinking in new nodes" often manifests in monolithic designs even in cases where it would have been avoidable. We will demonstrate these claims in the following based on some examples.

As already mentioned, X3D provides *components* to cluster sets of nodes with related functionality. X3D browsers may or may not implement such a component. In order to provide the functionality for specific use cases, the X3D specification includes very domain-specific components. For instance, the *Distributed Interactive Simulation* (DIS) component supports communication between an X3D world and distributed simulations using a very specific protocol mainly used in the military context. This DIS component is supported in a single X3D implementation. With nine active X3D implementations, 11 out of the 36 available components are implemented by three or less browsers.[25] Hence, the approach leads to an ever increasing number of components and nodes with lack of critical mass of implementations, authoring tools, and exporters providing the new functionality.

With increasing complexity of the required functionality, nodes often tend to provide multiple logically distinct aspects of the rendering process, i.e. they become monolithic. The Humanoid Animation (H-Anim) component [Web04a, Hum06] of X3D is a good example for a domain-specific component with a monolithic design: It interweaves multiple distinguishable aspects of presenting a 3D virtual character, including its hierarchical structure (skeleton), the interpolation and rendering of the attached mesh (skinning), and the animation of the skeleton (skeletal animation):

1. **Rendering:** Skinning requires the interpolation of vertex attributes based on the provided skeleton. H-Anim provides means to interpolate the position and normal attributes, but other vertex attributes are not supported. For instance, it is not possible to include tangents into the interpolation, although having tangents available is important for many applications in order to render detailed skin or more detailed characters using normal mapping techniques.

2. **Skeleton definition:** In H-Anim, each joint (<HAnimJoint>) defines a single transformation. This requires authors to transform the mesh into the coordinate system the animation data is defined in or, alternatively, transform the animation data to the space the mesh is modeled in. However, it is a common technique in character rigging to separate these coordinate systems by defining a reference coordinate system that defines the relationship between the animation system and the geometry,

[25] Source: `http://www.web3d.org/wiki/index.php/Player_support_for_X3D_components`, accessed 18 December 2015.

a so called *bind pose* (cf. [MCC11]). In this case, an additional transformation is required that brings the mesh into the bind pose before the animation gets applied. Having a standard bind pose helps reusing animations. However, this separation is not possible in X3D.

3. **Instancing:** It is not possible to reuse a humanoid description (*HAnimHumanoid*) with individual animations. The shapes and materials can be reused using the USE/DEF mechanism. However, it is not possible to instance a human just reusing the animations. Instead the structure of each humanoid needs to be replicated including all routes.

4. **Animations:** In order animate an H-Anim model, the author needs to declare the data-dependencies for each joint using routes. Again, using routes to setup the dataflow graph is very verbose. For instance, assuming a key-frame animation of a modest H-Anim model with 80 joints requires to define 160 interpolation nodes (one for the rotation and one for the translation of the joint) and 320 <ROUTE> statements to connect the interpolation nodes with the joints and a triggering node (e.g. a <TimeSensor>). Since route-based dataflows are not reusable, this declaration needs to be repeated for each H-Anim humanoid. Not even the animation data can be shared.

   Working with multiple animations requires rewiring the routes with another set of interpolators to exchange animations. Jung et al. state, that "for simple scenarios, like a single animation to be played, H-Anim/X3D works well, but it is hard to use in cases where multiple animation sets are available, which are combined and concatenated dynamically during runtime. The overall structure of such an application gets unmanageable and confusing because of the vast amount of nodes, routes and missing information about membership to specific information" [JB08].

Additionally, the fixed functionality of the H-Anim humanoid definition requires each application to transform their internal representation to the H-Anim data structures and hence contradicts our requirements on flexibility of data structures. We discussed this example here in more detail to be able to compare it with our approach to compose data processing functionality from basic building blocks. Despite the higher flexibility achievable with our approach, it is more concise, allows reuse of data, and can be mapped to GPU.

Visual Quality In terms of visual quality, X3D lacks behind modern scene graph libraries such as OpenSG and Ogre. This is mainly because X3D offers a single lighting model based on Blinn-Phong shading [Bli77], which was the default shading model of the OpenGL/DirectX fixed-function pipeline in the 1980/90ies. The lighting can be configured with a single material model and simple uniform parameters (<Material> node). Only the coefficient for the diffuse shading can be defined per-vertex or can be fetched from a single texture.

Schwenk et al. [SJBF10] propose an additional material model that offers some more control over shading including using textures maps for the specular and shininess parts and normal mapping. However, although the proposal is a great improvement to the

outdated standard material model, it still lacks behind the flexibility and visual quality achievable with programmable shaders. This includes important features such as arbitrary procedural textures and multi-texturing of arbitrary shading aspects.

Alternatively, X3D offers using custom GPU shaders. However, this comes with all the disadvantages already discussed in Section 4.1 and, hence, is rarely used in practice. Similar to all file formats and 3D libraries that aim to be portable, X3D is caught in the material dilemma described above. For a modern declarative approach it appears imperative to overcome this dilemma in order to be able to compete with non-portable solutions in terms of visual quality and flexibility. In Chapter 7.5 we suggest a way out of the dilemma introducing a novel domain-specific language to describe portable materials (shade.js).

4.4 3D in the Browser

With the advent of WebGL [Khr09], plugin-free hardware-accelerated 3D graphics has arrived in all major Web browsers. WebGL introduces 3D graphics to a powerful and mature software application development environment, which is probably – beside apps on mobile platforms – the most important platform for software today. The Web is cross-platform and can be accessed regardless of whether one is using a smart phone or a super computer. Consequently, WebGL settles upon OpenGL ES 2.0, the least common denominator in order to run on the majority of devices. WebGL also brought together 3D graphics and JavaScript. JavaScript is a language that previously did not play a major role in 3D graphics, but is – due to the relevance of the Web – one of the most important programming languages today.

Similar to desktop graphics applications, the direct use of WebGL is not required for most applications. Dozens of WebGL libraries emerged to simplify the development of 3D web applications. In this section we describe approaches of existing WebGL libraries and – in some more detail – the approach of X3DOM, which integrates X3D into HTML and is, alongside our approach, the only other approach that uses the DOM as scene API.

In general, WebGL libraries that aim for a higher abstraction level either define their own scene model or adopt an existing one. Additionally, a library can define its own specific API or it can use the existing generic DOM API. The design space spanned by these options is illustrated in Figure 4.5. Similar to the common C++ 3D toolkit libraries, most counterparts for WebGL define their own abstract model and API to describe and modify 3D scenes (e.g. three.js[26], SpiderGL [DPGS10]). In contrast, the API of OSG.JS[27] is kept as similar as possible to the API of OpenSceneGraph [BO04]. Similar to an external image, Cobweb[28] lets users embed X3D scenes into a web page using a single <X3D> element. The scene can be accessed using the X3D-specific scene access interface (SAI).

SceneJS[29] is specific in that it is based on a declarative scene description in JSON [Cro06]. However, since JavaScript offers no generic way to monitor changes in objects, the scene needs to be modified using a specific API.[30]

[26] `http://threejs.org/`, last accessed 01 July 2016.
[27] `http://osgjs.org/`, last accessed 01 July 2016.
[28] `http://titania.create3000.de/cobweb/`, last accessed 01 July 2016.
[29] `http://scenejs.org/`, last accessed 01 July 2016.
[30] The ECMAScript 6 proposal defines *Proxy* objects, that can be used to track changes in JavaScript objects.

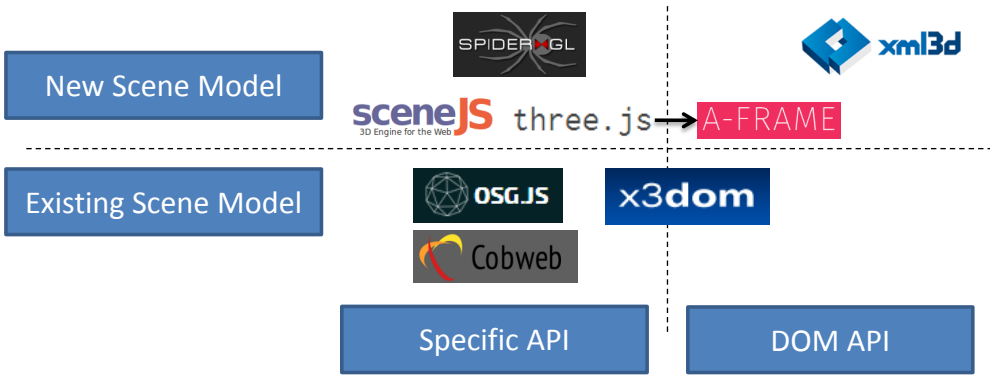


Figure 4.5: Different approaches to integrate 3D graphics into the web: SceneJS, three.js and SpiderGL define their own APIs and concepts; OSG.JS tries to keep the model and the API as close as possible to OpenSceneGraph. Cobweb maintains an X3D scene description outside the DOM and provides the X3D-specific scene API (SAI) to access it. X3DOM (Section 4.4.1) adapts the X3D model and a subset of the SAI as well, but additionally embeds the scene into the DOM. A-Frame (4.4.2) provides a DOM API for the three.js scene model using Web Components. XML3D, on the other hand, is specifically designed as extension to HTML. Consequently, its (new) scene description is embedded in the DOM, it uses the DOM API for modifications, and provides new concepts only where necessary.

Despite being part of the *execution environment* of the web page and rendering the scene to an HTML <canvas> element, these libraries are not further integrated into the web technology stack. In particular, the 3D content is not part of the DOM, which is the central common data structure for all web applications. The majority of web frameworks modify the DOM in response to user actions and DOM events in general, or bind application data to DOM elements and attributes or CSS properties (jQuery, AngularJS, Backbone.js, D3.js, etc.). Due to the common data structure, these frameworks can interoperate which each other. This is a unique and important characteristic of the web and cannot be found in other application environments. If, in turn, a WebGL library defines its own data structure and event model, the interoperability with these frameworks is lost. Hence, these libraries are not deeply integrated into web technologies as specified in our design criteria.

Additionally, none of these libraries aim to be platform-independent but are closely tied to WebGL. This contradicts with our requirements on platform and renderer-independence. Similar, since each scene description has its own serialization and its own API that developers are required to learn, these libraries contradict with our requirements on learnability.

4.4.1 X3DOM

X3DOM [BEJZ09] is an approach to integrate the existing scene description X3D into the DOM. Therefor Behr et al. stripped down the functionality of X3D to a subset defined in a new *Web* profile and replace scripting and other aspects with functionality already available in the browser. X3DOM's streamlined abstract model omits some essential X3D concepts, for instance Prototypes.

The approach of X3DOM is similar to XML3D in that it uses the DOM as the core data

As soon as widely available, a JSON based scene description could gain more momentum.

structure for the scene description. In contrast, it tries to adapt an existing 3D graphics format to fit into the web, rather than extending the current web technology where necessary. X3DOM tackles the poor integration of X3D into the web technology stack and allows existing X3D content – as long as it does not exceed the proposed profile – to be rendered in the browser without a plug-in.

On the other hand, by using an abstract model that was developed largely independently from the web, it not only inherits all the limitations of X3D described in the previous section, but also runs into serious issues when X3D concepts are incompatible with web concepts and where similar concepts already exist in the web technology stack. In the following we describe a selection of issues that come with the X3DOM integration model:

1. **Incompatible concept: Internal DAG** The DOM is a *tree-structure* representation of the web page. Since the main serialization format of X3D is XML [Web04b], and any XML representation can be parsed to a DOM representation by definition, the DOM-integration of X3DOM seems to be straightforward.

   However, the runtime data model of X3D is a *directed acyclic graph* (DAG) with all edges between a node and its parents having the same rank. X3DOM exposes the DOM tree as a runtime data structure to the user, but needs to synchronize the DOM structure with its internal DAG structure in the back end.

   This synchronization creates a number of confusing and undefined behaviors. For instance, removing a sub-graph that is referenced from another sub-graph has a different semantic in both representations. Figure 4.6 shows a simple example: In the DAG, removing the node results in deletion of one of the multiple references to the Shape. In the DOM, the node named "Shape 1" is gone and Shape 2 holds an invalid reference. Hence, the DOM representation and the X3DOM DAG are out of sync.

   Two strategies for synchronization exist: In the first approach, the DAG serves as reference and the DOM tree needs to adapt in order to represent the serialization of the DAG's new state. This would replace Shape 2 with the subtree defined by Shape 1 and consequently one shape would still be rendered. Modifications of the DOM not initiated by user code but by the system are very unusual for the web. To our knowledge, no other web technology exists that would manipulate the DOM implicitly. Additionally, the semantic of deleting a defining node and of dangling references would be in stark contrast to SVG where nodes with dangling references are in an invalid state and hence ignored [W3C09d].

   The second approach takes the DOM tree as reference and the internal DAG needs to be transformed to reflect the new DOM state. With this approach, the operation above would result in no shape being rendered at all. This would comply to SVG's behavior. However, X3D's abstract model is not designed for this behavior since all X3D properties are defined locally to <Shape> nodes. As a result, it would require a user to analyze each sub-tree before removal since it could contain defining nodes used in other branches.

   The X3D community is not yet in agreement on how to deal with this issue.[31] The

[31] See discussion on the x3d-public mailing list: `http://web3d.org/pipermail/x3d-public_web3d.org/`

current implementation of X3DOM (1.6.2) produces an error if a defining node is deleted and stops rendering. This is a bug. However, regardless of the approach chosen, exposing the DAG based on the tree-based DOM interface comes with disadvantages for the user in either way.

2. **Duplicate concept: API** The DOM API provides generic access to manipulate the structure of the document and to manipulate element attributes based on a string representation of the attribute value. Additionally, each HTML element provides a specific interface for more convenient attribute manipulation, i.e. based on typed values. Instead of employing the same concept as HTML, X3DOM additionally provides a subset of the Scene Authoring Interface (SAI) [BJDA11], which is designed to work on X3D's DAG structure.

3. **Duplicate concept: Identifiers** In X3D the identifier of nodes is specified using the *DEF* attribute, whereas HTML uses the *id* attribute. X3DOM allows using both, *DEF* and *id* with a rule set that specifies what happens if both or only one of the attributes is specified.

4. **Incompatible concept: Fields and Routes** The DOM collects all child elements in a single *children* container accessible via the API (e.g. *appendChild* to append elements to this container). In contrast, X3D's data structure is a multi-container model, i.e. X3D nodes can have multiple fields that collect one or more children. For instance, a <Shape> node has three single node container fields: *appearance*, *geometry* and *metadata*.[32]

   However, the X3DOM integration model cannot discard the multi-container concept, because containers are node fields and therefor an essential part of X3D's Routing concept. Consequently, developers need to learn X3D's field concept – which is not visible in the DOM structure – in order to use the Routing concept.

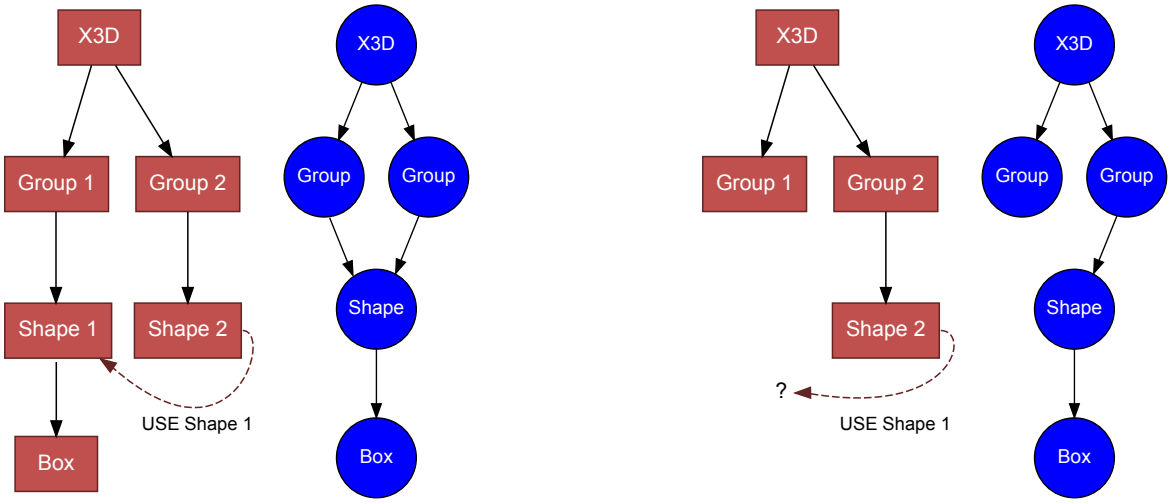
5. **Incompatible concept: CSS Property Inheritance** A central concept of CSS is the inheritance of values: If a value of an element cannot be computed, it uses the computed value of the parent element. Applying the same mechanism to a multi-parent data structure with equally ranked parent-child relationships creates ambiguities. Consequently, there is not inheritance of properties in X3D. Instead, all properties (<material>, <texture>, <lineProperties>, etc.) with the exception of transformations are defined local to the affected <Shape> node.

   Adding cascading states would be contradictory to X3D's concept to define all states locally. In addition, applying CSS to X3D's DAG would require additional rules for cases where the inheritance path is not unique and the resulting style has conflicting definitions. Consequently, X3DOM supports defining transformations using CSS Transforms, but no deeper integration with CSS is proposed.

6. **Duplicate concept: Event model** X3D and HTML both come with their own event model. Behr et al. [BEJZ09] state that "these two models differ dramatically". X3DOM

`2015-April/003310.html`, accessed 18 December 2015.

[32] In X3D's XML serialization, the parent's container a node belongs to is specified by its *containerField* attribute.



(a) (b)

Figure 4.6: Discrepancy between X3DOM's exposed DOM representation (red) and its internal DAG representation (blue). On the left side, both representations are synchronized. Shape 2 references Shape 1 in its USE attribute. In the X3D graph (blue), this reference is represented by a single node with multiple parents. The representations are out of sync, e.g. if the user deletes the defining node. The right side displays both representations after Shape 1 has been deleted from the DOM. In order to resynch the representations, either the DOM tree or the internal DAG need to be modified by the system.

supports both event models. Consequently, web developers need to additionally learn the X3D event model to be able to use the interactive elements of X3D.

7. **Duplicate concept: Metadata** In the context of 3D scene descriptions, metadata is information that is associated with a 3D object but is not a direct part of the scene representation. X3D provides a collection of nodes to annotate 3D objects with metadata. However, the web technology stack comes with technologies to annotate web documents, e.g. non-visible data using HTML5 custom data attributes [W3C14c], or embedding rich metadata on attribute-level via RDFa [W3C12a].

In addition to these duplicated and incompatible concepts, the main legacy adopted from X3D is the abstract model with its many outdated data structures. The X3DOM team has proposed and added (yet again more) new nodes to X3DOM, including nodes for the delivery of meshes [BJFS12, LJBA13, LTBF14] and for a more advanced material model [SJBF10]. While each of these new nodes by itself adds very useful functionality, the inadequacies of the core X3D model regarding integration into the DOM remain and there is still a large gap in terms of flexibility of X3DOM compared to imperative approaches.

When we initially analyzed the situation we did not see a way of consolidating the web technology stack with X3D. Despite many efforts, the above discussion shows that this indeed remains an unsolved problem. As an alternative, we approached the issue from the other side developing a fully web integrated approach picking the best possible ideas, without being limited by backwards compatibility.

4.4.2 A-Frame

A-Frame[33] is a recent declarative entity-component-system (ECS) [Sto06] framework on top of the three.js scene model. It uses Web Components to define *entities* in the DOM (<a-entity>). A selection of predefined components (e.g. material, geometry, sound) can be attached to this entities using DOM attributes. On top, it offers a selection of *primitives*, which are pre-configured entities with a fixed semantic (e.g. <a-box>, <a-sky>):

```
<a-entity geometry="primitive: box; depth: 2"
          material="color: #6173F4; opacity: 0.8"></a-entity>

<!-- Same functionality as above using a primitive -->
<a-box color="#6173F4" opacity="0.8" depth="2"></a-box>
```

A-Frame has a strong focus on WebVR applications and therefore provides primitives for instance for 360° video (<a-videosphere>) and for curved planes as user interfaces (<a-curvedimage>).

A-Frame uses the DOM and DOM events, but has no integration with CSS. It has no advanced material concept, but adapts the default material from three.js or, alternatively, allows use of specific shaders within the material component. A-Frame allows loading external assets from COLLADA, but provides no means to control included animations or configure any other parts of these assets. Similar, it has no concept for general data processing. Instead, it provide a very simple description for animations restricted to transition of single vectors, colors, numbers, and booleans. Hence, A-Frame does not meet our requirements on a general but flexible scene description.

Our approach is modeled closer to HTML: We deliberately decided against using an ECS as basis for our scene description, because HTML – unlike ECS – uses an inheritance principle and provides the basic functionality as elements with a fixed semantic. We think it can be beneficial for authors to find elements such as lights and geometry in the DOM.

However, using Web Components, one could still implement our approach using an ECS and provide the XML3D elements – as well as other convenience elements – as pre-configured entities with a fixed semantic similar to primitives in A-Frame. Research in this direction has been initiated by Lemme et al. in [LSSS16].

4.5 Summary

This section presented an overview of the state-of-the art in abstract 3D scene descriptions and 3D in the browser. As one can see, many diverse approaches exist. Hence, we had to restrict ourselves to the most important ones and to those that were most influential to our approach.

It can be observed from the discussion that 3D graphics lacks

1. a renderer-independent interactive scene description that is expressive enough to fulfill our requirements in terms of functionality, flexibility and visual quality. In particular describing materials and dynamic effects in a portable way remains an unsolved issue.

[33] `https://aframe.io/`, last accessed 2 August 2016.

2. a format tightly integrated into the web technology stack that allows seamless integration into this powerful application development environment.

Note that an expressive, render-independent scene description would be beneficial for the 3D graphics community also beyond the web context. In turn, such a scene description is essential for integration into likewise portable HTML5. Chapter 5 introduces the XML3D architecture, our approach towards such a scene description.

5 The XML3D Architecture

In the previous chapter we have shown that the existing approaches do not meet the design criteria we developed for a web-integrated 3D scene description as formulated in Chapter 3. Consequently, we developed the XML3D architecture that comprises novel concepts and technologies necessary to meet these design goals.

The XML3D architecture consists of seven core concepts that are realized with four technologies:

1. XML3D: A declarative abstract scene description designed as an extension to HTML5
2. Xflow: A declarative dataflow graph concept and system integrated into XML3D
3. shade.js: A language to describe programmable materials
4. Blast: A container format for efficient delivery of structured and compressed binary data

Figure 5.1 depicts the relations between the core concepts and these technologies. In this chapter, we describe the concepts of the XML3D architecture and why it was necessary to introduce them: Some of the novel approaches were required to integrate 3D graphics seamlessly into the web architecture, others to meet the application requirements in terms of functionality, flexibility, visual experience, and performance (see design criteria in Chapter 3).

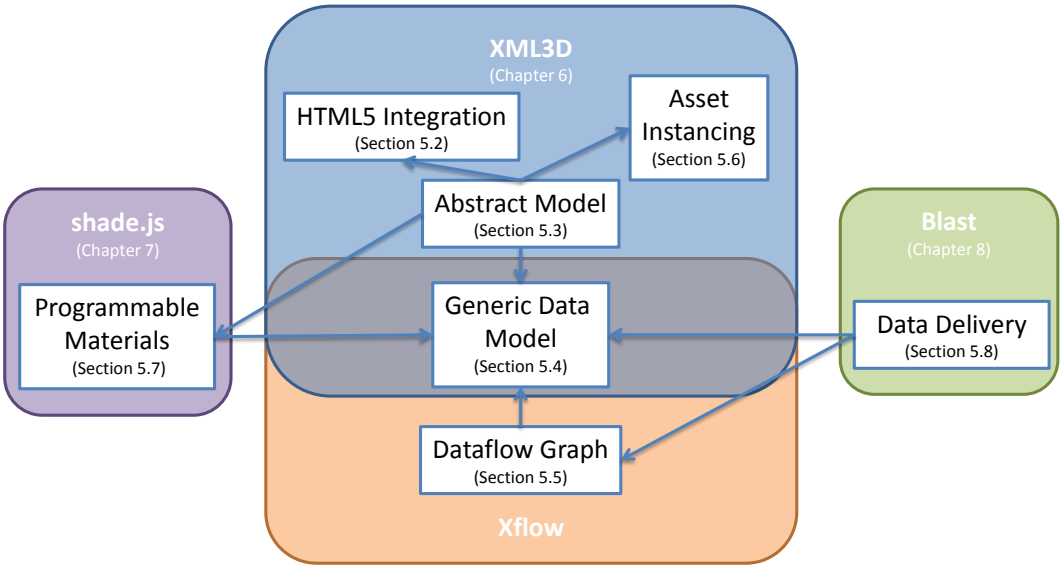


Figure 5.1: The core concepts and technologies of the XML3D architecture and their "integrates" relation.

In the following, we describe the current status of the XML3D architecture more as a snapshot rather than as a final version.[1] While it is designed to be extended and refined with the input from the larger community and new application requirements, we believe that the core concepts described in this chapter will remain valid.

This chapter is organized as follows: In the first section, we give a very brief overview of the concepts in order to provide context. In the remaining sections we discuss them in more detail.

5.1 Overview

Abstract Model and HTML Integration The core of the XML3D architecture is its abstract scene description XML3D. We decided to develop a new abstract model for XML3D for three reasons:

1. As shown in the previous chapter, the existing scene descriptions are either platform dependent or not flexible enough compared to non-declarative approaches, which is contrary to our design goals.

2. As we have shown for X3D/X3DOM, integrating an existing approach results in duplicated and conflicting concepts.

3. Developing a new scene description gives us the opportunity to start from HTML5 and design the abstract model in a way that it integrates as seamless as possible with existing web technologies.

As a result of these considerations, we started with HTML5 and designed XML3D as an extension to it. XML3D adds a lean set of new elements to HTML5, where – similar to scene graph libraries – elements represent abstract 3D objects such as geometry, lights, and materials but also generic data or whole subscenes (assets).

As a result of the integration with HTML, we can reuse existing HTML elements (e.g. <img> as source for textures), can attach properties to elements via CSS [W3C09a], and use DOM Events [W3C00] and DOM Scripting with JavaScript [ECM11] to enhance the scene with interaction.

Using the DOM API in combination with JavaScript plus the DOM event model allows for creating highly interactive web pages. Due to the DOM integration, XML3D events work exactly the same way as HTML events. For instance, adding an "onclick" listener works for visible XML3D elements and triggers a JavaScript callback whenever the user clicks on the object with the mouse pointer. In the callback, in turn, the user can modify the scene arbitrarily using the DOM API, e.g. change a transformation or the parameters of the camera.

Dataflow Graph Integration It is possible to arbitrarily modify the structure and data of an XML3D scene using the DOM API and JavaScript. However, for effects that require heavy computations (e.g. animation of meshes or manipulation of images) single-threaded JavaScript and the generic, string-based DOM API may be too slow. For these kinds of

[1] For the latest specification of XML3D refer to `http://www.xml3d.org/xml3d/specification/latest/`.

computations, we want to exploit the parallelism offered by the multi-core, many-core, and SIMD architectures of modern hardware. However, hardware architectures are very heterogeneous and not all hardware capabilities are exposed via APIs on all devices. To liberate web developers from taking care about finding the most suitable of the available APIs for anyone of the possible devices, we integrated *Xflow* – a declarative dataflow graph – into XML3D. Instead of describing *how* an effect is computed, the developer just describes *what* should be computed. The Xflow system is then capable of mapping subgraphs of the dataflow to available APIs.

However, the dataflow approach not only contributes to the performance criteria. It is also an important concept to increase the expressiveness of the scene description. Instead of providing monolithic elements that come with a single fixed functionality, the dataflow approach provides means to construct complex data processing functionality from building blocks. Additionally, Xflow allows adapting the scene description to suit the application's data structures. We discuss the latter in Section 10.2.

Programmable Materials The quality of rendering is an important design criterion for XML3D. Providing a suitable set of predefined material models is sufficient for many use cases. However, to be able to compete with platform-dependent rendering libraries, it is mandatory to provide scene authors the possibility to create customized materials.

However, existing shading languages are not suitable for describing custom materials because they *a*) lack platform and renderer independence – a key design criterion for XML3D, *b*) mix up concerns that are handled otherwise in XML3D (e.g. lighting is defined by light emitting objects in the scene and displacement and animation is handled via the dataflow approach), and *c*) describe materials not general enough to share them among different assets and platforms. Consequently, we propose *shade.js* – a novel material description language.

shade.js is based on JavaScript. It allows for programmable material descriptions that are able to adapt to their execution environment. Additionally, shade.js has an interface to the lighting system and thus does not interfere with other aspects of XML3D. A compiler uses just-in-time specialization of the material when it gets attached to a visible object in the scene. Then, the material is compiled to specialized shaders such as GLSL [KBR03] or OSL [GSKC10].

Generic Data Model We require a mechanism to define arbitrary named and typed parameters for the programmable material descriptions and for the Xflow dataflow graphs. Since DOM attribute and CSS properties cannot support this we had to develop a new data model for these parameters.

In XML3D, we provide a *single* uniform concept for the definition of data, which can be used to define the parameters of geometry objects, materials, lights, views, etc. It forms the basis for XML3D's abstract model, for the programmable material concept, and for the dataflow graph concept. Finally, it is also the foundation for the composition of data from various sources and for specialization of data sources and assets.

Instancing The DOM is a tree structure. Hence, it does not allow referencing, i.e. reusing data from other branches of the tree. Although it could be useful, the reuse of data is not

essential for 2D web applications. However, it is a very important concept in 3D graphics.

XML3D provides three ways of reusing data: Properties such as materials can be attached to multiple subtrees via CSS and therefore be used by multiple objects. *Data Instancing* allows reusing resources that provide (or evaluate to) *shallow* structured generic data such as parameters for the 3D scnene elements. On the other hand, *Asset Instancing* provides means to reuse *richly* structured data, i.e. whole scenes with meshes, materials, transformations, etc. Both instancing techniques provide means to override parameters from the referenced resource (specialization). The concepts for reusing or instancing objects in XML3D are carefully designed to overcome all the issues that come with integrating a general DAG data structure into the DOM (cf. X3DOM in Section 4.4.1).

Data Delivery We evaluated several existing approaches for the transmission of 3D resources to the client [DSR*13]. As a result, it became clear that none of the existing approaches was suitable for the heterogeneity of 3D resources, networks, and devices. Unlike for images and videos, which are homogeneous and have established compression schemes and container formats, nothing similar exists for highly structured and heterogeneous 3D resources.

Hence, we propose *Blast*, a container format for efficient delivery of large, structured, and heterogeneous binary data. As a container format, it is schemaless and compression-agnostic following a code-on-demand approach. The Blast encoder flattens a data structure into self-contained and encoded chunks. With all the encoding information available in the file format, the Blast decoder can reconstruct the decompressed data structure on the receiving side. Since all chunks are self-contained, they can be streamed and decoded in parallel to their transmission.

With this brief overview of the core concepts of the XML3D architecture and their relations in mind, we will discuss each concept in more detail in the following sections.

5.2 HTML Integration

In this section, we will discuss why we decided to design XML3D as an extension to HTML, review the main concepts of HTML and its related technologies, and discuss what we need to determine for XML3D's abstract model.

5.2.1 Design Rationale

The essential idea of our approach is integrating a modern abstract 3D scene description into the web technology stack. As shown in previous work (Chapter 4), there are two principle options to approach this idea: Either the base data structure of the scene description is a composition of JavaScript objects that live in the JavaScript execution environment of the web page. Alternatively, it is possible to make the scene description part of the web document that is represented in the DOM data structure.

Additionally, we had to decide to either use one of the existing scene descriptions as a basis and adapt it to make it fit into the web technology stack, or to design a new scene description from scratch. The resulting design space consists of four alternatives illustrated in Figure 4.5.

We decided to develop XML3D as an extension to HTML5 as a primary design choice. This decision implies we had to create a new abstract scene model that uses the DOM as base data structure and its API.

The choice for an HTML-based approach has been founded on the consideration that we require XML3D to be seamlessly integrated into other web technologies. This approach allows reusing many of the web technologies already available, including the DOM event model, CSS, and some existing HTML elements. In addition, XML3D is not only compatible with related W3C technologies such as RDFa, but also with all libraries based on DOM functionality (e.g. jQuery).

One of our design goals was to introduce new concepts only where necessary. Consequently, we decided against adapting an existing approach. Developing the abstract scene description from scratch, we are able to avoid the issues that result from integrating previous scene graph concepts into the DOM (cf. 4.4.1). Additionally, we can approach the issues of previous general scene descriptions, including the material dilemma discussed in Section 4.1 and the lack to describe dynamic effects more generally.

5.2.2 HTML Concepts

We gave an overview on HTML and its related technologies in Chapter 2. In this section, we will discuss the main HTML concepts in more detail as a basis for the subsequent discussion on how to adopt and where to extend these concepts.

Elements HTML's abstract model consists of elements organized in the DOM. "Elements in the DOM represent things; that is, they have intrinsic meaning, also known as *semantics*" [W3C14c].

```
<input>
```

In this example, the <input> tag is an HTML element that "represents a typed data field, usually with a form control to allow the user to edit the data" [W3C14c]. For each HTML element, the specification includes *a*) a description of what the element represents, *b*) the attributes that may be specified on the element, *c*) description of what content must be included as children and descendants of the element, and *d*) the DOM interface the element is required to implement.[2]

Attributes DOM attributes can be used to parameterize DOM elements. Each HTML element has a predefined set of attributes. Some global attributes are common to all HTML elements, e.g. the *id* attribute to identify the element uniquely and the *class* attribute, which contains a set of space-separated tokes that represents the various classes that the element belongs to.

In general, DOM attribute values are string representations of simple data types (enumerations, numbers, etc.):

```
<input class="my-form" type="number" name="quantity" min="1" max="5">
```

[2]This list is not comprehensive. For a complete list refer to the HTML specification [W3C14c].

In the example above, the *type* attribute of the <input> element not only determines the kind of the data field that must be displayed, but it also defines which attributes are valid for the element. For instance, the attribute *min* is valid for <input> elements of type *number*, *range*, and *date*, but not for the other 16 predefined input types.

CSS CSS properties are a second orthogonal concept to parametrize HTML elements. Each browser supports a fixed set of CSS properties and each of these properties has its predefined set of allowed values specified as string. The property *border-top-color*, for instance, specifies the top color of the box around an element and can hold either a color value, the value "transparent", or the value "inherit".

By its very nature, CSS is used to layout and style elements on the page, but also to define animations for these properties. Using CSS selectors, CSS style rules can be applied to elements based on characteristics such as element names, class names, order in structure, occurring attributes, and many others. As a result, it is possible to style a document without changing the structure or content of the DOM (and vice versa).

Content Each HTML element has a content model, i.e. a description of the element's expected contents. This may include a list of other elements and text (represented as *Text* nodes in the DOM).

Most HTML elements are self-contained, i.e. they can also be valid with a different parent in the DOM. For instance, a paragraph defined in a element could become a child of a <section> element in an other branch of the DOM tree. However, there are some HTML elements that may only occur as children of specific elements. The <option> element, for instance, "represents an option in a select element or as part of a list of suggestions in a datalist element" [W3C14c] and thus can only occur as child of such elements. Here, a child element is not self-contained but describes a parameter of the parent element as an alternative to a DOM attribute or CSS property. In general, this mechanism is used to describe more complex parameters.

Embedded Content Some elements such as <img> and <video> allow embedding content from another resource in the HTML document. These elements typically reference the binary and large content of an image or a video using a URL:

```
<img src="images/search.png">
```

As a result, the HTML document remains manageable and the external resource can be loaded asynchronously by the browser. Thus, the browser is able to layout and present a provisional version of the web page to the user before the external resource is finally loaded.

Layout The process of positioning objects in space is called *layout*. The *CSS Box Model* [W3C09a] requires rectangular boxes to be generated for all elements in the DOM. CSS provides various techniques to layout a web page based on these boxes.

The layout techniques can be divided into two categories: Using *implicit* layout techniques, the browser manages the layout for the user. The most commonly used implicit

layout technique is the *flow layout*. Here, the browser's layout engine positions the elements in order of appearance from the top of the window to the bottom avoiding vertical scroll bars. However, the layouting is highly configurable using CSS properties.

Implicit layout techniques ensure flexibility because the browser will try to follow the user's intentions independent of the size, shape and resolution of the browser window. If one of these parameters change, the layout engine will adapt the layout to fit to the new setting.

Using *explicit* layouting techniques, the user can bypass the browser's layout strategy and position elements freely; either relative to its parent element or to the overall page. Again, the HTML's positioning is based on the bounding boxes derived for all involved elements. For instance, one can specify element A to be positioned on the right side of element B but translated 13px to the left:

```
div.right { float: right; position: relative; left: -13px; }
```

Most web pages switch between implicit and explicit layouting within the page, depending on if the spatial relation of elements is of importance, or if it is more important to ensure that the elements are well positioned in the visible part of the web page (viewport).

DOM and DOM Interfaces The DOM is "an API for accessing and manipulating documents (in particular, HTML and XML documents)" [W3C09c]. It represents the elements of an HTML document in a tree of *Element* objects, the base class of all HTML elements. Each Element object provides generic functionality including string-based modifications of attributes, modification and traversal of child and sibling elements, and access to its parent element and owning document. The interface of the owning document supports programmatic creation of new elements.

Additionally, each HTML element provides an individual interface in order to provide element-specific functionality including typed access to its attributes. The following example illustrates generic as well as specific functionality provided by the DOM API:

```
// Create an element programmatically
var c = document.createElement("canvas");

// Modify the current document
document.head.appendChild(canvas);

// Modify an attribute using the <canvas>-specifc API
c.width = 1024;
typeof c.width; // "number"

// Access the same attribute using the generic DOM API
c.getAttribute("width"); // "1024"
typeof c.getAttribute("width"); // "string"
```

DOM Events HTML uses the well-known DOM event model [W3C00], which allows users to register event listeners for specific event types on DOM elements:

```
// Get the first <div> tag in the documents
var c = document.querySelector("div");

// Add a listener that triggers when a user clicks into the div
c.addEventListener("click", function(evt) {
    // The first parameter holds information on the event
    console.log("Element clicked:", evt.target);
});
```

The DOM event model has an event flow mechanism that propagates events through the DOM tree. As a result, events can not only be handled locally at the DOM element that causes the event, but also "centrally from an EventTarget higher in the document tree" [W3C00]. In the examples above, the event listener not only triggers for clicks on the element, but also when any other descendant element has been clicked.

In addition to attaching event listener programmatically, HTML defines more than 60 global event types that are also available as event attributes [W3C14c]. In general, these attributes correspond to the name of the event type with the prefix "on":

```
<div onclick="alert(this);">
  ...
</div>
```

5.2.3 Discussion

We decided to design XML3D as an extension to HTML. However, this design choice still leaves open how to utilize and transfer the HTML concepts to 3D content. In particular, we need to decide:

- What are the "things" in a 3D scene we want to represent as DOM elements?
- How do we parametrize these elements, i.e. which scene parameters are defined using either CSS properties, DOM attributes, or child elements?
- An essential functionality in 3D scene description is sharing and instancing of data but also of whole subscenes. A common approach to support this is organizing the scene in a DAG. How can we provide this functionality when we employ the DOM tree as base data structure?
- What parts of the 3D content should be defined in external resources, similarly to embedded content in HTML?
- Can we employ the CSS layout strategies also for 3D content?
- How can we reuse existing HTML elements, CSS properties, and DOM events for the 3D scene description and which additional do we need?
- What are the DOM interfaces of the newly introduced elements?

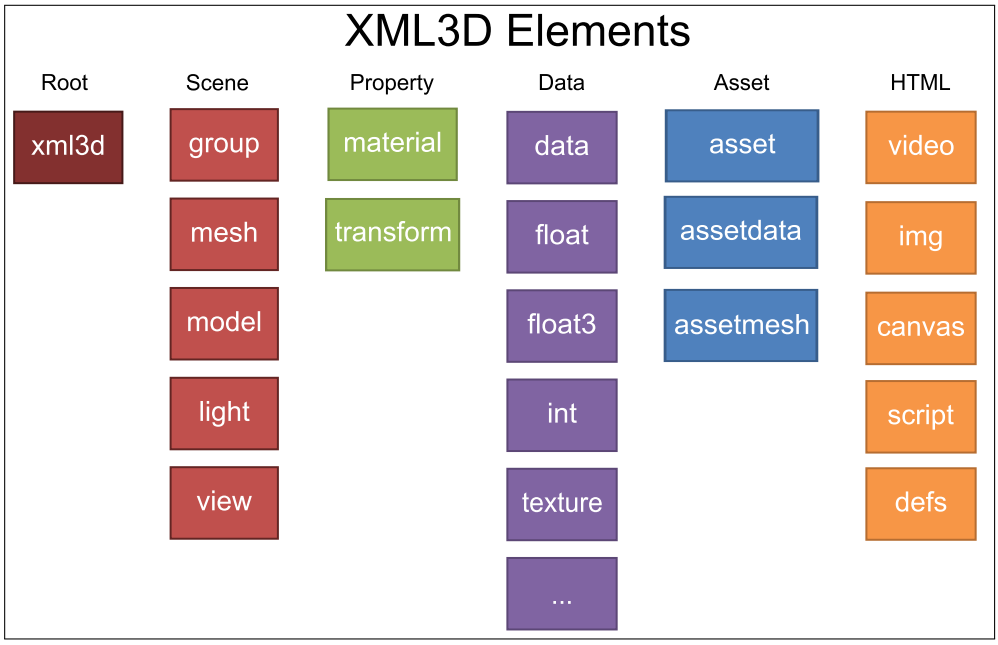


Figure 5.2: The elements of XML3D's abstract model.

Beside these questions, i.e. how to apply and reuse existing concepts, we will also need to introduce new concepts if (and only if) the existing concepts are not sufficient for a flexible and competitive scene description.

In the following sections we will answer these questions and present our concepts for XML3D, the first 3D scene description that tightly integrates with HTML. We will discuss the novel concepts of our architecture, but we will not describe each and every element, attribute, and interface in its very detail. Refer to the XML3D specification [SSK16] for a detailed formal description.

5.3 The Abstract Model of XML3D

In a first step, we need to identify the real life "things" we want to represent as elements in the 3D scene description. Existing 3D formats and scene graph libraries give a good indication on what is needed to describe a 3D scene. However, in contrast to previous 3D approaches (and similarly to HTML), we want to identify the essential basic functionality needed to describe the addressed domain. In particular, we abstain from introducing elements (or attributes) that:

- provide convenience functionality that can be equally implemented using JavaScript. For instance, the functionality of X3D's <Switch> node [Web04a] can easily be implemented setting the *visible* properties of the children of a grouping element accordingly.
- describe layout or style if it can be better described using CSS. Since the introduction of CSS, all HTML elements that are meant to describe layout or style have been deprecated (e.g.). Similar, we provided a <transform> element in XML3D

to describe 3D transformations. Transforming a 3D element can be considered to be part of the layout of the scene. The recent CSS 3D Transforms [W3C09b] standard provides means for describing 3D transformations as part for the CSS *transform* property making the <transform> element obsolete.

- describe (internal) data processing that can be equally described using our declarative dataflow processing approach (see Section 5.5). For instance, X3D's <ElevationGrid> represents "a uniform rectangular grid of varying height in the Y=0 plane of the local coordinate system" [Web04a]. The functionality of this node can be equally implemented with two operators: One that emits a grid with the specified dimensions, and one that varies the vertices of the grid based on the other parameters of the node.

In X3D, every specialization has its dedicated node. For instance, X3D's <PointLight> node represents a light in general and, more specifically, a light source model that emits light equally in all directions. Similar to the HTML's <input> element, we decided to split these two levels: The element represents a general concept and attributes specify a specific characteristic, also resulting in a varying parameter set.

As a result, XML3D consists of only five scene elements (see *Scene* column in Figure 5.2):

<mesh> is an element that describes a set of geometric primitives. The kind of primitive is specified by its *type* attribute. Currently, XML3D supports all primitives available in WebGL (triangles, lines, points, etc.). Depending on the active primitive type, the set of valid parameters may change. We will discuss the evaluation of the <mesh> element in detail in Section 6.1.1.

<light> is an element that describes a set of light sources. The used light model is specified via its *model* attribute. Currently, XML3D supplies a set of predefined common light models (point, directional, etc.) identified via URN (e.g. urn:xml3d:light:point). Depending on the active light model, the set of valid parameters may change. We will discuss the evaluation of lights in detail in Section 6.1.2.

<view> is an element that defines a view into the scene, i.e. a projection of the 3D scene to 2D. The used projection model is defined by the *model* attribute. XML3D supports a set of predefined common view models (perspective, projective, etc.) identified via URN (e.g. urn:xml3d:view:perspective). Depending on the active view model, the set of valid parameters may change.

<group> is an element that collects scene elements in the DOM tree in order to describe hierarchies (parent-child relationships). Otherwise it has no special meaning.

<model> is an element that references a subscene (asset), i.e. a collection of all the elements above. Additionally, elements of the referenced asset may be configured using parameters of the <model> element. We will discuss the configuration of assets in more detail in Section 5.6.

We call these five elements *SceneElements*. In the remainder of this section, we will discuss how we embed the SceneElements into HTML and how existing HTML concepts apply to our newly introduced elements.

In the subsequent sections, we will discuss the new concepts we developed since available alternatives did not meet our requirements. This will include some important topics we leave open for now, for instance how to *a*) parameterize the Scene Elements (Generic Data Model in Section 5.4), *b*) animate these parameters (Dataflow Graph in Section 5.5), *c*) customize the shading of surfaces (Programmable Materials in Section 5.7), and *d*) deliver possibly large 3D content to the client (Data Delivery in Section 5.8).

5.3.1 3D Layout

The SceneElements listed above need to be positioned in 3D space. In this section we will discuss if the existing HTML layout concepts can be applied to 3D content.

Implicit layout techniques play an important role in HTML. Here, the browser actively manages the positioning of elements. The used layout strategy can be configured via CSS. In our view, there is little reason for implicit layouting of 3D scenes, because the position of the 3D objects is an inherent part of the 3D content. If the browser would change these positions, e.g. rearrange the 3D objects of a factory scene in order to make it better fitting into the viewport, it would break the meaning of the scene. As a result, we allow explicit positioning of SceneElements only.

Another important concept in HTML layouting is the relative positioning based on the spatial extend of the element (2D axis-aligned bounding box). This concept could be easily transferred to 3D content, using the 3D axis-aligned box as basis for positioning. Therefor the set of CSS properties would have to be extended by *front* and *back* properties to enable authors configuring the additional dimensions. For instance, in order to position rows and seats within a row in a virtual theater, seats and rows could be positioned based on their spatial extends:

```
/* Position seats to the right to their predecessor **/
mesh.seat { position: relative; float: right; left: -13; }
/* Position rows to the back to their predecessor **/
group.row { position: relative; float: back;  front: 210; }
```

In the example above, we can compensate the too conservative bounding-box approximation of the seat using a negative offset. Likewise, we can provide space for aisles between the rows providing a padding to the front. Replacing a seat with a larger partner seat within a row for instance would not require any changes in the transformations of the other elements representing seats.

However, due to the fact that this kind of layouting is neither established in 3D scene descriptions nor available in DCC tools, we decided to not support positioning based on spatial extend (for now) and leave this idea for future work.

Instead, XML3D provides a classical positioning of 3D elements based on coordinate systems: Each SceneElement defines a coordinate system, which is relative to its parent coordinate system. Since CSS is responsible for layouting, we should use CSS properties to describe these coordinate systems. Providentially, we can use the already existing *transform* property to define 3D transformations [W3C09b]:

```
<style>
  .mat {
    transform: matrix3d(1.884100, -0.280036, 0, 0, 0.549607,...);
  }
</style>
...
<!-- Transformation property assigned via class attribute -->
<group class="mat">
  <!-- Transformation property assigned in local style -->
  <mesh style="transform: scale3d(2.5, 1.2,0.8) rotateY(10deg);" src="teapot.json">
  </mesh>
</group>
```

Transforming nested <group> elements with the *transform* property can be used to build transformation hierarchies as commonly found in scene graph libraries.

The *transform* property can also reference a <transform> element. However, this element is deprecated. Instead, we allow referencing dataflow graphs which additionally supports dynamic transformations. We will discuss this feature in Section 5.5.

5.3.2 Embedding 3D Content

We need to specify the content model for XML3D SceneElements, i.e. in which contexts these elements may appear. It would have been possible to categorize SceneElements as *Flow Content* [W3C14c] and thus allow them whenever flow content is expected, e.g. as children of or <p> elements. To allow a free mix of HTML and XML3D SceneElements, it requires to define how the SceneElements influence the 2D layout of the document, i.e. we need to define a 2D rectangular box for each SceneElement that can be used for the CSS Box Model. We can derive such a box from a 3D object by projecting its 3D bounding box to the screen and use the extends of this projection to generate the rectangular box.[3]

Changing the projection of the 3D bounding box, which happens when moving the camera for instance, results in a modification of the derived rectangular box of the SceneElement and the browser needs to (re-)layout the page based on the changed extend of the SceneElement. Since this may occur every frame, navigation would lead to flickering of the web page.

Anyway, most use cases require a fixed-size "viewport" that provides a view into the scene specified by the field of view of the virtual camera. All content within this field of view is visible, content overflowing this region is clipped.

As a result, we decided to introduce the <xml3d> element with SceneElements only appearing as descendants of this element. The <xml3d> element defines a drawable region within the HTML document that has an explicit size which may be controlled using CSS for instance. All CSS properties that influence the style and behavior of the HTML layout also apply to the <xml3d> element. Additionally, it defines the viewport of the projection defined by the currently active <view> element, which is referenced from the <xml3d>'s *view* attribute.

[3] SceneElements such as <light> and <view> do not have a spatial extend. Nevertheless, their 3D position can be projected to 2D and a zero sized box can represent their 2D position.

```
<div>
  <xml3d class="right" view="#defaultView" >
    <view id="defaultView" model="urn:xml3d:view:perspective">...</view>
    ...
  </xml3d>
</div>
```

In the example above, all SceneElements are clipped against the frustum defined by the referenced perspective view model.

The <xml3d> element also provides the root coordinate system for the scene, a right-handed 3D coordinate system in the (0,0,0) origin, with the +x-axis pointing to the right, +y-axis upward, and +z pointing towards the viewport (the user). Applying the CSS transform property to the <xml3d> element will affect (e.g. skew) the rendered 2D representation of the viewport, but will not change the root coordinate system.

Vice versa, HTML elements may appear as children of SceneElements. Since we can derive the rectangular box for each SceneElement, we can position these elements relatively to the SceneElement using standard CSS properties:

```
<div>
  <xml3d>
    <group>
        <div style="top: -5px;">Label</p>
        <mesh src="..."></mesh>
    </group>
    ...
  </xml3d>
</div>
```

In the example above, the element's content is placed at the top-left corner of the projected bounding box with an offset of 5 pixels to the left.

5.3.3 Style

Existing CSS Properties XML3D reuses existing CSS properties where suitable. We have already discussed how we leverage the *transform* property to describe the local coordinate system of SceneElements. We identified a set of CSS properties that can be used to configure the rendering of SceneElements. For instance, we can use the *display* property to configure the visibility of objects including their children:

```
<mesh style="display: none;" src="teapot.json"></mesh>
```

HTML supplies the *z-index* property to configure the order of rendering for stacked elements. We reuse this property in XML3D to support stacked (or layered) rendering: All SceneElements are rendered in order of their evaluated z-index. Elements with identical z-index are composed using z-buffering. Layered rendering can be useful for instance for 3D widgets that should never be occluded by other SceneElements.

CSS Transitions allow "property changes in CSS values to occur smoothly over a specified duration" [W3C13a]. This technique can also be used on 3D transformations, allowing smooth transitions between 3D objects' poses.

Since we have a 2D bounding box representation for all SceneElements, we could also use all CSS properties that allow styling this representation (border, background, etc.). However, due to its limited use cases we leave this for future work. For a list of all currently supported CSS properties refer to [SSK16].

Materials In addition, CSS provides a set of properties to configure the predefined shading of the actual content. This set includes simple parameters such as *color*, and *opacity*. The SVG shading model supports the *stroke* and *fill* properties which may define, besides colors, also gradients. However, the existing CSS parameters are not sufficient to style 3D content with its variety of materials and their interaction with light.

As a result, we introduce <material> as a new element which is used to define a material model and its parameters. To attach such a material to a SceneElement, we introduce the new CSS property *material*:

```
<material id="phong" model="urn:xml3d:material:phong">
  ...
</material>
...
<!-- Apply the material to all child nodes if
  no other CSS rules apply -->
<group style="material: url(#phong);">
   <mesh>...</mesh>
   ...
</group>
```

The cascading nature of CSS also applies to XML3D materials: if more than one stylesheet rule applies, the implementation has to determinate which specific stylesheet rule has highest priority for an XML3D element. Similar to other CSS properties, the material property is derived from an element's parent if not otherwise specified. The document has a default material assigned. With these rules, we can determine exactly one assigned material for all <mesh> and <model> elements.[4]

Similar to <light> and <view>, the *model* attribute specifies the material model that should be applied. XML3D provides a set of predefined material models identified by a URN (e.g. *urn:xml3d:material:phong*). Alternatively, it can point to a programmable material model. We will discuss the concept of XML3D materials in detail in Section 5.7.

5.3.4 DOM Events

For XML3D we adopt the DOM event model. Consequently, web authors can create interactive 3D elements in exactly the same way as for HTML elements. For instance, it is possible to attach event listeners to XML3D elements using event attributes:

```
<group onmouseover="highlightMe()">
  <mesh src="teapot.json" onclick="alert('onclick triggered.')"></mesh>
</group>
```

Alternatively, event listeners can be attached to XML3D elements using the *addEventListener(type, listener)* method:

[4] Like for other CSS properties, we can evaluate the material property not only for these but for all elements in the document. However, for other elements the result will simply be ignored.

```
// Select a mesh from the document
var mesh = document.querySelector("mesh");
// Attach a listener that triggers when the mesh is clicked
mesh.addEventListener("click", function(evt) {
    console.log("Handler for 'click' triggered");
    // Hide the object below the mouse pointer
    evt.target.style.display = "none";
});
```

XML3D's <mesh> and <model> elements generate mouse pointer events [W3C00], touch events [W3C13c], and others.[5] Using these events, we can identify objects, e.g. below the mouse pointer, a common functionality for interactive 3D systems that we also required for XML3D (see Section 4.2). In contrast to common scene graph approaches, we do not rely on paths to identify the picked object: Since all SceneElements are organized in a tree, a simple reference (`evt.target` in the example above) is sufficient for distinct object identification. Additionally, the DOM event flow allows listening for these events on <group>, <xml3d>, or any other parent level.

Many web developers do not use the DOM API directly, but use libraries such as jQuery that make DOM manipulation and DOM events even more convenient to use. jQuery for instance provides wrappers that use the DOM API internally. Since XML3D is based on DOM and DOM events, jQuery can be used with XML3D out of the box:

```
// Selects *all* meshes in the document and attaches click listeners
$("mesh").click(function(evt) {
  console.log("Vanishing mesh!");
  // jQuery sets the display property internally
  $(evt.target).hide();
});
```

In comparison to X3D, where adding simple interaction based on picking requires sensor nodes and setting up chains of Routes, DOM events are easier to set up and more intuitive in many cases. On the other hand, DOM events cannot be configured. In particular configuring filters that allow dispatching only a subset of occurring events is useful for event types that possibly generate many events. This is one of the reasons why DOM MutationEvents have been made obsolete to be replaced by the MutationObserver API, which allows for fine-granular configuration of mutation events (e.g. only listening to specific attribute changes).

However, from the experiences we gathered, even large-sized scenes (see Chapter 9) did not suffer from emitting large amounts of events. Hence we sticked to the well-know and simpler DOM events.

[5] For the list of all supported events refer to the XML3D specification [SSK16].

5.3.5 DOM API

Since XML3D is an extension to HTML, all SceneElements derive the functionality of the DOM *Element* and the *HTMLElement* interface. Additionally, we define element specific functionality. For instance, the <xml3d> element's `getElementByPoint` method provides picking of objects based on screen coordinates as an alternative to picking events:

```
var scene = document.querySelector("xml3d");
var obj = scene.getElementByPoint(300, 200);
if (obj) {
    console.log("Found object!");
}
```

For a detailed documentation of XML3D element interfaces refer to the XML3D specification [SSK16].

5.3.6 Discussion

In this section we discussed the unique abstract model of XML3D which we designed as an extension to HTML. As result of this decision, web developers have all major HTML concepts also available for 3D content: We discussed how XML3D is embedded into HTML and vice versa and how useful CSS properties, the DOM API, and DOM events can also apply to XML3D elements. Consequently, libraries such as jQuery can be used with XML3D "as-is".

We do not support all layout concepts available in HTML. 3D scene elements can be positioned explicitly, relative their parent, not taking their extend into account. We consider positioning 3D objects based on their extend as a useful concept that we leave for future work. In contrast, we cannot image enough use cases that would justify implicit positioning. Hence, it is currently not available in XML3D.

While some CSS properties render useful for XML3D, we do not use CSS properties to define the shading of XML3D objects. The shading models of HTML and SVG are is too limited to describe the variety of possible 3D surfaces and their interaction with light. A possible approach would be to extend the set of available CSS properties to support a specific 3D shading model. However, such an Ubershader would still be to limited compared to the flexibility provided by programmable GPU shaders offered by implicit approaches. Consequently, we introduce the <material> element and an alternative approach to describe the surface of an object based on programmable materials, which we will discuss in Section 5.7. Nevertheless, we stick to CSS in order to assign these advanced material descriptions to 3D objects.

Due to the HTML integration, many of the common scene graph concepts[6] come virtually for free. This includes the scene graph (DOM tree), API (DOM API), event model (DOM events), and the file format (DOM serialization). On top of that, it grants access to powerful technologies such as CSS and CSS Selectors that have not been previously accessible for 3D content.

The lean abstract model of XML3D adds only seven elements to HTML that provide functionality and semantics: <xml3d>, <group>, <mesh>, <model>, <light>, <view>, and

[6] See Section 4.2.

<material>.[7] This fits very well to HTML, which has been rationalized throughout the years, stripping out functionality that can be implemented with CSS and JavaScript.

On the other hand, HTML5 provides richer semantic elements that do not introduce new functionality, but make web pages easier to understand for both, humans and machines. These new elements include <aside>, <header>, <footer>, <nav>, and <section>. One could imagine similar useful elements for 3D content, e.g. a <floor> element that can group elements that represent a scene's floor and would allow a navigation library to use such elements to restrict vertical navigation.

Until now, we did not discuss how to parameterize the XML3D SceneElements, how to animate these parameters, how we can organize large scenes (including delivery to the browser), and how we can describe the variety of possible materials. For these topics we had to develop new concepts that we will discuss in the following sections.

[7] We will introduce additional elements for the definition of data, data sets and assets. These elements, however, do not provide a meaning for the scene on their own.

5.4 Generic Data Model

In the previous section, we left open how to parameterize the introduced XML3D elements. In this section, we will introduce our generic data model which allows parameterizing the <mesh>, <light>, <view>, and <material> elements, but is also the foundation for the data processing concept that we will discuss in the next section.

The <model> element is not considered in this section. Since it instantiates richly structured assets, it requires a specific way to address and parameterize elements within this structure. We will discuss how we approach parameterization (or configuration) of assets in Section 5.6.

For the other elements, we require a parameter approach that supports:

- describing scalars, tuples of scalars, and textures.
- describing arrays of scalars, tuples of scalars, and textures. However, we do not expect deeply nested parameters.
- arbitrarily named parameters. This is a requirement derived from the dataflow graph (Section 5.5) and programmable materials (Section 5.7) concepts.
- describing the way textures are accessed (sampling parameters).
- changing and debugging parameters during runtime easily.
- describing possibly very large arrays efficiently. In particular, we do not wish the usually large-sized per-vertex attributes and index data of geometry to be part of the character-encoded HTML document. In X3D, this kind of data makes up 95 percent of the document on average [BJFS12].
- sharing data between multiple elements. For instance, <mesh> elements could have individual styles and poses but share an identical set of parameters (instancing). In other use cases, an element may share a set of parameters with other elements, but also provides some element-specific parameters (specialization).
- compose data from multiple sources in the sense of a *mashup*. Imagine a Dual Reality factory application where the geometry of a machine is stored in a database and live data from the real machine is provided as a web service. In such a setup, a visualization of the factory should be able to compose the geometry data with the live data from the real factory.
- supports simple mapping to GPU data structures. In particular, we do not want processing (other than parsing) to be required to map vertex attributes and index data to GPU buffers.

In contrast to X3D, we aim for a single consistent approach.[8] In Section 5.2.2, we discussed the four mechanisms used for the parametrization of HTML elements: *i*) Element attributes are not well-suited because they can only store strings and cannot be shared

[8] We discussed in Section 4.3 that X3D has four different ways to declare parameters depending on the characteristics of usage.

between multiple elements. *ii*) CSS properties can apply to multiple elements, but are string-based as well. Moreover, there is only a predefined set of supported CSS properties, thus it is not possible to define arbitrarily named parameters with CSS. *iii*) Similar to images and video in HTML, we could point to an external resource that represents the parameters efficiently. Multiple elements can point to the same resource. Thus, instancing is supported. However, since parameters from an external resource are not exposed in the DOM, they cannot be changed or debugged during runtime. *iv*) Finally, we can describe the elements' parameters using their child elements. This approach is the most flexible as we can define arbitrary elements that describe the parameters. On the downside, all data is exposed in the DOM resulting in very large documents (similar to X3D, where vertex attributes are encoded as child elements of geometry nodes).

We can conclude that none of the existing concepts to parametrize HTML objects meets our requirements on its own. As a result, we developed a novel data model for XML3D combining multiple of these approaches.

To make the model more tangible to the reader, we will first introduce the markup of the data model alongside examples in the next section. After that we will formalize the underlying data structure and discuss the novel characteristics of our approach.

5.4.1 The Markup

In order to define generic data, we introduce a set of additional elements to XML3D's abstract model (see *Data* column in Figure 5.2).

```
<!-- Arrays -->
<int name="index">0 1 2 1 2 3 ...</int>
<float3 name="position">0 1 0 ...</float3>
<float name="temperature">32.0 98.6 ...</float>

<!-- Scalar -->
<float name="transparency">0.5</float>

<!-- Texture -->
<texture name="diffuseTexture" filter="nearest linear">
  <img src="wood.png">
</texture>
```

Figure 5.3: A selection of DataEntry elements defining various parameters.

As Figure 5.3 depicts, we provide several elements that specify the type and precision (e.g. float, int, etc.), and how many elements are contained in a tuple (e.g. float2, float3, etc.) via the element's name. We call these elements *DataEntry* elements. The name of a DataEntry is defined by its *name* attribute. The value of a DataEntry is defined as text content of the element. The data model is generalized to array values, i.e. scalar values are represented as arrays with a single value.

The DOM API can be used to manipulate the values of DataEntry elements. However, since the DOM API is string-based, it requires the implementation to parse the string to set the internally typed value. Therefore, each DataEntryElement provides the *setScriptValue*

method to set the values more efficiently based on TypedArrays [Khr11].[9]

DataEntries of type *texture* are slightly different: For the definition of the texture's image data we can reuse HTML elements. Currently we allow one or multiple <img> elements for static textures, <video> for video textures, and <canvas> elements for modifiable textures as data for the texture (see *HTML* column in Figure 5.2). For the future, we envision also using arbitrary HTML content for the definition of interactive textures. The <texture> element additionally provides attributes such as *filter* and *wrap* to configure the sampling of the textures.

For a complete list of all available DataEntry elements including their attributes and interfaces refer to the XML3D specification [SSK16].

Parameters DataEntries can be used to define parameters for the <mesh>, <material>, <view>, and <light> elements. The applicable parameters depend on the element and its used model. The <mesh> element, for instance, requires a parameter named *position* of type *float3* and has an optional parameter named *index* of type *int*. It may have additional applicable parameters based on the attached material. The <light> element expects an optional entry named *direction*, if the light attribute refers to the predefined *urn:xml3d:light:directional* light model.[10] For a list of applicable parameters for each element refer to the XML3D specification [SSK16]. If parameters are defined that are not applicable, they are simply ignored and no error is raised.

Keys Some use cases require the definition of data series with the same semantic and therefore also with the same type, tuple size and name. All kinds of sampled animations (mesh animations, keyframe-based pose animations, etc.) require such series. For instance, imagine a dynamic meshes sampled at different points in time. The position of the vertices varies over time, but the semantic in the context stays the same. Instead of using similar names (e.g. "position0", "position1", "position2", etc.) to identify such related data, we provide a *key* attribute for all DataEntry elements:

```
<!-- Series of data entries with same name -->
<float3 name="position" key="0.0">0 1 1 ...</float3>
<float3 name="position" key="1.0">2 3 5 ...</float3>
<float3 name="position" key="2.5">8 13 21 ...</float3>
```

Figure 5.4: Defining an ordered series of position data with spatial relation.

The key value supports defining an arbitrary amount of variations of a parameter with the same name in a linearly ordered way. Using a floating point value for the key, we can additionally model the spatial relation between the variations (which would be a temporal relation in the example above). If not specified key defaults to zero. Consumers of the data use the entry with the smallest available key. In Section 5.5, we will show how to interpolate between data entries with the same name using Xflow operators.

[9] For details refer to the XML3D specification [SSK16].

[10] For a detailed discussion on how parameters of meshes, materials, and lights are evaluated refer to Section 6.1.

DataMaps In addition to the DataEntry elements, we introduce the <data> element. The <data> element is a generic container that collects DataEntries and constructs a *DataMap*:

```
<data id="#simple">
 <!-- Arrays -->
 <int name="index">0 1 2 1 2 3 ...</int>
 <float name="temperature">32.0 98.6 ...</float>
 <!-- Scalar -->
 <float name="transparency">0.5</float>
 <!-- Textures -->
 <texture name="diffuseTexture" filter="nearest linear">
  <img src="wood.png">
 </texture>
</data>
```

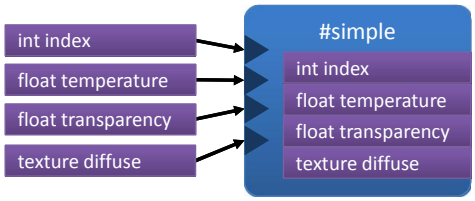


Figure 5.5: The definition of parameters within a <data> element on the left and the resulting DataMap on the right.

The <data> element has no meaning on its own but forms the basis for data sharing and composition. In the following we will discuss the functionality of the <data> element. Note that XML3D elements such as <mesh>, <material>, <view>, and <light> can be considered as specialized <data> elements: They provide the same functionality as the <data> element but add a meaning to the collected data.

Specialization & Sharing <data> elements are the basis for sharing and specialization of data. A <data> element – as well as the specialized data elements – can reference another data element via its *src* attribute:

```
<data id="baseData">
 <float3 name="position">1 0.4 -0.5 ...</float3>
 <float3 name="normal">0 1 0 ...</float3>
 <float2 name="texcoord">0.0 0.0 ...</float2>
</data>

<mesh src="#baseData" id="dataA">
 <int name="index">0 1 2 ...</int>
 <float2 name="texcoord">1.0 0.5 ..</float2>
</mesh>
```

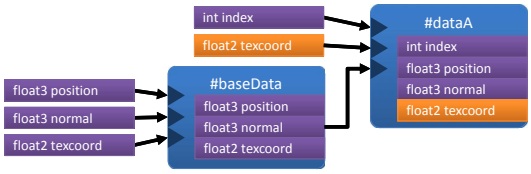


Figure 5.6: Referencing other internal <data> elements and the resulting DataMaps.

The example above shows how the DataMap from a referenced <data> element is merged into the <mesh>'s local DataMap. A set of overriding rules that we will define in the next section allows *specializing* data. In the example above, the local *texcoord* entry overrides the entry with the same name from the referenced <data> element.

Since multiple SceneElements can point to the same <data> element, *src* references allow for *sharing* data from between multiple branches in the DOM tree:

```
<data id="baseData">
 <float3 name="position">1.0 0.04 -0.5 ...</float3>
 <float3 name="normal">0 1 0 ...</float3>
 <float2 name="texcoord">0.0 0.0 ...</float2>
</data>

<!-- Reuse #baseData -->
<mesh id="meshA" src="#baseData">
 <int name="index">0 1 2 ...</int>
</mesh>

<!-- Reuse #baseData -->
<mesh id="meshB" src="#baseData">
 <int name="index">1 11 12 ...</int>
</mesh>
```

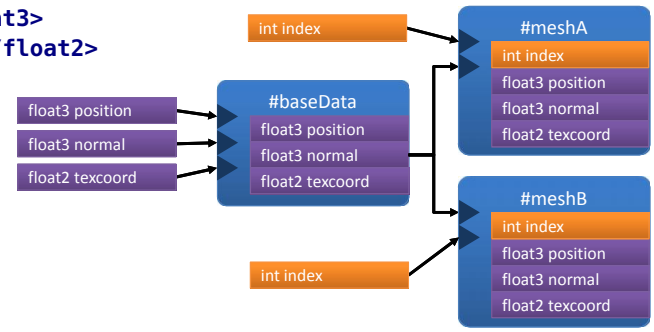


Figure 5.7: Two meshes share a common set of vertex attributes while maintaining individual indices.

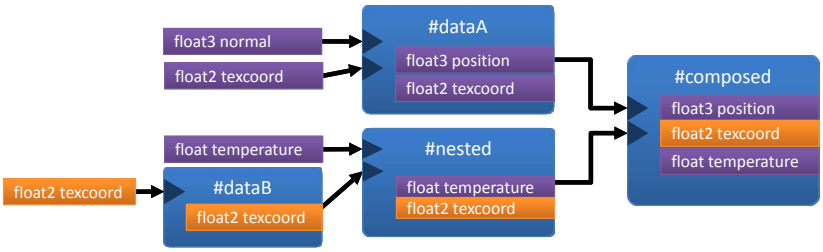
Figure 5.7 above shows a common use case in graphics: rendering a single mesh with multiple different materials. On the GPU, the mesh is represented as a set of Vertex Buffer Objects (VBOs) that contain the mesh's positions and other vertex attributes. To render this mesh, one would bind the VBOs containing the vertex attributes, activate the shader related to the first material, and render the first subset of the mesh based on an buffer containing the indices for the triangles that have this this material attached. Then, without changing the bound vertex attributes, one would change the shader or shader parameters and trigger the second draw call based on a second set of indices (and so on).

This use case is common enough many common rendering libraries offer a specific data structure for sharing vertex attributes between multiple draw calls. For instance, Ogre [JS06] has a *SubMesh* class that has a unique shader attached, but shares buffers with its parent *Mesh*. Similar, Unity's *Mesh* class[11] gives access to a *submesh* collection, which stores indices that have a single material assigned.

In that sense, XML3D's <mesh> is a submesh: it has a single material assigned and the capabilities of our generic data model allow sharing common vertex attributes between meshes without the need to distinguish between meshes and submeshes. In addition, our model not only allows sharing *identical* sets of vertex attributes, but also supports specializing for instance the used texture coordinates per <mesh> instance.

[11] See `https://docs.unity3d.com/ScriptReference/Mesh.html`, last accessed 28 June 2016.

Compositing The <data> element cannot only merge data from a *referenced* <data> element, but also from *nested* <data> elements. DataEntries from nested <data> elements override data entries from previous nested <data> elements and from the referenced <data> element (see Figure 5.8). Since nested <data> elements can in turn reference and nest other <data> elements recursively, one can compose DataMaps from an arbitrary amount of data sources.




```
<data id="dataA">
 <float3 name="position">1.0 0.04 -0.5 ...</float3>
 <float2 name="texcoord">0.0 0.0 ...</float2>
</data>

<data id="dataB">
 <float2 name="texcoord">1.0 0.04 -0.5 ...</float2>
</data>

<data id="composed" src="#dataA">
 <data id="nested" src="#dataB">
   <float name="temperature">20.3 18.2 ...</float>
 </data>
</data>
```

Figure 5.8: DataMap *composed* is composed from referenced element *dataA* and nested element *nested*, which in turn references *dataB*. In this example, the *texcoord* entry originates from *dataB*, since it occurs in *nested*'s DataMap which overrides the previous entries from *dataB*'s DataMap.

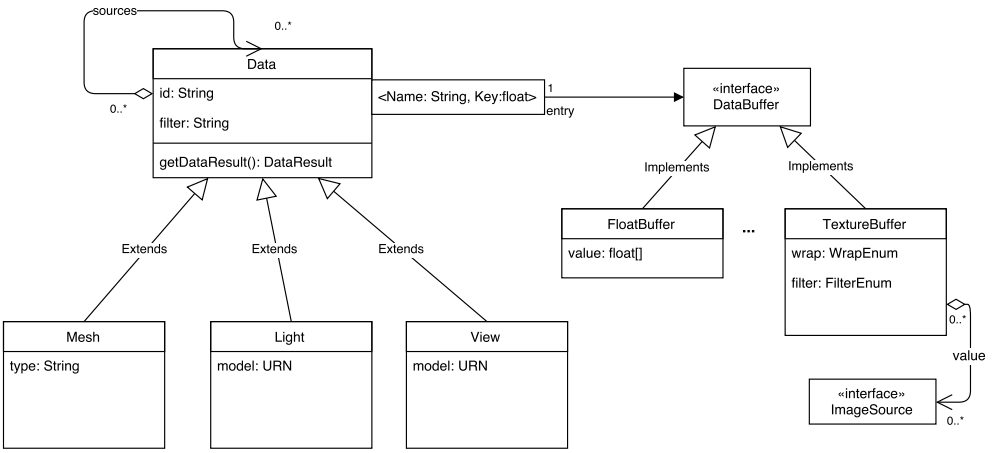


Figure 5.9: Schematic class diagram of XML3D's generic data model.

5.4.2 The Data Model

In the following, we provide a more formal definition of the data structures behind XML3D's generic data model. Figure 5.9 show a schematic diagram of this data model.

Each <data> element maps to an instance of the *Data* class. Note, that the data structure generalizes the concept of referenced and nested <data> elements: A Data instance may reference an ordered collection of other Data instances as *source*.[12] *Mesh*, *Light*, *View*, and *Material* derive from the *Data* class.

The DataEntry elements define a $\langle name, key, type, value \rangle$ tuple. The Data class collects these entries in the *entry* map with the $\langle name, key \rangle$ tuple as the key. Each *type* has its own class implementing the *DataBuffer* interface, which is the abstraction for the storage of the values.

Data Storage In general, the data model does not stipulate where to store the data buffers but leaves it to the implementation. For instance, a JavaScript implementation could store all data values in TypedArrays on CPU. However, the XML3D data types offer many options for mapping data to buffer data structures offered by common graphics APIs which allow storing the data in high-performance graphics memory.

Obviously, images are stored as image data in a *Texture Object* in graphics memory. More recent versions of OpenGL allow storing sampling parameters in a *Sampler Object*. Hence, we can store all information of an XML3D <texture> in such a Sampler Object.

In Chapter 6, we will discuss how we can derive the kind of data (vertex attributes, uniform data) based on its usage. The buffer abstraction of XML3D's data model allows storing vertex attributes in *Vertex Buffer Objects (VBOs)* directly in GPU memory. A Data instance containing multiple vertex attributes map well to *Vertex Array Objects (VAOs)*, which allow activating a set of VBOs with less overhead. A set of DataEntries that are used as uniform attributes can be stored in a *Uniform Buffer Object (UBO)*. Similar to data that can be shared between multiple <material> elements, UBOs can be shared between multiple GPU shaders.

[12] The <data> element referenced via the *src* attribute is, by definition, always the first instance in the ordered source collection.

The newest generation of graphics APIs offers an even more generic approach to data management based on buffers. We expect that these fit even better to our generic data model.

Data Result Each Data class can be requested for its *DataResult*. The DataResult is a data structure similar to the *entry* map of the Data class. In contrast, it not only contains local DataEntries but the union of local DataEntries and DataEntries from the referenced *source* Data instances.

A simple set of override rules defines which data entries to use if ambiguities occur, i.e. if $\langle name, key \rangle$ tuples of DataResults appear multiple times:

1. The DataResults of all Data Instances in the *source* container get evaluated. Note that this will trigger evaluation of DataResults recursively according to the same set of rules until no further references to Data Instances appear.

2. The resulting list of DataResults get merged. Here, DataEntries override DataEntries from preceding Data instances with the same key.

3. "Local" entries, i.e. entries that are defined in the *entry* map override entries contributed by a DataResult from a referenced Data instance.

The principle of local entries dominating entries from referenced data sets allows specialization of data at arbitrary levels.

The DataResult of a Data instance can be influenced by a filter expression. Currently, XML3D supports three kinds of filters: The entries of a DataResult can be *a*) renamed (rename), *b*) kept by name (keep), or *c*) removed by name (remove). In the markup, these filters are defined using the *filter* attribute. For a more detailed description including syntax of the expressions refer to the XML3D specification [SSK16]. The filter functionality is in particular important when setting up data processing using Xflow (see Section 5.5) when fields are not merely statically defined but the result of an operation.

Data Graph and Data Flow Data classes can be composed in a directed acyclic graph (DAG)[13]. A change in a referenced Data instance updates the DataResult of all dependent Data instances. This way, we create a data flow in the opposite direction of the edges of the DAG (see Figure 5.10). Elements such as <mesh>, <material>, and <light> have no incoming references and therefore are data sinks of the dataflow. This dataflow mechanism provides the basis for the dataflow graph concept (see Section 5.5), which additionally allows for processing of the data in each node of the graph.

[13]Neither the data structure nor the markup prevents from creating cycles. In this case, an implementing system should report an appropriate error message.

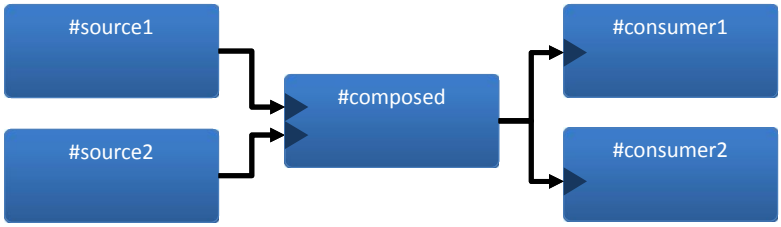


Figure 5.10: Data classes in a DAG: The data of multiple source instances can be composed and data instances can be reused. The figure depicts the direction of the data flow of which is opposite to the direction of the references in the markup.

5.4.3 External Data Resources

The <data> element's *src* element contains a URI, which cannot only point to a <data> element in the same document, but also to external resources or to elements within external resources.

Since the markup is only one way of constructing the generic data structure described above, the external resource is not required to be in XML3D format. As a result, every resource that can map to this data structure, in its simplest form to a single Data instance, can be used as external data resource for XML3D elements. For instance, common 3D mesh data formats such as PLY and STL[14] can be used as a data source for <mesh> elements in XML3D:

```
<group style="material:url(#my-material);">
   <mesh src="./path/to/model.stl"></mesh>
</group>
```

Additional external binary data formats supported by XML3D include the mesh compression formats Open3DGC [Mam13] and OpenCTM [Gee09], and our own delivery format Blast, which we discuss and compare in Chapter 8.

Organizing data from external resources is important to keep the scene description concise, to simplify the organization of a scene, and to reduce the main document loading time and therefor the time to first rendering. Composition of data from multiple source can be found in many web application applications and is called a *mashup* in that context. However, mashups can also be interesting for 3D applications. Figure 5.11 shows how data can be composed from an external API and a static file resource.

5.4.4 Discussion

In this paragraph we described the XML3D architecture's concept for definition, reuse, and composition of parameters for SceneElements. Our data model allows both, using child elements for the definition of generic data inside the DOM and definition of data sets in external immutable resources – depending on whether or not an application needs to modify the data during runtime. On top, our approach allows sharing data sets between multiple objects, composition of data sets from multiple internal and external resources,

[14] A plug-in that provides a mapping for STL resources is available here: `https://github.com/xml3d/xml3d-stl-plugin`.

```
<data id="dataC">
 <data src="http://api.rest3d.com/anim?gS4VlyqTI"/>
 <data src="./external.json"/>
</data>
```

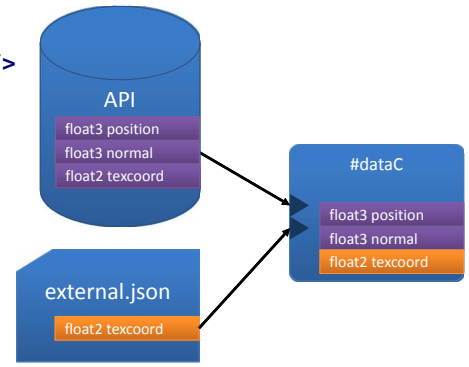


Figure 5.11: An XML3D data set can be composed from various internal and external resource. In this example, the data set is composed from a static file resource and from a REST API call that fetches data from a database.

and specialization of data sets using overriding rules. Both, internal and external resources can be freely combined, because both rely on the same data model.

In the following section, we will discuss an extension to the generic data model that allows describing data processing along the dataflow graph.

5.5 Dataflow Graph Processing

Animations, i.e. modifications of scene parameters over time, are an inherent feature of interactive 3D graphics. In XML3D, animations can be realized using one of the time APIs provided by JavaScript that create repeated callbacks (e.g. `window.setInterval`). Within these callbacks, the scene author can change all scene parameters exposed via the DOM. This includes transformation properties, material parameters, vertex attributes, and data described by the generic data model in general. For many use cases JavaScript is reasonable fast. As shown in [SS11a] for instance, it is fast enough to allow manipulation of thousands of poses per frame on a laptop with an Intel Core i7-840QM CPU.

However, there are other important use cases that require modifying large numbers of e.g. vertex attributes of a mesh or pixels of an image every frame. For instance, the animation of virtual characters and other skinned objects requires interpolating each vertex's position and normal based on the joints of a skeleton. Another common use case is marker based tracking where the camera position and the pose of objects is derived from a marker in an image or a realtime video feed. Both use cases require complex computations per frame. These kinds of computations can easily be parallelized to exploit parallel computing environments. Obviously, in particular graphics hardware offers data parallel computations ideal for many of these use cases. Exploiting parallelism available on modern hardware not only increases the performance of the system, but can also improve the power efficiency, which is in particular important for applications on mobile devices.

In Section 2.1.6 we gave an overview of existing and upcoming APIs that give access to hardware capabilities (WebGL, WebCL, etc.). Web authors can use these APIs and transfer the result of a computation to XML3D. However, this direct use is impractical for multiple reasons:

1. The hardware architectures of target devices are very heterogeneous and thereby also the number of available APIs. Authors need to provide multiple implementations for an operation in order to make sure it runs on every device.

2. Many common operations for animations, such as the modification of transformations or vertex attributes can be computed very efficiently in the vertex shader during the rendering process as part of the rendering pipeline. Another advantage of the computation within the pipeline is that no additional memory needs to be allocated for intermediate results.

3. Some APIs for parallel computing can share resources with graphics APIs, for instance WebCL and WebGL. In contrast to a manual use of the APIs, a solution within the XML3D architecture can leverage the reduced overhead of a direct integration of the computation in the rendering process.

Instead of using low-level APIs directly, we want to provide an abstraction for processing of data that can be mapped onto multiple APIs and hardware architectures.

Dataflow graphs (DFGs) are an established paradigm to compose application-specific functionality from existing building blocks. In Section 4.2 we presented systems that use DFGs to configure the processing required for scientific visualization in a more flexible way. In these systems, each node of the DFG represents an operator. *How* the operator is implemented is left to the implementation.

However, in systems such as VTK, where richly structured data flows along the edges of the processing graph, it is very hard to merge the operations of multiple operator nodes. On the other hand, constructing dataflows with less structured data using Routes in Open Inventor and X3D systems is too complex, because all inputs and outputs need to be explicitly connected and other fields not involved in the operation are not automatically passed along the dataflow (cf. Section 4.3).

Our Approach In order to achieve a high abstraction level for general data processing we decided for a declarative dataflow graph approach, which we call *Xflow*. Similar to previous dataflow approaches, Xflow allows for constructing application-specific functionality from building-blocks (*operators*). Due to its declarative nature, the dataflow graph can be easily mapped to different APIs and hardware architectures.

Conceptually, Xflow is just a small extension to the generic data model described in Section 5.4. This model already allows the definition of Data instances using the <data> element in the markup or from external data sources. Such Data instances can be composed in a DAG with a dataflow along the edges between such instances. Xflow extends this concept by operators that can be attached to Data instances.

The presence of such an operator changes the evaluation logic of an Data instance's DataResult according to the following rules:

1. Evaluate the DataResult as described in Section 5.4 as *input*.
2. Use this input DataResult in order to determine the parameters of the attached operator.
3. The *output* DataResult of the Data instance has all the entries of the input DataResult plus all output parameters of the operator. If the key of an output parameter of the operator already exists in the input DataResult, it overrides the original input entry.
4. All consumers of the Data instance get the output DataResult as input.

Note that also those fields of the DataResult not affected by the operator are implicitly passed to the next data block "as-is". This mechanism makes the definition of dataflows more concise as there is no need to explicitly route fields to later nodes of the DFG.

Operator Expressions In the markup, the used operator, its parameters, and its return values are described in an *operator expression* that is simply attached to the <data> element using the *compute* attribute:

```
<data compute="(output1 [,output2...]) = xflow.[operatorName]([argument1, ...])">
```

Figure 5.12: The syntax of an operator expression is similar to a function call: The operator name is specified after the *xflow.* prefix, followed by a list of actual arguments in parentheses, separated by commas. The return value(s) can be named by assigning them to one or multiple names, separated by commas. Parentheses can be omitted for a single return value.

The functional call syntax allows connecting fields with the input parameters of the parameter, but also the naming of the output fields.

```
<data compute="sum = xflow.add(a, b)" >
 <float3 name="a">1.0 1.0 1.0</float3>
 <float3 name="b">2.0 2.0 2.0</float3>
</data>
```

Figure 5.13: Addition of two vector using Xflow's compute operator.

According to the evaluation rules stated above, the output map of the example in Listing 5.13 has three entries: The original fields *a* and *b* as defined in the markup, and additionally the field *sum* containing the result of the addition.

XML3D comes with a list of commonly used operators for vector and matrix operations, interpolation, and skinning. For the complete list of operators and a detailed description of the syntax of operator expressions refer to the XML3D specification [SSK16]. The xml3d.js implementation allows extending the list of available operators using a plug-in mechanism (see Section 6.2.1).

Because they can be considered as specialized <data> elements, it is also possible to directly attach operators to <mesh>, <material>, <view> and <light> elements.

Processing Graphs The generic data model allows dataflow expressed as a DAG of <data> elements. Xflow extends this capability to data processing with data dependencies described within the DAG, i.e. the input arguments of an operator can be the result of multiple operations and multiple operators can consume the result of an operator. Figure 5.14 shows a <data> element processing the result of a nested <data> element.

```
<data compute="position = xflow.morph(position, posAdd2, weight2)" >
 <float3 name="posAdd2" >1.0 0.0 0.0 ...</float3>
 <float name="weight2" >0.6</float>

 <data compute="position = xflow.morph(position, posAdd1, weight1)" >
  <float3 name="position" >1.0 0.04 -0.5 ...</float3>
  <float3 name="posAdd1" >0.0 1.0 2.0 ...</float3>
  <float name="weight1" >0.35</float>
 </data>
</data>
```

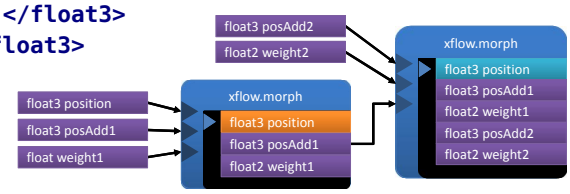


Figure 5.14: Basic construction of an Xflow graph with two nested operators. Both operators modify (and thus override) the *position* entry of the input DataResult. All other entries are passed through the graph without modifications.

Operators on Keys Xflow provides operators that operate on series of data entries with the same name but different keys. For instance, the operators *lerp* and *slerp* apply a linear or spherical linear interpolation on two series based on *key* parameter:

```
<data id="dynamic" compute="position = xflow.lerp(position, key)" >
 <float3 key="2.0" name="position">1.0 1.0 1.0</float3>
 <float3 key="3.0" name="position">2.0 2.0 2.0</float3>
 <float name="key">2.5</float3>
</data>

<data id="static">
 <float3 key="2.5" name="position">1.5 1.5 1.5</float3>
 <float name="key">2.5</float3>
</data>
```

Figure 5.15: The *dynamic* data block interpolates between the both position entries based on the key parameter. The DataResult of the *dynamic* data block is equal to the DataResult of the *static* data block below.

In contrast to other operators, key operators discard all data entries with the target name and output a single entry with the target name (see Figure 5.15). Hence, key operators can be considered as a selection mechanism that can range from simple multiplexing to more complex interpolation or merge operations.

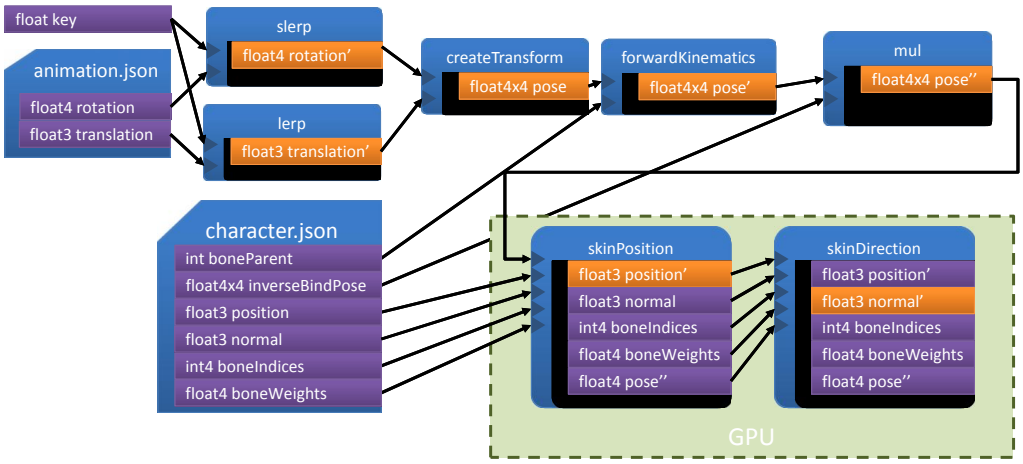
Evaluation of the Dataflow Graph Xflow uses a push+pull evaluation strategy: When a DataMap that is the source of a dataflow gets modified, it is marked for re-evaluation. Then all dependent DataMaps are recursively marked for re-evaluation (push). Since this traversal of the DFG just marks the nodes, the push phase of the evaluation is very efficient and fast.

The possibly heavy computations within the operators are only triggered if a sink requests the data (pull). This lazy evaluation strategy prevents unnecessary computations, for instance if a mesh is not visible. All dependent nodes that have been marked during the push phase get reevaluated. For all other nodes, the implementation may use cached results.

Flexibility The idea of Xflow is to build complex functionality from building blocks. Consequently, the predefined operators are small and generic enough to serve multiple use cases. The various possible combinations of these operators make the system more flexible compared to monolithic abstractions as provided by X3D.

For instance, we decided to not provide a single monolithic skinning operator. Instead, we provide a set of Xflow operators useful for skinning. A skeleton animation driven by a keyframe animation, for instance, can be constructed from seven operators (see Figure 5.16). Varying the combination of operators, the animation can be defined in a pose coordinate system, in local object space, or in world space. Adding tangents to the interpolation – not supported by X3D – is as easy as simply adding another *skinDirection* operator (see Figure 5.17). However, the key difference compared to previous approaches, is that Xflow allows adapting the data processing to the application's needs, i.e. to the kind of input data available for the application.

Constructing complex functionality from small generic operators comes with two disadvantages: Firstly, the dataflow graph tends to become larger and, secondly, the larger amount of intermediate results may hamper the performance of the evaluation of the




```
<data id="characterA" compute="dataflow['http://xml3d.org/skinning.xml#skinning']">
  <data src="./character.json">
  <data src="./animation.json">
  <float name="key">0</float>
</data>
```

Figure 5.16: The markup and resulting DFG for keyframe-based animation on a skinned skeleton. The flow graph is instantiated from an external dataflow template. While all cheap per-joint operations are executed in JavaScript on CPU, the expensive per-vertex operations (within green box) can be executed in the vertex shader on the GPU.

graph. In the following we will present two Xflow concepts to overcome these disadvantages.

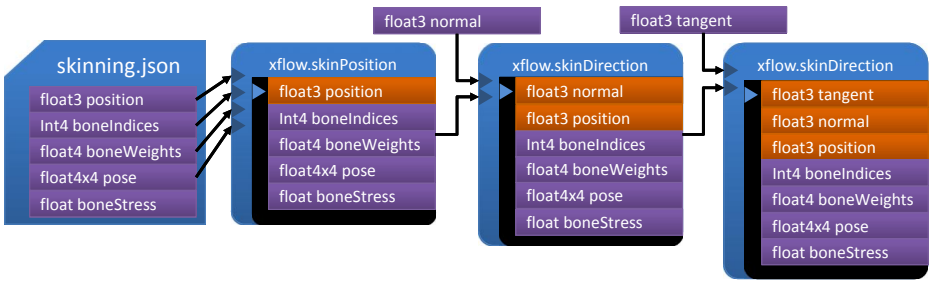
Dataflow Templates In order to avoid large graphs inside the scene description and redefinition of graphs (e.g. for each character to be animated), Xflow provide means to declare dataflow graphs in the sense of a template: Dataflow templates cluster dataflow graphs in reusable units and hence provide an abstraction similar to subroutines in programming languages.

Similar to other XML3D resources, dataflow templates can be defined in external documents and therefore be organized in libraries or requested as a service. Figure 5.16 shows an example, where the character, the animation, and the skinning dataflow are referenced from multiple external resources. For details on dataflow templates refer to the XML3D specification [SSK16].

Mapping to GPU Depending on the sink and the type of used operators, it may be possible to map subgraphs of the DFG to the GPU. For instance, if an entire subgraph operates on individual vertices, Xflow merges the operators, generates a vertex shader and computes the set of input parameters that need to be transferred to the GPU (see Figure 5.18). Figure 5.16 shows, how for instance the interpolation of vertex positions, normals and tangents within a skinning dataflow graph can be mapped to GPU.

The merging of operators into a single vertex shader program not only exploits the multi-parallel execution of the operator, but also eliminates intermediate results.

Not only the skinning algorithm can be decomposed into smaller operators which can




```
<!-- Skinning example -->
<data compute="tangent=xflow.skinDirection(tangent, boneIndices, weights, pose)">
 <float3 name="tangent">1.0 0.0 0.0 ... </float3>
 <data compute="normal=xflow.skinDirection(normal, boneIndices, weights, pose)">
   <float3 name="normal">0.0 1.0 0.0 ... </float3>
   <data compute="position=xflow.skinPosition(position, boneIndices, weights, pose)">
     <data src="skinning-data.json">
   </data>
</data>
```

Figure 5.17: A skinning DFG with *position*, *normal* and *tangent* being interpolated using the skinning algorithm. Adding more vertex attributes to the interpolation can be achieved by simply adding more operators into the graph.

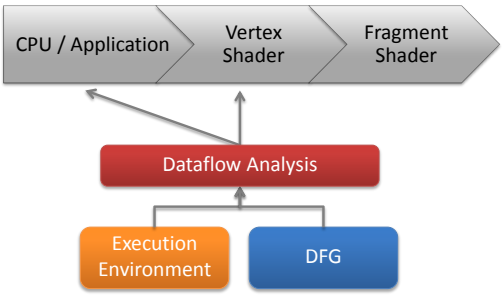


Figure 5.18: The Xflow system analyzes the dataflow graph and its input parameters from the execution environment and maps per-vertex computations to the vertex shader stage of the rendering system.

then be partially mapped to GPU. For instance, we showed that Sequential Image Geometry (SIG) [BJFS12], which allows decoding geometry from multiple images, can be described with Xflow without sacrificing the execution on GPU [KSRS13]. While Behr et al. propose a dedicated <ImageGeometry> node for X3D, we can decompose the functionality into five generic operators (Figure 5.19)[15]. With our approach, we are able to generalize the SIG approach to arbitrary vertex attributes (not provided by the original approach). Even more important, we can combine SIG with other operations. For instance, we can use SIG as input for a skinning dataflow graph. Such a combined graph decodes geometry from images and subsequently interpolates the result. Both sets of operations can be combined and the result can be mapped to the GPU. In contrast, the <ImageGeometry> functionality proposed in [BJFS12] is not combinable with the skinning functionality provided by H-Anim or any other processing node in X3D.

[15]Note that these operations can still be packed into a single dataflow template. Hence, our approach is not necessarily more verbose compared to providing a dedicated node for the algorithm.

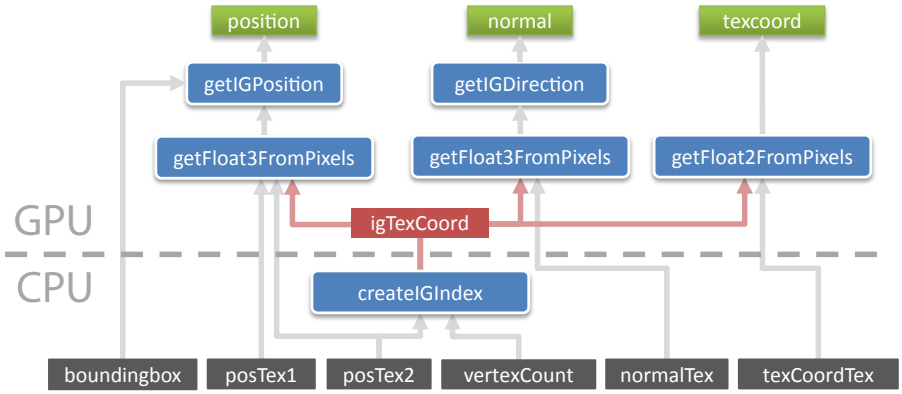


Figure 5.19: The Xflow graph for Sequential Image Geometry. Four textures are used to store vertex positions (16-bit precision), normals (8-bit), and textures (8-bit). Similar to the proposed implementation, a texture coordinate buffer is generated on the CPU based on the *vertexCount* parameter, whereas the values are fetched from the textures and mapped to values in the vertex shader. In contrast to the proposed X3D node, Xflow allows adding more vertex attributes with the desired precision by simply adding additional inputs and operators to the graph.

Rigid Body Animations We discussed in Section 5.2 how CSS 3D Transform can be used to define transformations and how CSS Transitions can be used to interpolate between different transformations. However, for more complex animations, we can use Xflow to describe the required processing. Consequently, we extend the capability of the CSS *transform* property and allow referencing any <data> element that has a parameter named *matrix* of type *float4x4*:

```
<data id="t1" compute="matrix = xflow.createTransform(translation, rotation);">
  <float3 name="translation">3 3 3</float3>
  <float4 name="rotation">0.707 0 0.707 0</float4>
</data>
...
<mesh style="transform: url(#t1);">...</mesh>
```

Figure 5.20: Using Xflow to create a geometric 3D transformation from a rotation and a translation. The result can be assign to any SceneElement via the CSS transform property.

Allowing transformations consuming the output of dataflow graphs, we can use Xflow to describe for instance keyframe-based rigid body animations similar to X3D's PositionInterpolator and OrientationInterpolator nodes [Web04a]. However, we provide a more flexible approach that allows deriving the transformation from arbitrary data processing. For instance, we used this mechanism to derive the transformation of the teapot in Figure 5.21 from an AR algorithm detecting the pose of a marker in a video in [KRS*13]. Again, the flexible approach based on building blocks allows modeling a dataflow graph that lets the teapot jump procedurally from one marker to the other.

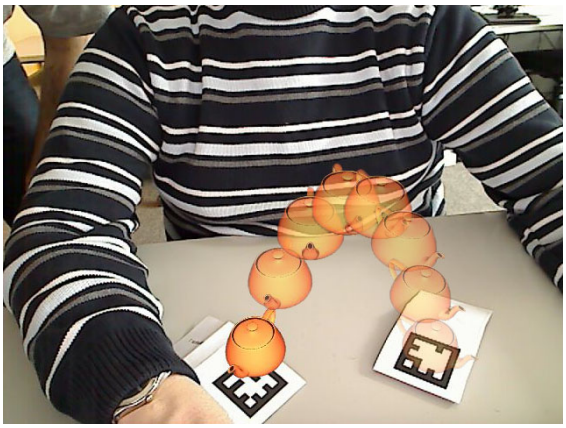


Figure 5.21: The displayed teapot jumps from one visible marker to the other. Both, the camera parameters as well as the transformation of the teapot are derived from the pose of the marker in the video using Xflow.

5.5.1 Discussion

The declarative dataflow approach is an important concept for the XML3D architecture, because it offers a general and platform-independent abstraction for data processing. It is capable to solve the majority of known important use cases (e.g. rigid body animations, mesh animations such as skinning and morphing, and image processing e.g. for Augmented Reality applications [KRS*13]), and is flexible enough to also solve many future use cases.

The key characteristics of Xflow include:

1. Xflow's declarative approach allows mapping data processing to multiple parallel compute environments. We have shown this for the vertex shader stage of the GPU and for Parallel JavaScript [KRS*13].

2. The approach to pass data not involved in operations to subsequent operators keeps the description more concise and better manageable compared to explicit routing in X3D. It is also a good fit for stream processing.

3. In contrast to X3D/VRML Routes and other previous approaches in graphics, Xflow operates on the shallow structured data types of the generic data model with its direct mapping to GPU buffer. As a result, Xflow graphs can be better analyzed, operators can be merged and computations can be mapped to available hardware.

4. Xflow allows constructing complex operations from building blocks. We have shown this for animated skinned characters, the SIG algorithm, and for Augmented Reality applications. In contrast to related approaches based on high-level abstractions (X3D, COLLADA, Ogre, OpenSceneGraph, etc.), operations can be combined in an orthogonal way.

5. Using dataflow templates, it is possible to represent complex operations in a concise and reusable way.

6. A key characteristic of Xflow is the ability to adapt the DFG, and hence the scene description, to the characteristics of the application's input data. We have shown this for animation data, which can easily be provided in different coordinate spaces without the need to convert from one representation to another upfront. We will discuss the novel application model resulting from this ability in more detail in Section 10.2.

We did not provide a in-depth discussion on the underlying dataflow analyses, transformations and compilations of the DFG to several platforms. It has been intentionally omitted as it is, as mentioned previously, research mainly conducted by Felix Klein for his PhD thesis.

5.6 Asset Instancing

The generic data model concept (Section 5.4) provides means for instancing data used as parameters for SceneElements. Hence, the heavyweight data, e.g. the vertex data of a <mesh> element, can be shared between multiple consumers. However, if a scene author wants to reuse whole subscenes – i.e. hierarchical organized SceneElements – instancing merely on data level is not sufficient: The scene author can reuse all parameters defined using the generic data model, but has to replicate the subscene's structure for each instance.

This replication comes with a number of drawbacks: the document grows in size and – as a result – is harder to handle in editors. Additionally, the loading time of the document as well as the memory cost increase.[16] On top, the author has to take care about the ids of the replicated subscene, i.e. ids need to be mangled for each copy and references need to be updated.

As we have shown in Chapter 4, a common approach for reusing structured data is using a DAG data structure that allows referencing an exact copy of the subscene multiple times. In the same chapter, we discussed the many disadvantages that come with this approach. The major issue for the DOM integration is that a node can represent multiple rendered objects, which requires paths for unique object identification. The DOM event model has no notion of paths but a single event target. Hence, using a DAG data structure renders impossible using the DOM event model as is.

Additionally, reusing identical subscenes is overrated in terms of its practicability for scene management. It might be useful for small subscenes (such as the legs of a chair); for more convincing scenes, however, visual variance of individual instances is required (e.g. for virtual worlds with hundreds of virtual humans). Moreover, instances often need to be animated individually (e.g. the wheels of a car are animated according to the individual speed and heading of the associated car). For these use cases, simple references to subscenes fall short.

On the other hand, there are component models such as X3D's Prototypes (see Section 4.3), which allow higher-level instancing of subscenes that not only contain data structures, but also behavior, i.e. scripting and events. Such a higher-level instancing model provides a good abstraction for arbitrary complex content and supports configuration of the instantiated scene through an interface. However, the complexity of these approaches (an internal script may modify the content arbitrarily) makes them inherently inefficient to instantiate. Instead, implementations fall back to replication of the internal structures, which is slow and memory consuming leading to similar issues as discussed for copying subscene manually above.

In order to fulfill our design goals, we aim for an approach that *a*) provides an abstraction over subscenes that allows reusing structured data including geometry, materials, and animations, *b*) allows configuring subscenes individually without exposing its structure, *c*) is simple enough to identify the configurations and hence can reuse all other resources, and *d*) provides a secondary namespace mechanism to prevent any kind of id conflicts.

[16] In particular since DOM elements have a large memory footprint due to their rich set of functionality, it is a good practice trying to keep the number of DOM elements small (cf. [KSSS14]).

Our Approach We have chosen to introduce *assets*, a data structure for the definition of subscenes which can be instantiated from either the main scene or recursively from other assets. We combine this data structure with a new scoped addressing schema and with a powerful and generic configuration mechanism.

Assets and Models An asset can be defined using the <asset> element which may collect arbitrary SceneElements similar to the <xml3d> element but without actually rendering it.[17] Another difference to the main scene is that all SceneElements are prefixed with "asset".[18] An asset with two meshes for instance can be defined as such:

```
<asset id="exampleAsset">
  <assetmesh style="material:url(#wood);">
    <!-- Mesh data declared here -->
  </assetmesh>
  <assetmesh style="material:url(#iron); transform:rotateY(45deg);">
      <!-- Mesh data declared here -->
  </assetmesh>
</asset>
```

An asset can be instanced from within an XML3D scene using the <model> element, which is placed inside the scene graph and which refers to an internal or external asset using a URI in the *src* attribute:

```
<group style="transform: translateX(5px)" >
    <model src="assets.xml#exampleAsset" ></model>
</group>
```

Consequently, all meshes declared in the asset will be visible in the scene, correctly transformed, and with materials attached. Note that the <model> element is the *target* element returned by the DOM event system if any of the asset meshes is picked by the user. This is important in order to remain compatible with the DOM event model: With the asset model, multiple <model> elements can point to the same asset. If an <assetmesh> from within the asset would be returned, path information would be required to identify which <model> element actually initiated the rendering of the mesh. Since the DOM event model is designed for the DOM tree structure and has no notion of paths, we return the triggering <model> as the event's target element.

Data and Namespaces Obviously, the <assetmesh> element is based on the generic data model and hence it is possible to compose, share, and specialize the data of asset meshes from internal and external documents in the same way. However, in addition to the functionality derived from the generic data model, we provide a specific data sharing concept for assets introducing the <assetdata> element and the *name* and *includes* attributes:

```
<asset id="sharedExampleAsset2">
  <assetdata name="shared" >
```

[17] The assets described in [KSSS14] and the current implementation are restricted to <mesh> and <data> elements, but this is not a conceptual restriction.

[18] We introduce this to better distinguish between SceneElements defined in an asset and in the main scene (see difference in addressing below). However, this convention might being dropped in the future.

```
    <!-- Shared vertex buffers -->
  </assetdata>
  <assetmesh name="woodMesh" style="material:url(#wood)" includes="shared" >
    <int name="index" >0 1 2 1 2 3 ...</int>
  </assetmesh>
  <assetmesh style="material:url(#iron)" includes="shared" >
    <int name="index" >1045 1046 1047 ...</int>
  </assetmesh>
</asset>
```

In this example, the content of <assetdata> is shared among the two declared <assetmesh> elements. It is also possible to refer to multiple <assetdata> elements by having white space separated names in the includes attribute.

Introducing a secondary addressing mechanism seems to be redundant. However, while document ids are in a single namespace for the whole document, the name/include references reside in a namespace scoped by the <asset> element. As a result, we have the option to define several assets in a single document without running the risk of having name collisions. This is in particular important if collections of assets are generated by a server, for instance as result of a database query: Using scoped namespaces, there is no need to perform any kind of name mangling before sending the result to the client.

Configuration In order to support configuration of individual model instances, we add a powerful *extension mechanism* to our <asset> elements. An <asset> element can extend a source <asset> element referred to by the *src* attribute. Consequently, all declared <assetdata> and <assetmesh> elements modify equally-named elements of the source assets: It is possible to add DataEntries to the addressed source element or to override existing DataEntries.

In the following example, we extend the asset from the example above by adding a new color attribute to the *shared* asset data:

```
<asset src="#sharedExampleAsset2">
  <assetdata name="shared">
      <float3 name="color">0.8 0.0 0.1</float3>
  </assetdata>
</asset>
```

The original source asset contains <assetmesh> elements that refer to the <assetdata> element that we configure. Since we now extend the shared <assetdata>, the new color buffer will be available for these meshes. Note, that this behavior is due to the semantics of the *includes* attribute: It does not include only the available data at declaration time, but also the data from the configuration. This means that, before we connect an asset entry to another via *includes*, we *first* recursively resolve its own includes-connections and extension chain. Similar to the generic data model, entries that exists deeper in the hierarchy, will be overridden. For an <assetmesh> entry, we can additionally overwrite the attached

material, transformation and all other CSS properties. In this example, we override the material and the color of an individual mesh from the asset definition above:

```
<asset src="#sharedExampleAsset2">
  <assetmesh name="woodMesh" style="material:url(#myWood)" >
      <float3 name="color">0.2 0.2 0.1</float3>
  </assetmesh>
</asset>
```

Finally, not only asset definitions can be extended, but also asset instantiations can extend the referred asset using exactly the same syntax:

```
<group style="transform: translateX(5px)" >
  <model src="assets.xml#sharedExampleAsset2">
    <assetdata name="shared" >
      <float3 name="color">0.0 0.5 0.0</float3>
    </assetdata>
  </model>
</group>
```

This extension mechanism enables the definition of 3D models with complex data dependencies in a self-contained asset without losing the option to extend individual meshes and materials. However, it not only allows configuring meshes, but arbitrary <assetdata> elements, which may of course include dataflow processing. This way, it is possible to configure data processing for each instance of an asset individually:

```
<model src="./resources/assets/avatars.xml#sniper">
  <assetdata name="animation">
    <float name="key">0.5</float>
  </assetdata>
</model>
```

Figure 5.22: Instancing an asset containing multiple meshes and materials, and a skeleton-based keyframe animation. The keyframe animation can be controlled with a single exposed parameter using the configurable asset approach. A screenshot of the scene is displayed on the right.

Recursive Assets An asset can recursively contain multiple subassets. Subassets can address <assetdata> elements from their parents using a "parent." prefix in the name. This mechanism allows defining hierarchical asset structures with or without dependencies between parent assets and subassets. For instance, a house with multiple meshes and materials could be a self-contained asset, but also be a subasset of a whole city asset. The houses of the city asset could in turn derive a global configuration (for instance a base size or color) from the city asset.

(a) Plain instantiation of a glTF asset.

(b) Instantiation of the same asset, but overriding the material of the chassis geometry with a procedural shade.js material.

Figure 5.23: Instantiation and configuration of glTF assets in XML3D.

Generalization Similar to the generic data model, assets are not restricted to the presented markup. All structured 3D formats can be used as a model in XML3D, providing that it can be mapped to XML3D SceneElements including the asset-specific naming and include schemes, dataflow processing, and attachment of materials to meshes via URIs. For instance, glTF can map to XML3D models[19]:

```
<model src="./resources/MilkTruck.gltf"></model>
```

Moreover, we can use the specialization mechanism of assets to also configure and specialize glTF assets, a functionality not natively supported by glTF. For instance, one can replace the material of a specific mesh inside the asset like this:

```
<model src="./resources/MilkTruck.gltf">
  <!-- Adresses mesh named Geometry-mesh002 and replaces material -->
  <assetmesh name="Geometry-mesh002" style="material: url(#mat);"></assetmesh>
</model>
```

Figure 5.23 show the two examples above in comparison.

5.6.1 Discussion

Facade Pattern With assets as an encapsulation mechanism, we can instantiate an arbitrary structured asset from a single <model> element and still configure all exposed, i.e. named, elements using the configuration mechanism described above. Figure 5.22 shows how a complex animated character can be instantiated and animated by modifying a single parameter, whereas all other aspects of the complex asset are hidden from the user. The exposed parameters, i.e. the collection of named parameters, can be considered as the asset's interface. Hence, the asset element implements a facade pattern as defined in [GHJV95]: It shields the author from subscene details, thereby reducing the number of objects to deal with and making the subscene easier to use.

[19] A plug-in that provides a mapping for glTF resources is available here: `https://github.com/xml3d/xml3d-gltf-plugin`.

Compared to full-fledged component models, assets cannot have internal scripts that control the instance on their own. However, since Xflow is an integral part of the asset's data structure, the assets are not restricted to geometry, appearance and transformations, but may also include arbitrary data processing. Exploiting these data processing capabilities, we can adapt the interface of assets to minimize the mapping from the application model to the 3D model (see Section 10.2). As a result, the asset can be configured based on a reduced and well-defined set of parameters. For instance, if a traffic simulation describes the heading of each car using a vector, we can use this vector *as is* for each car asset instance, and map this vector internally to a 3D rotation necessary to transform the wheels using Xflow.

This data-driven approach allows constructing a dataflow graph that represents the data dependencies between instances, i.e. with shared data between instances and optional specialization per instance. We can reuse the dataflow graph functionality, i.e. the dataflow graphs constructed for assets get analyzed, optimized and possibly mapped to the GPU.[20] As a result, we get very close to the functionality of component models without the need to copy the whole internal structure per instance.

Limits Our asset approach reuses the Xflow dataflow graph approach to optimize the instances in terms of memory and runtime performance. In general, Xflow is optimized to make changes of exposed and therefore changeable data entries cheap. However, if the user changes the structure of the asset or of the overrides, i.e. if the user adds or deletes elements, and therefore also the structure of the dataflow graph, the implementation needs to reanalyze the graph, which is more expensive.

For all use cases that were naively designed requiring such structural changes we were able to restructure the asset in a way that avoids structural changes. However, this requires some experience and a novel users might run into performance issues by using the asset mechanism in a disadvantageous way. We discuss the possibility to author scene with poor performance also when reviewing our design criteria in Section 10.1.

Although the combination of configurable instancing with the expressiveness of Xflow results in a very flexible scene encapsulation mechanism, it intentionally prevents full freedom provided by component models that offer internal scripting. However, Web Components[21], offers a collection of technologies that allow defining reusable components for the web including custom elements with internal scripting. We expect that XML3D can be adapted to be usable with Web Components at little cost and initial research in this direction has been conducted in [LSSS16].

Conclusion With assets we provide an encapsulation mechanism for subscenes for the XML3D architecture. We designed the asset structure carefully to be able to instantiate 3D models with maximum reuse of resources while maintaining configurability adapted to the application's needs.

[20] Again, we do not discuss the analysis and optimization of the dataflow graph as it is research conducted by Felix Klein.

[21] See Section 2.1.7.

5.7 Materials

In HTML5, CSS properties are used to describe the appearance of HTML objects. These CSS properties can be used to *configure* the predefined HTML shading functionality.

In contrast, GPUs made the transition from configurable predefined functionality to fully programmable shading (see Section 2.2). It is used, among other aspects, to achieve the multifaceted appearances required to describe scenes convincingly. Consequently, we deem necessary programmable materials for Dec3D in order to be competitive with graphics APIs in terms of rendering quality and flexibility.

However, as we have shown in Chapter 4, 3D file formats, 3D toolkit libraries, and game engines are all stuck in the same material dilemma: In 3D graphics, there is no programmable material description format, which is portable and adaptive to varying scene data.

Using platform-specific GPU shaders to define materials comes with all the issues discussed at length in Section 4.1 and conflicts strongly with our design goals. It is thus essential to solve this dilemma because providing a set of predefined material models is not sufficient for many use cases and, hence, does not meet our design criteria.

Our Approach For the XML3D architecture, we decided to offer both: a set of predefined material models already sufficient for a wide range of applications, while for customized materials, we additionally provide a novel imperative material description language: *shade.js* [SKSS14].

```
<material id="myMaterial" model="urn:xml3d:material:phong">
 <float3 name="specularColor">0.8 0.8 0.8</float3>
 <float3 name="diffuseColor">0.6 0.2 0.2</float3>
 ...
</material>
...
<group style="material: url(#myMaterial)">
 <mesh src="teapot.json"></mesh>
 <mesh src="teapot.json" style="transform: translate3d(...);">
  <texture name="diffuseTexture">
   <img src="metal.jpg">
  </texture>
 </mesh>
</group>
```

Figure 5.24: Predefined materials in XML3D are referenced via URN in the material's *model* attribute. The material is then attached to a SceneElement via CSS. In this example, two <mesh> elements share the same material that has been attached to a <group> element. The second mesh, however, specializes the material by defining a mesh-specific *diffuseTexture*.

Predefined material models can be referenced using a URN (see Figure 5.24). For a list of currently supported predefined material models, refer to the XML3D specification [SSK16]. For instance, XML3D provides a material that has an ideal diffusely reflecting surface (urn:xml3d:material:diffuse) and one that additionally applies the specular Phong reflection model [Pho75] (urn:xml3d:material:phong). The predefined material models are Ubershader that support a number of variations based on the type and existence of certain pa-

rameters. For instance, most coefficients of these material models can be either uniform or the result of a texture fetch (e.g. *diffuseColor* and *specularColor*). The required specialization of the target shader is handled by the XML3D implementation.

For *custom material descriptions* we developed shade.js. shade.js is based on a subset of JavaScript. Similar to OSL [GSKC10] it has interfaces to the lighting system based on radiance closures. As a result, it does not mix-up the two distinctive concerns of lighting and surface shading and we can use shade.js to exclusively describe the latter.

Our language supports inspection of its execution environment and, hence, can adapt to available parameters and functionality provided by the current execution platform. Additionally, shade.js exploits the polymorphism that comes with the non-explicit declaration of types in JavaScript. These two novel features allow for writing adaptable and thus more general material descriptions that can be organized in reusable libraries and shared across the Internet. Figure 5.25 shows how a shade.js material is defined in XML3D.

```
<script id="model" type="text/shade-javascript">
 function shade(env) {
    var shininess = env.shininess || 0.25
    var kd = env.kd || new Vec3(0.8);
    ...
    if (env.kd && env.kd.sample2D && env.texcoord) {
        kd = env.kd.sample2D(env.texcoord).rgb();
    }
    ...
    return new Shade().diffuse(kd, env.normal)
                      .phong(ks, env.normal, shininess);
}
</script>
...
<material id="myMaterial" model="#model">
 <float3 name="ks">0.8 0.8 0.8</float3>
 <float3 name="kd">0.6 0.2 0.2</float3>
 ...
</material>
...
```

Figure 5.25: Definition of a material using shade.js. The shade.js compiler specializes the material by removing all code not applicable for the current execution environment. Note that both, the <material> definition as well as the <script> element containing the programmable material are referenced via URL and could therefor also be defined within an external resource. This allows to collect both, material models and materials, in material libraries.

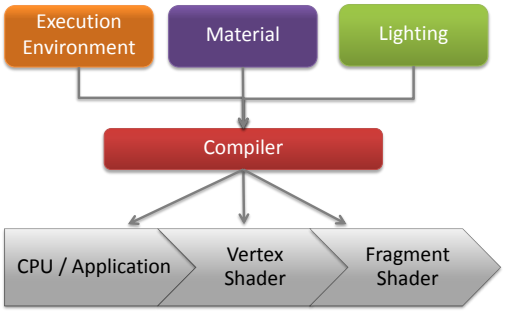


Figure 5.26: The shade.js compiler specializes the material based on the current execution environment. The specialization functionality can also be applied to the implementation-specific lighting code (written in shade.js) or alternatively map to native functions provided by the rendering system. The compiler is able to extract uniform and per-vertex expressions and maps the computation of these expressions to the appropriate stages of the rendering system (e.g. to vertex and fragment shaders is GLSL).

Based on a specific execution environment given at runtime, the accompanied compiler generates specialized shader code that is optimized for the target rendering system and algorithm (see Figure 5.26). Finally, the compiler generates implementation-specific code in the target shading language. Currently, the shade.js compiler supports GLSL and OSL.

We describe shade.js in detail in Section 7.5. There, we also evaluate shade.js with examples targeting four different rendering approaches (forward and deferred rasterization, simple Whitted-style ray tracing, and global illumination) and show that we can improve convenience and flexibility for specifying materials without sacrificing performance and expressiveness.

5.8 Data Delivery

A web page mainly consists of textual content (HTML, style sheets, scripts) and binary content (images, audio, and videos). Textual content typically gets minified [RLZ10] and then compressed using HTTP's compression capabilities before transmission to the client (e.g. GZIP [Deu96b] or DEFLATE [Deu96a]).

Images comprise more than half of an average web page's payload[22]. The HTML <img> element supports common image formats such as JPG, PNG, and GIF. Some of these formats support progressive transmission, i.e. the viewer of the image gets an early visual feedback. This is achieved by either showing those pixels that have already been received or by showing a rough image in the beginning while improving quality when the remaining data is received by the browser.

The <audio> and <video> elements provide support for playing audio and video. A standardized container is used to transmit video and audio (e.g. MP4 [ISO05]) and a selection of common codecs exists to compress the content (e.g. H.264 [ISO03] for video).

In contrast to images, audio, and video, 3D data is much more structured. For the transmission of 3D data, no common container formats exist and there are no established and general compression schemes. There are some established algorithms for progressive streaming of geometry (e.g. progressive meshes [Hop96]), but again no established container format is available to stream the required data to a client.

For XML3D, we require a streamable container format to efficiently deliver 3D data to the client. Additionally, this format should support compression in a flexible manner to take into account the heterogeneity of 3D data.

Our Approach With *Blast*, we propose a novel binary container format for structured data. The format consists of *chunks* that are self-contained. The chunks can be streamed and individual received chunks can be decoded on the client, for instance in a separate thread. For the compression of the chunks, we decided for a code-on-demand approach: Each chunk uses a URL to identify the decoder. In the event that a system does not have the required decoder available, it can download a default implementation from the URL. This approach allows using very specific compression schemes without "baking" them into the format. Similar to XML, we provide a mechanism to identify individual objects in a container. This way, we can transmit multiple resources in a single request.

Blast is generic and not restricted to 3D resources. It is specifically designed (but not limited) to work well with Web APIs such as the Web Workers API [W3C15b] for parallel decoding, the WebSocket API [W3C09e], and upcoming APIs for streaming [W3C14e].

Although Blast is not tied to XML3D, its generality fits very well to XML3D's generic data model. Additionally, we provide Xflow dataflows for a set of well-established decoders, and Xflow may map parts of the reconstruction to the GPU. This not only reduces the CPU load but may also eliminate (possible large) intermediate results that otherwise would have a copy in CPU memory. For instance, if a Blast encoder quantizes data to reduce the payload, one can describe the dequantization as an Xflow operator. If suitable, the Xflow system maps this operator to the vertex shader and the quantized data block

[22] According to `http://httparchive.org/`, accessed 18 December 2015.

gets transfered to the GPU in a buffer *as is*. This way, it is possible to entirely bypass the dequantization in JavaScript. We discuss the details of Blast in Chapter 8.

5.9 Discussion

In this Chapter, we presented the major novel concepts of the XML3D architecture. It covers all the aspects required for the presentation of interactive 3D scenes in a web browser: From the actual declarative scene description including dynamic aspects, encapsulation and compositing of unstructured and structured data, to the description of materials and the efficient transfer of structured (scene) data. Figure 5.28 shows an XML3D scene using the core concepts of the architecture.

Design Goals We presented an orthogonal set of independent but interoperable concepts that are not motivated just by themselves. Instead, these concepts are the building blocks required to put together the architecture that meets our design goals as set up in Chapter 3.

In contrast to previous approaches, XML3D has a single concept for (shallow structured) parameters used for the whole abstract model and for its related concepts. This increases the consistency and generalizability of the architecture. Hence, this concept contributes to the learnability. Additionally, it fulfills our requirements on the organization of generic data: It gives developers fine-granular control on organization of data for an application and hence contributes to the design criterion "Scene Organization". Finally, the ability to instance data reduces the memory requirements of the application and therefor contributes to our requirements on memory performance.

Xflow allows constructing functionality from reusable basic blocks. Hence, it contributes to our requirements on flexibility and scene organization. We have shown, that that we can map subgraphs of an Xflow graph to the GPU. As a result, we can exploit multi-parallel hardware without the need to sacrifice platform independence by using specific shaders. Hence, Xflow also contribute to our performance requirements.

Similar, the asset instancing concept contributes to design aim "Usability" and in particular contributes to the organization of scenes and their performance.

The programmable material concept contributes significantly to "Renderer Independence", but also increases flexibility and performance of the architecture. Since shade.js is based on JavaScript – the lingua franca of the web – it is also better integrated into the web compared to other approaches.

With Blast we can represent and deliver data efficiently and hence it decreases the page loading time. Thus, Blast contributes to the performance of the XML3D architecture. Blast also increases the user experience of XML3D by reducing start render times: the self-contained data chunks can be streamed to the browser, decoded in parallel, and rendered before the whole container is delivered to the browser.

Table 5.1 outlines, how the concepts of the XML3D architecture contribute to our design criteria. After evaluating the XML3D architecture based on various application examples in Chapter 9, we will review the design goals of XML3D in more detail in Chapter 10.

Conflicting Design Goals In Section 3, we discussed conflicting design goals for the XML3D architecture. For instance, flexibility and simplicity are conflicting design goals

	HTML5 Integration	Abstract Model	Generic Data Model	Dataflow Graph	Asset Instancing	Programmable Materials	Data Delivery
Web Integration	X	X				X	
Renderer Independence		X		X		X	
Usability	X	X	X		X	X	X
Expressiveness	X	X		X	X	X	

Table 5.1: This mapping shows how the concepts of the XML3D architecture contribute to our design goals.

and trade-off needs to be make.

Concepts such as Xflow and shade.js provide a novel degree of flexibility in XML3D missing in previous declarative approaches. Constructing dataflow graphs for instance for skinning algorithms or algorithms that fetch geometry information from images are tasks a novice user is likely not going to achieve. Similar, we expect that only expert users will use shade.js to create advanced materials.

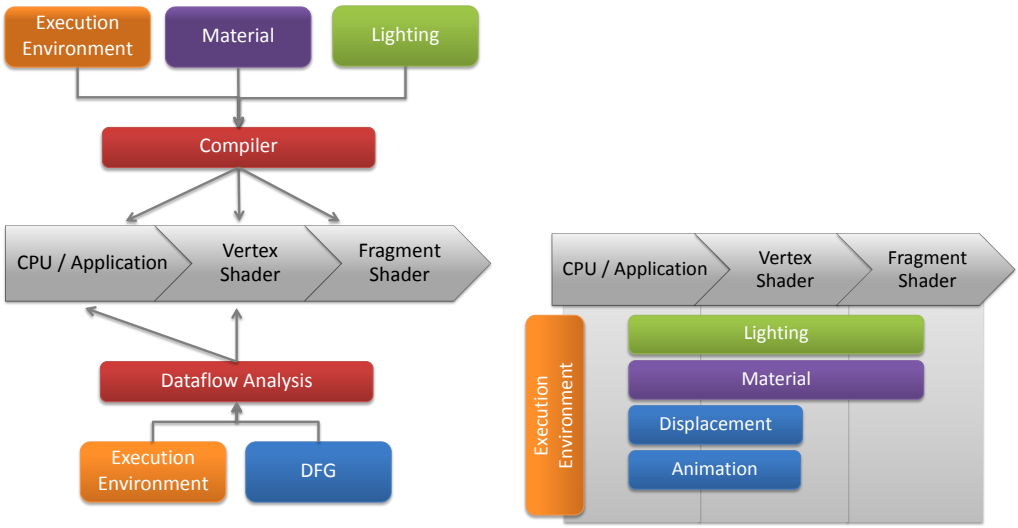
However, since both, dataflow graph templates as well as shade.js material descriptions are self-contained and can be encapsulated (e.g. organized in libraries), it is not necessary to be able to achieve this tasks in order *to use* these advanced concepts. In addition, XML3D's asset concept allows encapsulation of subscenes (including dataflow processing and shade.js materials) into reusable resources with a well-defined interface. These concepts shield users from the underlying complexity of a material description, dataflow graph, or a whole subscene without preventing the user to get into the details if wanted or required. This fits very well to our aim to make simple things simple but complex things still possible.

We were aware that the targeted higher abstraction level (implied by renderer-independence and HTML integration) conflict with high performance execution and flexibility. This was a key conflict to solve in order to reduce the gap between declarative scene descriptions and imperative 3D toolkits.

In the case of programmable materials, we solve the conflicting goals between a portable and hence more abstract material description versus high performance due to high specialization using compiler technology. Similar, we use a dataflow graph to describe data processing declaratively and then apply dataflow analysis in order to map these high-level descriptions to specialized vertex shaders and APIs that offer data parallelism. Figure 5.27 shows how both concepts map their computations to the different stages of a rendering system.

As a result, we expose the power of programmable graphics hardware to the scene author without the need to expose the low-level rendering pipeline stages. Thus, logically distinct aspects such as materials, lighting, animations, and displacement can be specified separately in the high-level scene description. Despite this separation, the system is able to generate efficient per-stage shaders without sacrificing performance.

Web Compatibility The data structures and the abstract model were carefully designed to take into account the specifics of the DOM tree data structure and the DOM event model, but on the other hand provide ways to instantiate 3D data and 3D assets efficiently. We want to emphasize that we managed to design XML3D to meet both requirements: reuse



(a) (b)

Figure 5.27: On the left: Combined mapping of functionality from programmable materials and the dataflow graph to the stages of the rendering system. The right side shows where the different aspects of the XML3D architecture are implemented. The system implements the lighting (green), shade.js implements the materials (violet) and Xflow implements animations and displacement (blue).

of data and scene elements while maintaining a tree structure as main data structure. This was achieved following a set of principle rules:

1. All XML3D elements are **organized in the DOM tree**, with the DOM parent-child relationship as first-level references.

2. XML3D provides no means to instance SceneElements. As a result, **each rendered object is uniquely identifiable** and corresponds to exactly one element in the DOM tree. This behavior is identical to HTML. We found it essential to retain this behavior, specifically with regard to the many DOM libraries that rely on a single leaf element as a target of events. Additionally, since each renderable object has only one parent, CSS inheritance rules remain simple.

3. All reusable XML3D entities, i.e. data elements, assets, materials, material models, and dataflow graph templates, are modeled as resources that can be referenced via URI from either attributes or CSS properties. Consequently, it is possible to define them either within the same document, or as **external resources**. This allows organizing these entities in libraries or gather them from web services.

4. The **generic data model** is used consistently in XML3D's abstract model. Its data structures are based on typed arrays and map well to recent graphics APIs without unnecessary preprocessing of the data.

5. Data elements can reference other data elements recursively. As a result, the data elements are internally represented in a DAG. The DAG on data level allows for fine-granular composition of data sources and is the basis of the **dataflow graph**. Only

generic data elements – which do not receive CSS properties and have no visible representation in the scene – are organized in a DAG. Hence, we do not run into issues with CSS inheritance and object identification.

6. For scene management it is also possible to **encapsulate subscenes in assets**. In contrast to subscene instancing, the instances of such assets are configurable from the main description. Again, we avoid conflicts with HTML and DOM technologies, because the <model> element in the DOM tree that references a configurable asset is a leaf of the SceneElement tree and therefor the entity visible for an application at DOM level.

7. We use a **overriding concept for specialization** for generic data, assets, and materials, where specific data entries override more general (or default) entries. This allows authors designing these objects in a *Convention over Configuration* [Mil09] style, a design philosophy and technique with a strategy of defaults over explicit configuration. The idea is to gain simplicity by making modules work without any configuration without losing flexibility. This pattern, originally described for software packages, also applies to 3D models.

In summary, we found a way to design the abstract model of XML3D in a way that it is still compatible with web technologies such as HTML DOM, DOM Events, and CSS. Nonetheless, we offer two powerful instancing techniques that allows for fine-granular scene management superior to simple subscene instancing and suitable for recent graphics APIs. Finally, we have a generalized and consistent approach to reference external resources via URIs.

Beyond the Web Although explicitly designed for the web, the concepts of our architecture contribute to graphics system beyond the web:

1. The lean abstract model reflects the basic building blocks for a 3D scene description. All other aspects can be considered as result of data processing which can be either handle by the data processing facilities or by scripting. This concept has been further elaborated by Lemme et al. in [LSSS16].

2. The portable and adaptive programmable material descriptions solve the material dilemma that prevents materials to be exchanged between applications. Hence it could also be used as materials description for 3D data formats.

3. Our concept of discarding monolithic high-level abstractions in favor of composing functionality from building blocks based on a general data processing graph that can be mapped to parallel architectures could be beneficial also for native environments.

4. Efficient schemaless delivery of compressed data is useful also for web applications that need to transfer other content than just 3D, but also as a general container format for binary data beyond the web.

5. On top, the concepts of the XML3D architecture combine the advantages of previous 3D application models. We will discuss this bridging in detail in Section 10.2.

In the next chapter we will discuss in more detail some important characteristics of XML3D, its implementations, and how to extend it.

```
<!doctype html>
<html lang="en">
<head>
  <title>The XML3D Architecture</title>
  <script type="text/javascript" src="../../script/xml3d.js"></script>
</head>
<body>
<div id="content">
  <!-- Start of the XML3D scene defining the size of the projection area
  and referencing a view element -->
  <xml3d view="#defaultView" style="width: 600px; height: 400px;">
    <defs>
      <!-- Definition of a time-varying material model in shade.js -->
      <script id="customMaterial" type="text/javascript">
        function shade(env) {
          var ks = env.specularColor || new Vec(0.8);
          var tx = env.texcoord.mul(200.0);
          var sweep = Math.sin(env.time * .25) * tx.x() * 0.005;
          var px = tx.x() + tx.y() * (1.75 * sweep + 1.0);
          var modAmount = (px / 48) % 3;

          var tints0 = new Vec3(1, .5, .2);
          if (env.tints0 && env.tints0.sample2D) {
            tints0 = env.tints0.sample2D(env.texcoord).rgb()
          }
          var tints1 = env.tint1 || new Vec3(.3, .8, .4);
          var tints2 = env.tint2 || new Vec3(.3, .6, 1);
          var tint = modAmount > 2 ? tints2 : (modAmount > 1) ? tints1 : tints0;
          return new Shade().diffuse(tint.mul(Math.fract(modAmount)), env.normal)
                            .ward(ks, env.normal, env.tangent.xyz(), 0.4, 0.1);
        }
      </script>

      <!-- Definition of a material, i.e. material parameters and a material model -->
      <material id="myMaterial" model="#customMaterial">
        <float3 name="specularColor">0.5 0.5 0.5</float3>
        <float name="time">0.5</float>
        <texture name="tints0" wrap="repeat clamp">
          <!-- HTML <img> element for the definition of the image source of a texture -->
          <img src="../textures/wood.jpg"/>
        </texture>
      </material>

      <!-- <data> element with attached Xflow operator that computes tangents based on
             the input parameters -->
      <data id="mesh_data"
            compute="tangent = xflow.calculateTangents(index, position, normal, texcoord)">
        <!-- Reference to an external binary mesh in Blast format -->
        <data src="../../resources/meshes/suzanne.blst"></data>
      </data>

    </defs>

    <!-- Assign a material to this <group> using CSS -->
    <group style="material: url(#myMaterial);">
      <!-- Clicking on this mesh displays an alert window with the given text -->
      <mesh type="triangles" onclick="alert('Hi, my name is Suzanne!');">
        <data src="#mesh_data"></data>
        <float3 name="tints1">0.8 0.0 0.2</float3>
      </mesh>

      <!-- Referencing a whole subscene using the <model> element -->
      <model src="../../resources/assets/robots/ciccio.xml#asset">
        <assetdata name="animation">
          <!-- We override the key parameter of the animation. This allows controlling
```

```
            the animation defined in the asset from the DOM API -->
        <float name="key">0.5</float>
      </assetdata>
    </model>
  </group>

  <!-- Definition of a light using the predefined spot light model
       and positioning of the light using CSS -->
  <light model="urn:xml3d:light:spot"
         style="transform: translate3d(-20px, 40px, 0px) rotateX(-90deg) rotateY(-30deg);">
    <float3 name="intensity">1 1 1</float3>
    <float3 name="attenuation">1 0 0.0008</float3>
  </light>

  <view id="defaultView"></view>

</xml3d>
<script type="text/javascript">
  // Keyframe animation with jQuery
  $("xml3d").on("framedrawn", function (e) {
    var key = computeKeyframe(e);
    $('*[name=key]').text(key);
  });
</script>
</div>
</body>
</html>
```

Figure 5.28: Example XML3D scene from [SKSS15], using the seven core concepts of the XML3D architecture. Details are explained in comments.

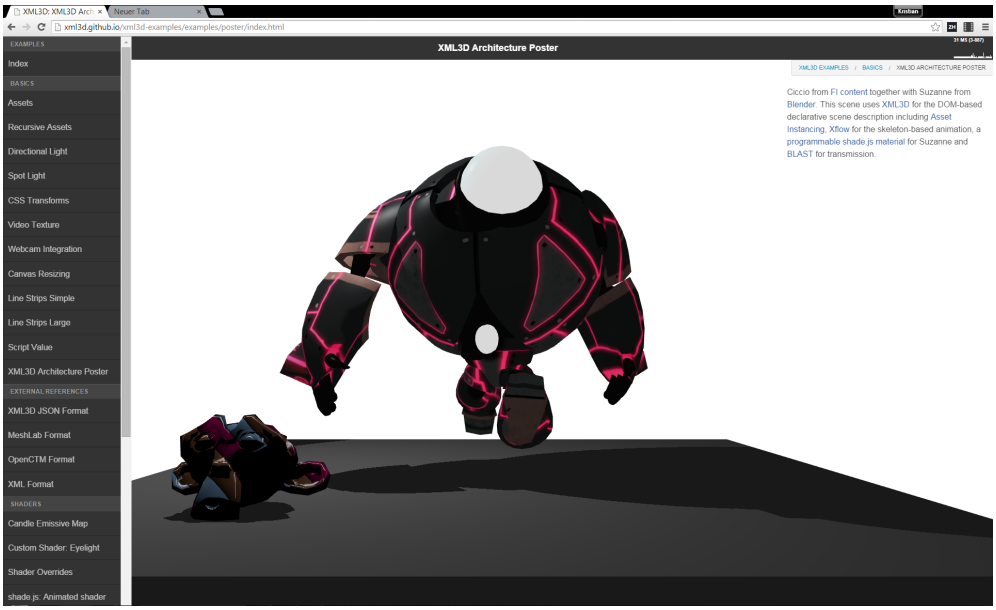


Figure 5.29: Screenshot of the scene described in Figure 5.28. Online at: `http://xml3d.github.io/xml3d-examples/examples/poster/index.html`, last accessed 20 August 2016.

6 XML3D

In this chapter, we will discuss some selected unique aspects of XML3D in more detail. In the following section, we will discuss the evaluation of geometry and lights. Then we will discuss two implementations of XML3D (Section 6.2), the native implementations using ray tracing or OpenGL 4.0 for rendering and our Polyfill implementation based on WebGL and JavaScript . Finally, we will discuss strategies for extending the functionality of XML3D based on three examples in Section 6.3.

We will refrain from a detailed discussion of the abstract model of XML3D As is the case with other web technologies, we consider it a living format that will be continuously refined based on users' and applications' needs. A detailed specification of the latest version of XML3D can be accessed on the project's website.[1] We will show in Chapter 10 that our lean abstract model is sufficient for a large range of use cases.

6.1 Evaluation of the Scene

In this section, we will discuss how XML3D geometry and lights are evaluated. We focus on both of these entities, because both differ significantly from the respective evaluations in previous approaches.

6.1.1 Geometry and Materials

Currently, the abstract model provides a single element for geometry: The <mesh> element describes geometry comprised of a set of primitives supported by recent graphics APIs, i.e. triangles, points, lines, etc. Depending on the primitives, the mesh has a set of parameters with a predefined semantic. Some of these parameters are mandatory, others are optional. For instance, the *position* is the only mandatory parameter of a mesh of type *triangles*. The parameters *index*, *vertexCount*, and *boundingBox* are optional parameters.

The first step during evaluation of the <mesh> is the computation of the number of required vertices. It is used to determine the least number of entries a parameter is required to have to qualify as a per-vertex attribute. The number of required vertices for a triangles mesh depends on its *position*, *index*, and *vertexCount* parameters. If no other parameter is given, the number of *float3* entries in the *position* parameter determines the number of required vertices of the mesh. If an index is given, the highest entry within the index field determines the number of vertices. Finally, using the *vertexCount* parameter, the user can explicitly define the number of required vertices. This parameter can be used to restrict the number of used vertices but also to emit vertices. The latter is useful if the position parameter should not be evaluated on the CPU, but instead be computed on the GPU.[2]

[1] `http://xml3d.org/xml3d/specification/latest/`, accessed 18 December 2015.

[2] For instance the SIG algorithm computes the positions from texture data fetched in the vertex shader (see Figure 5.19). Here, the vertexCount parameter is necessary to emit the vertices.

```
requiredVertices = exists(vertexCount) ? vertexCount :
                   exists(index)       ? max(index)  :
                   exists(position)    ? count(position) : throw Error()
```

Listing 1: Pseudo code for the evaluation of the required vertices.

Listing 1 depicts the computation of the number of required vertices in pseudo code notation. Note that the number of required vertices is not necessarily the number of actual used vertices in the case an index is given. If the number of required vertices cannot be determined or if the dataflow graph will not provide (enough) positions, the rendering system will report an error.

The computation of the number of required vertices need to be reevaluated if the values of *vertexCount* or *index* change during runtime. Additionally, reevaluation is required if the *position* buffer changes its size. Since Xflow's event system differentiates between value changes and size changes of parameters it is possible to change the values of the *position* buffer without reevaluating the number of required vertices.

The *position* and the optional *index* parameters are also used to compute an axis-aligned bounding box of the geometry. This bounding box is used for visibility culling but can also be accessed using the DOM interface of the <mesh> element. If the positions are updated steadily, for instance as result of a skeleton-based interpolation, it is computational expensive to reevaluate the bounding box for each frame. In cases where the actual positions are only computed on the GPU, it would require compute shaders and an expensive read-back from GPU memory to receive the bounding box. Hence, it is possible to annotate a (conservative) approximation of the bounding box using the *boundingBox* parameter, in which case no reevaluations are performed.

The second step of the evaluation is the determination of material parameters. As discussed in the previous chapter, XML3D has no predefined material parameters. Instead, only the attached material model defines the parameters required to evaluate the material. A list of (possibly) used material parameters are annotated for predefined materials and computed by the compiler for shade.js materials (see Section 7.4 for details). Based on this list of material parameters, the rendering system needs to evaluate the existence, the frequency (uniform or per-vertex) and the type for each parameter.

In general, there are three sources for material parameters: Uniform parameters and textures can be defined in the <material> element or alternatively as parameters of the <mesh> element itself. If a <mesh> parameters qualifies as vertex attribute (based on the determined required number of vertices as computed above), it is used as (the more specific) vertex attribute, otherwise as a uniform parameter.[3] Similar, the more specific <mesh> parameters override the more general parameters defined in the <material> element. This behavior fits very well to the general overriding concept for specialization that also applies to data and asset definitions.

Then the Xflow graph is queried for any of the possible material parameters and returns information on the existence, the frequency and the type of each parameter. The existence

[3]Since the number of vertices is typically larger than the number of entries in an uniform array there are very rare cases where a parameter meant as uniform attribute is used as a per-vertex attribute. Additionally, Xflow provides means to annotate operator inputs to be used as uniform parameters even if they would qualify as per-vertex attribute.

and the type of the parameters is static for a valid dataflow graph and can only alter if the graph changes its structure. The frequency, i.e. if the parameter qualifies as vertex attribute, depends on its size and, hence, can only change if a buffer in the graph changes its size. Structural changes, i.e. changes in existence, frequency, or the type of a material parameter, require a re-specialization of the corresponding shader. This specialization may result in a new variation of the shader and consequential compilation in GPU-based implementations.[4]

Note, that the evaluation direction is reversed compared to classical scene graphs with predefined semantics for material parameters. In classical systems, the predefined attributes of geometry objects get evaluated and then the material (and the corresponding shader) are specialized based on the results of this evaluation. Our approach has the advantage, that parts of the dataflow graph are only evaluated if actually needed. For instance, a tangent operator which computes mesh tangents based on its normals and uv-coordinates gets only triggered if the attached material requests the "tangent" field. Or, if a material is attached that colors the mesh based on temperatures, the query for the "temperature" field could lead to an additional request to a server in order to download this information. This fine-granular evaluation based on data-dependencies is only possible due to the deep integration of the dataflow graph concept. In particular, the evaluation logic of XML3D takes into account that the starting point for data queries is the demand of the shading process and not the other way around.

6.1.2 Lights

In offline renderers there is a trend towards image-based lighting (from HDR environment maps) and lighting from light-emitting geometry. This trend is approaching realtime graphics as well. We are convinced that these techniques will be the major two lighting approaches in the future. However, simple light models are still ubiquitous in DCC tools, easy to understand, and inexpensive if used for direct illumination only.

For XML3D, we require an abstract model that makes it *a*) easy for authors and content tools to specify simple light sources based on common light models, *b*) easy to identify light sources in the scene, and *c*) possible to define more complex lighting including use of light-emitting geometry and programmable light models in the future.

Hence, we developed an approach which unifies the common lighting approaches in a single model. XML3D has a <light> element, which defines the occurrence of a light source in the scene. The *model* attribute specifies the light model to be used. XML3D supports a set of predefined light models that are identified via URN (e.g. *urn:xml3d:light:point*).[5] Each light model has a set of supported attributes, which are defined using the generic data model. With useful default values for the attributes, defining simple light sources is easy and concise. This example defines a directional light with default parameters:

```
<light></light>
```

This definition is equivalent to:

[4]The specialization of shaders required on contextual changes will be discussed in detail in Chapter 7.5

[5]For a list of predefined light models refer to the XML3D specification: `http://xml3d.org/xml3d/specification/5.0/`, accessed 06 January 2016.

```
<light style="light-model: url(urn:xml3d:light:directional)">
  <!-- The direction of the light in object space -->
  <float3 name="direction">0 0 -1</float3>
  <!-- The RGB intensity of the light -->
  <float3 name="intensity">1 1 1</float3>
</light>
```

A light's position and direction in object space is defined by the corresponding parameters with the same name (if applicable for the chosen light model). In the example above, one can adapt the default direction of the directional light by defining the *direction* attribute. The final global pose of the light is determined by applying the transformation defined by the position of the <light> element in the transformation hierarchy to the pose in object space.

Note that we can use the functionality of the generic attribute system for composing and sharing light configurations. Additionally, light configurations may be defined in external resources. Finally, light attributes can be computed and animated using Xflow.

A particularity of XML3D is that the <light> element not only allows for the definition of a single light instance, but also for arrays of lights. Therefore we introduce a light iterator concept, similar to vertices for <mesh> primitives: Each light model has a specific required attribute that determines the number of light instances that the <light> element emits. Based on this number, all other attributes either qualify as uniform or as per-light instance attribute: If the number is equal or greater than the iterator size, the attribute is considered as a per-light attribute, otherwise it is considered to be uniform for all light instances.

In this example, we define three point lights with three different intensities:

```
<light style="light-model: url(urn:xml3d:light:point)">
  <float3 name="position">1 1 1 -1 -1 -1 0 0 0</float3>
  <!-- Intensity qualifies as a per light instance attribute -->
  <float3 name="intensity">0 1 0 0.8 0.8 0.8 0 1 0</float3>
</light>
```

This example defines the same three point lights with a uniform constant attenuation for all three instances:

```
<light style="light-model: url(urn:xml3d:light:point)">
  <float3 name="position">1 1 1 -1 -1 -1 0 0 0</float3>
  <float3 name="intensity">0 1 0 0.8 0.8 0.8 0 1 0</float3>
  <!-- Attenuation qualifies as a uniform light parameter -->
  <float name="attenuation">1 0 0</float>
</light>
```

The light is intentionally designed to be very similar to the <mesh> element. In fact, both models can share data. This example defines a triangle with point lights for each vertex:

```
<data id="d1">
  <float3 name="position">1 1 1 -1 -1 -1 0 0 0</float3>
  <float3 name="intensity">0 1 0 0.8 0.8 0.8 0 1 0</float3>
</data>

<mesh type="triangles" src="#d1"/>
<light style="light-model: url(urn:xml3d:light:point);" src="#d1"/>
```

On the conceptual level, one can consider the <mesh> as an element that emits primitives and the <light> as an element that emits light sources. It would even possible to unify both models:

```
<!-- A mesh emitting lights and primitives -->
<mesh style="light-model: url(urn:...:point); primitives: triangles;" src="#d1"/>

<!--
     Next both definitions are equivalent:
     a) A mesh emitting only lights
     b) A light does not emit primitives by definition
 -->
<mesh style="light-model: url(urn:xml3d:light:point); primitives: none;" src="#d1"/>
<light style="light-model: url(urn:xml3d:light:point);" src="#d1"/>
```

Although this unification is interesting in theory, we consider it confusing in practical settings. In particular, it contradicts our requirement that users should be able to easily identify light sources in the scene. Hence, we do not support this concept but require explicit elements for lights and geometry.

Summary The abstract model of XML3D allows for defining simple light sources in a simple and concise way. For most use cases, the definition of the desired light model and a subset of parameters is sufficient. The user can easily identify light source in the scene by just querying all <light> elements. On the other hand, we allow the definition of arrays of lights that can share attributes with a mesh and therefor allow definition of light-emitting geometry but also other many-light techniques. Hence, our model for lights meets the requirements formulated earlier. In terms of our design criteria, it fits well to the guideline of making simple things easy and complex things still possible.

The definition of light models for image-based lighting and programmable light models is work in progress (see Section 11.2).

6.2 XML3D Implementations

We developed three implementations of XML3D in order to evaluate our concepts. The first two implementations are modifications of Mozilla's Firefox[6] and of Chromium[7], respectively (Section 6.2.2). Since these implementations extend the native browser in order to support XML3D, we call them *native* implementations in the remainder of this thesis.

The third implementation, xml3d.js, emulates the native XML3D functionality but is actually developed as a JavaScript library that monitors the XML3D DOM and renders the result to a canvas using WebGL. Web libraries that emulate functionality not natively available in the browser using JavaScript and third-party APIs are often called a "Polyfill". Hence we call xml3d.js also the *Polyfill* implementation.

A schematic overview of our implementations can be seen in Figure 6.1. In the next section, we will discuss the Polyfill implementation in more detail, followed by a brief discussion of the native implementations.

[6] `https://www.mozilla.org/en-US/firefox/products/`, accessed 18 December 2015.

[7] `https://www.chromium.org/`, accessed 18 December 2015. Chromium is the open-source version of the Google Chrome browser.

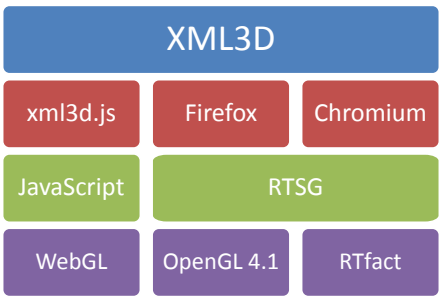


Figure 6.1: Overview of all implementations of XML3D. The native implementations are based on RTSG for scene graph management, RTfact for ray tracing, and OpenGL 4.0 for rasterization. The Polyfill implementation xml3d.js is based on JavaScript and WebGL.

6.2.1 The Polyfill Implementation

The Polyfill implementation of XML3D is called *xml3d.js*. It is a library that emulates the functionality of XML3D using JavaScript and WebGL. Thus, adding the library to a web page enables using XML3D in every WebGL-capable browser. This line of code adds the latest stable and compressed version of xml3d.js to a web page:

```
<script type="application/javascript"
          src="http://xml3d.org/xml3d/scripts/xml3d-min.js"></script>
```

In the following, we will describe the architecture of this XML3D implementation including its limitations. This section of the thesis has been previously published in [SSK*13].

6.2.1.1 Overview

When the browser loads an HTML page, it parses it to the corresponding DOM representation. This representation may contain one or more XML3D scenes enclosed in the <xml3d> element. The new XML3D elements that we introduced are not known to the browser, which creates *HTMLUnknownElement*s that provide generic access to their attributes and child elements.

Obviously, such generic elements do not implement the DOM interfaces that we specified for each of the XML3D elements [SSK16]. Thus, each XML3D element needs to be *configured* depending on its type (Section 6.2.1.4). In addition, the library attaches *Adapters* to the XML3D elements (6.2.1.5). Apart from observing changes in the DOM structure, each adapter implements a specific functionality and belongs to a functional unit we call a *Module* (6.2.1.6). *Data compositing including Xflow* (6.2.1.9) is implemented as a module, the *WebGL Renderer* (6.2.1.9) is implemented in a separate module.[8]

Adapters of different modules can communicate with each other. Instead of accessing other adapters directly, one can request an adapter with a specific interface from the central *resource manager* (6.2.1.7). This mechanism provides us a way to allow adapters not only for DOM elements, but also for other resources that can be identified by a URI. At the same time, the source of the adapter remains transparent to other modules. This concept

[8] In Section 6.3.1 we will discuss the XML3D Physics extension, which was implemented as another module.

gives us also the means to easily *extend* the XML3D framework with additional types of resources, e.g. materials and data formats (6.2.1.8).

6.2.1.2 DOM Initialization and Monitoring

The DOM API provides a generic way to access and manipulate the document. Once the DOM has been created a corresponding event is dispatched by the browser. This event is intercepted by xml3d.js, which then initializes its internal data structures for all <xml3d> scenes.

The DOM API does not limit the modification of the DOM structure: Every single element and attribute may be modified, and we must be prepared to react to all these kinds of changes, i.e. we can not consider any parts of the scene described in the DOM as completely static. The *MutationObserver* is part of the DOM specification and provides means to react to changes in the DOM. An observer can be configured to trigger events whenever structural changes due to addition or removal of DOM elements, attribute changes, or changes in text content occur. Using MutationObservers, xml3d.js can monitor all changes in the DOM and modify the internal data structures accordingly.

6.2.1.3 CSS Monitoring

Similar to changes in the DOM, the library has to initialize the style and to react to style changes. The `Window.getComputedStyle(elem)` method gives the evaluated values of all the CSS properties of an element. This way we can initialize the style of the XML3D elements.

Unfortunately, there is no functionality to explicitly monitor style changes similar to the MutationObserver for DOM changes. Instead we have to monitor all DOM changes that may lead to style changes and then reevaluate all styles using the method we use for initialization. For more efficient means, we propose extending the MutationObserver to also support monitoring style changes. In the context of our research we contributed this proposal to the W3C Dec3D Community Group.[9]

Moreover, the browser discards all unknown CSS properties and values. Hence it is not possible to use new CSS properties introduced by XML3D. This regards mainly the *material* property. Instead, materials can be attached to groups and meshes using the *material* attribute as a fallback:

```
<!-- Assigning a material to a group using CSS -->
<group style="material: url(../material-library#gold)">...</group>

<!-- For xml3d.js we have to use an attribute instead -->
<group material="../material-library#gold">...</group>
```

Unfortunately we loose the powerful mechanisms of assigning materials using CSS selectors. However, all standard CSS properties that influence XML3D elements (e.g. for transformations and visibility) can be applied without restrictions.

[9] `http://xml3d.org/xml3d/specification/styleobserver/`, accessed 18 December 2015.

6.2.1.4 Configuration

When parsing the DOM, the browser creates generic HTMLUnknownElements for all XML3D elements. Obviously, these elements do not implement the interfaces we specified. Thus, each XML3D element needs to be configured depending on its type.

The configuration of methods is straight-forward. All DOM elements are exposed as JavaScript objects. Due to the dynamic nature of JavaScript it is possible to attach functions to the elements, which make them behave as methods of the object.

The configuration of attributes requires synchronization between two representations: Each attribute can be accessed and modified using the generic DOM methods `getAttribute(name)` and `setAttribute(name,value)` that work merely with strings. Additionally, there is a typed access to the attribute values. These are exposed as element-specific properties of the DOM API. For instance, we have two ways to set the *translation* attribute of a <transform> element:

```
var t = document.getElementsByTagName("transform")[0];
// generic attribute modification
t.setAttribute("translation", "0 -1 0");
// in sync with access via property
console.log(t.translation); // XML3DVec3(0, -1, 0)

// typed property modification
t.translation.x = 5;
// in sync with access via attribute
console.log(t.getAttribute("translation")); // "5 -1 0"
```

We use JavaScript's `Object.defineProperty` to add additional properties to the element's interface during runtime. Each property has a getter and setter method. This way, we can synchronize the stringified DOM attribute value with the typed property version. Another approach would be to compute the typed version on-the-fly from the string representation. This is a trade-off between runtime and memory performance.

Additionally, DOM elements can be created at any point during runtime using the `document.createElement(name)` method. The xml3d.js library "captures" this method and configures new XML3D elements during their creation.

Canvas The configuration of the <xml3d> element is special because it defines the transition from the HTML layout model to the XML3D three dimensional coordinate space. The <xml3d> element has meaning for both models: In the HTML layout model it defines the drawable area within the page where the 3D scene will be rendered to. It behaves just like any other CSS-styleable HTML element. It also has the attributes *width* and *height*, because the size of the drawing area is not derivable from its children. On the other hand, the <xml3d> element defines the root coordinate system of the XML3D scene.

To implement this behavior, the configuration module adds a <canvas> element to the DOM at the actual position of the <xml3d> element. The <xml3d> element becomes a sibling of the canvas element.[10] All accesses to HTML-specific interface methods and

[10]This behavior is not optimal as it changes the DOM implicitly. As soon as the ShadowDOM [W3C15a] functionality is widely available in browsers, the <canvas> element can be hidden inside the <xml3d> element's shadow tree.

properties of the <xml3d> element are delegated to the canvas object using the JavaScript capability to attach methods and to define getter and setters for properties. This includes the attributes *class*, *style*, *width*, *height*, and several methods (e.g. `getBoundingClientRect`).

6.2.1.5 Adapters

Contrary to the common scene graph approach seen in other scene graph libraries (e.g. Open Inventor [SC92], Performer [RH94], OpenSG [Rei02]) we do not implement functionality such as culling, rendering, and bounding box computation in dedicated scene graph traversal actions.

Instead, our implementation provides a different abstraction level: it does not define the way how *to implement* a functionality (e.g. using a scene graph traversal), but only the way how *to request* it. Each element (or other resource) provides a set of services (e.g. calculation of a bounding box) that can be requested. Services are represented by their interface and requested by wrapping a DOM element or resource to provide that interface. Such a service is implemented in objects that we call *Adapters*. Adapters implement a wrapper pattern. *ElementAdapters* are specialized adapters. Besides implementing services, ElementAdapters observe their corresponding DOM element and react to changes. Thus these adapters implement the observer pattern as well as the wrapper pattern.

Because adapters can contain a persistent state (e.g. cached images, WebGL handles, etc.) their lifetime is connected to the lifetime of their corresponding resource. For instance, when a element is removed from the scene graph we dispose all of its ElementAdapters.

This modular concept, previously implemented in the RTSG2 [RGSS09] scene graph library, allows us to implement complex operations that require data exchange between elements more easily. For instance, an implementation of the <mesh> element might require data provided by a <data> element referenced in its `src` attribute. A traversal algorithm would have to store the data of the <data> element in its state and retrieve them when the <mesh> element is visited. We can achieve the same by having the mesh adapter requesting the data service of the *data* adapter of the <data> element.

6.2.1.6 Modules

It is possible to attach multiple adapters to one DOM element. This way we can separate the different concerns necessary to run the system (e.g. rendering and data access, see Figure 6.2). Each adapter belongs to a module that implements the concern. For example, all elements in the scene hierarchy provide a `getLocalBoundingBox` method that returns the bounding box in the local coordinate system. For such a case, the implementation of the methods queries all attached adapters for the desired methods and finally returns the result of the first adapter that provides the functionality.

Some modules are bound to a specific resource. For instance, the rendering adapter has data structures that belong to a specific *WebGLRenderingContext*. Such a WebGLRenderingContext is used to manage OpenGL state and render to the drawing buffer and cannot be shared with others. In these cases we do not have just one adapter per concern but also per context. This is necessary because XML3D resources can be referenced from multiple scenes. For instance, two meshes from two different scenes may share their vertex data. However, the corresponding WebGL resources, in this case WebGL buffers for vertex

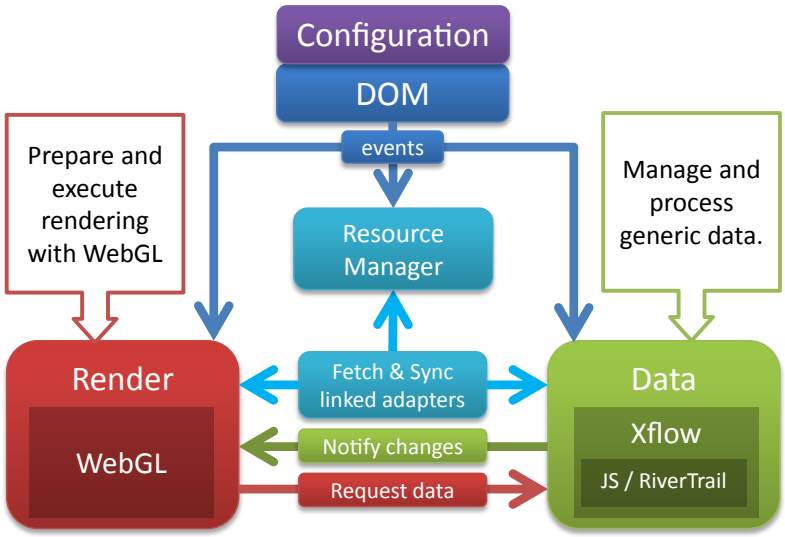


Figure 6.2: Separation of concerns in xml3d.js. We have separate modules to manage rendering and generic data access/processing, both connected to the DOM via adapters.

attributes (VBOs), cannot be shared between two WebGLRenderingContexts. Hence, we create an adapter per context, each managing a context-specific set of VBOs from the same data source.

6.2.1.7 Resource Handling

An XML3D scene may reference different kinds of internal and external resources via URI. This includes media objects, such as images and videos used as surface texture, external mesh and material data in various formats, and predefined shaders (via URN).

External resources may be encoded in various formats and the number of supported formats can be extended using plug-ins (see next section). For instance, an external resource might be XML3D content (e.g. data or a material) within another HTML document. Alternatively, the same content might be encoded in JavaScript Object Notation (JSON), for which parsing is slightly faster than for XML. Another option is to encode the content in Blast, our binary container format (see Chapter 8).

We have to handle the asynchronous loading of these external resources. An external resource is not available until it has been transmitted and parsed. Consequently, we need to update the state of the internal scene representation whenever an external resource becomes available. In order to simplify this process, we have a per-document entity that is in charge of handling all external resources: the *ResourceManager*.

The ResourceManager keeps track of all requested resources and caches received files to make sure each resource is loaded only once. It also monitors the loading state of each requested resource and implements a callback mechanism to notify other modules about completed resource transfers. This design allows for implementing an XML3D specific *onload* event triggered when a specific resource and when all resources of the XML3D document have been loaded.

The ResourceManager only exists once per document and is used by all XML3D scenes

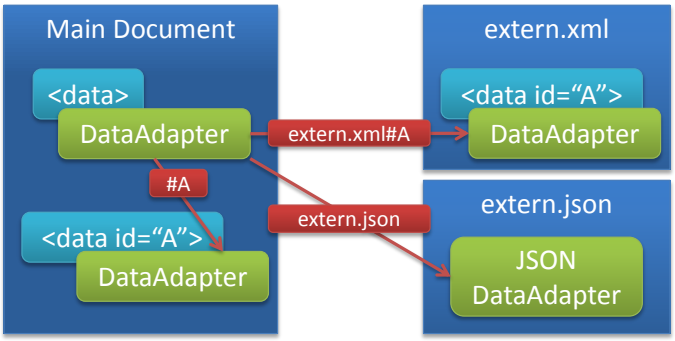


Figure 6.3: Internal elements and external resources are all wrapped by DataAdaters providing the same interface. Consequently, the source and format of these resources remains transparent for other modules and adding a data format only requires implementing a new data adapter.

inside that document. This allows us to share resources among multiple XML3D scenes to save memory.

The handling of external resources poses a design challenge, since all URI references may point to DOM elements in the same document or to external files. Because external files can be of any format and are not required to be DOM-based, we cannot simply resolve URIs to DOM elements or their adapters.

To avoid handling two different kinds of references, we resolve all URI references to direct adapter-to-adapter connections. For each scene graph feature we may provide multiple, format-specific implementations of the same adapter type. This allows us to implement adapters which do not depend on the DOM and have direct access to the data of an external file (see Figure 6.3).

With this design, the ResourceManager becomes the central point for resolving all URIs. An adapter can be requested from the ResourceManager by providing a URL, the type of the requested adapter, and a canvas context. The ResourceManager reuses cached adapters if an adapter for the same resource, type, and context has already been requested previously. Otherwise it creates a new adapter. Instead of returning an adapter directly, the ResourceManager returns an *AdapterHandle*. The handle is returned synchronously even though the adapter might not yet be available due to the asynchronous loading of resources.

AdapterHandles provide access to the adapter – if available – and a callback mechanism in case the state of the adapter connection changed. Changes in the adapter connections do not only occur due to loading files, but also when the local document has been changed (e.g. DOM element additions and removals, modification of identifiers, and so on). In addition to loading files, the ResourceManager tracks these document changes to make sure that URIs are correctly resolved or set to dangling if their target has been removed.

6.2.1.8 Extensibility

The xml3d.js framework provides three ways for extending its functionality: Adding new modules, registering new adapters that can handle a specific external format and registering new Xflow operators. We discuss how to extend XML3D in order to allow running physics simulations on XML3D scenes in Section 6.3.1. To implement such a physics

simulation, we added another module to xml3d.js that implements the aspects of that simulation. The existing infrastructure for initialization, modifications, and data handling can be reused if an aspect is added via a module.

Users can extend the set of supported external formats for data, materials, etc. Therefore we provide a plug-in mechanism that allows registering new factories for an adapter type and a specific *media type* [FB96]. The adapter type describes the provided interfaces (e.g. providing data, materials, etc). The media type is a general identifier for the format of a resource (e.g. `application/xml`) and is provided in the header information of an HTTP response.

The media type alone provides not enough information to identify the plug-in for a specific data format. For instance, there could be multiple JSON-encoded data formats from different exporters all delivered with the generic media type `application/json`. Each of these formats may provide data, but require different plug-ins for the structure to be translated to a DataMap representation (see Section 5.4). For these cases, i.e. if multiple factories register for the same media and adapter type, each candidate factory is queried if it supports the content of a given resource.

The plug-in mechanism for external formats is used for all natively supported data formats. Additional plug-ins exist for MeshLabs JSON, STL, and OpenCTM.

6.2.1.9 Module implementations

In this section, we describe the two main modules of xml3d.js: The module for data compositing and processing and the module for rendering and picking.

Data compositing and processing The implementation of the generic data model and dataflow processing can be separated from other aspects. Hence, we implement this functionality in a dedicated module. The *DataModule* provides *DataAdapters*, which manage the data composition including processing and provide an interface to access data results.

Xflow itself is implemented in a separate software library that can be used independently of xml3d.js. This library keeps its own optimized structure for dataflows with an API for constructing and updating such dataflows. The DataModule propagates any changes in the document relevant for the data graph to the Xflow library to keep the internal structure synchronized. The library provides an interface to request data entries from nodes in the dataflow graph. Each time, requested data entries change as a result of a change in the dataflow graph, all requester get notified via a callback. In the case of xml3d.js, the data adapters hold references to their corresponding nodes in the dataflow graph and render adapters request data relevant for the rendering process.

Rendering The render module is responsible for the creation and administration of WebGL data structures and the actual rendering process including hardware accelerated picking. During initialization, the library traverses the DOM tree and attaches rendering-specific adapters. <mesh> elements are leaf elements for the rendering related part of the DOM. Thus, each mesh adapter creates a *RenderObject* that collects all information required for rendering of the mesh, namely geometry data, material information and transformations. Similar, the adapter of a <model> element constructs a list of RenderObjects.

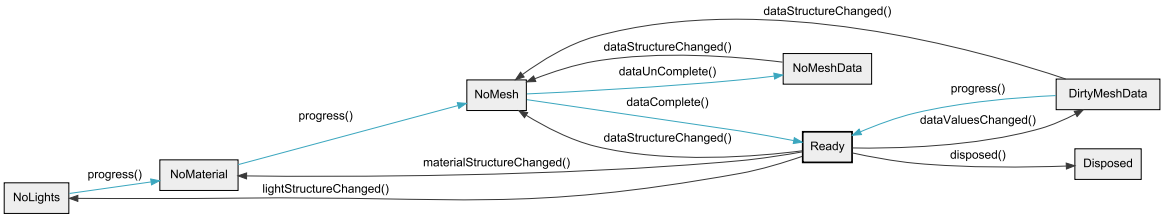


Figure 6.4: The state machine for internal render objects. Transitions in blue are triggered at any time by events, whereas black transitions are triggered in a batch during the pre-render phase. Some obvious transitions are omitted for readability.

RenderObjects register to the *RenderObjectHandler*, that holds two lists: One for RenderObjects that are not yet complete, for instance because an external resource is not yet available and another list for those RenderObjects ready for rendering.

A good scheduling strategy is essential for the overall performance of the system as well as for a good user experience. This is even more important than in classical rendering frameworks for two reasons:

Asynchrony With all referenced resources being loaded asynchronously, it is not guaranteed if and when the loading process will complete. When the data eventually arrives, it is not certain if it contains the required set of information. This holds for images, data for meshes, asset, materials, lights, and prototyped Xflow graphs.

Execution phases Events are triggered in different execution phases. For instance, UI events (mouse, keyboard, touch), mutation events, timer events, and loading events (XMLHttpRequest) all have dedicated executions phases.

We try to limit the computations and WebGL calls within the callbacks of these events to not block the event phase. It is of particular importance to not block the UI thread to guarantee a smooth user experience.

Most browsers perform the actual 3D rendering in a dedicated thread for security reasons and the web runtime communicates with this thread using remote procedure calls. Calling a WebGL-API functions comes with an administrative overhead for communication with the rendering thread. Thus we try to batch WebGL calls wherever possible.

We added a finite-state machine (FSM) to the RenderObjects that gives us granular control over scheduling. In the FSM (see Figure 6.4) we have seven states:

1. Each RenderObject starts in the *NoLights* state. The RenderObject is created in the initial DOM traversal phase, but the final number and kinds of lights is available only at the end of the traversal. If the number or kind of lights change during runtime, a *lightStructureChanged()* event is triggered that sets all RenderObjects back to this initial state.

2. When all light information are available, the system triggers a transition of all objects in the NoLights state to the *NoMaterial* state. If the parameters of a material were changed in their structure (e.g. a texture was added or removed), all RenderObjects that have that material attached go back to the NoMaterial state.

3. The RenderObjectHandler tries to push the RenderObject to the *NoMesh* state in the pre-rendering step. This transition is only successful, if all material information is available and the corresponding shader program compiles successfully. If the data for a mesh has structural changes, i.e. a parameter was removed or added, a *dataStructureChanged()* event is triggered, which puts the RenderObject back to the NoMesh state from any state. This is necessary, because such a change may lead to an incomplete mesh.

4. The RenderObjectHandler tries to push the RenderObject from the NoMesh state to the *Ready* state. To perform this transition, the Xflow graph needs to be evaluated (see Section 6.1.1) and it is only clear after the evaluation, whether all required data is available.

5. If the mesh data is incomplete, the transition to the Ready state is canceled and the RenderObject transits to the *NoMeshData* state. A mesh can only leave this invalid state, if there is another structural change in its data.

6. If Xflow triggers a notification that a value has changed that requires the re-evaluation of the dataflow graph, the system pushes the RenderObject to the *DirtyMeshData* state. Note that the notification may take place in the UI thread and only marks the object dirty (setting it to the DirtyMeshData state). However, the actual evaluation of the dataflow graph happens in the pre-rendering phase, when the RenderObjectHandler pushes back the RenderObject into the Ready state.

7. A RenderObject enters the *Disposal* state, whenever a *dispose()* event gets triggered due to removal of the corresponding <mesh> element from the scene. Entering this state releases all used resources. If the <mesh> elements is added to the scene again, a new RenderObject is created.

Note that transitions triggered by scene changes, e.g. by DOM mutation events do not involve heavy calculations. In contrast, in all transitions that may occur in the pre-rendering phase involve dataflow graph evaluation, WebGL calls and other maybe heavy calculations. These transitions are only executed in a dedicated callback triggered by the browser before the next repaint (`requestAnimationFrame` [W3C13b]). This way we ensure that we do not block the execution phases of UI and loading events and that we batch WebGL calls and execute them in the provided callback.

We can exploit the current view frustum in the pre-rendering phase to push only those RenderObject from the DirtyMesh state back to the Ready state that are within the view frustum and thus only evaluate the dataflow graph for visible objects.[11] Obviously this optimization is only applicable if the geometry has a user-defined bounding box, because otherwise the animation to be evaluated could still move the object into the frustum.

Picking Scene authors can attach event listeners such as *onclick*, *onmouseover*, or *ontouch* to elements such as <group> and <mesh>. For the browser these elements have no obvious

[11] Actually, this is an approximation. Objects outside the view frustum may still influence the lighting of the scene. In particular, they may produce shadows within the view frustum, thus light frustums need to be taken into account as well.

2D spatial representation and thus the browser will not trigger these events on its own.[12] Thus the library has to detect these events in the scene and to trigger these events on the corresponding elements. This means, we need the ability to determine the visible object at a given location on the screen.

All mouse events may occur at a high frequency. For instance, the *mousemove* event is triggered by every change of the pointer position. Hence, detecting the underlying 3D objects needs to be fast and thus we decided for using hardware accelerated picking. WebGL is based on OpenGL ES 2.0, meaning no Selection Buffers nor Multiple Render Targets (MRT) are available. Hence, we render each object's ID into the color target of a (non-visible) FrameBufferObject devoted to picking (*PickingBuffer*). Because the accuracy of user interaction is not pixel accurate, we can safely reduce the resolution of the Picking-Buffer specifically on high-res displays to increase performance. This is particularly useful for devices with low GPU bandwidth.

When a mouse event is received by the <xml3d> element we first determine whether the PickingBuffer must be redrawn due to changes in the scene (e.g. an object has moved, the camera has been rotated, etc). If no such changes were made since the previous pass the data in the PickingBuffer is reused. If a redraw is required we draw each object using a simple color coding shader. This temporary assigned and implementation-defined shader encodes unique object IDs into the RGBA color channels of the PickingBuffer. To retrieve the ID of the object under the mouse pointer we simply read a 1x1 pixel area of the PickingBuffer and decode the resulting color data. This gives us the original object ID, which is used to find the correct RenderObject and thus the correct <mesh> DOM element to dispatch the event on.

We also provide position and normal information about the picked point on demand, both of which require their own render passes over the picked object, again encoding the relevant information into the color channels of the framebuffer. These values are accessed through methods attached to the MouseEvent dispatched to the relevant listeners but are only rendered if explicitly requested by the user invoking these methods.

Picking *normals* is ambiguous if the semantic of the normal is not defined. A normal could refer to the surface normal of the currently active primitive, to the interpolated value (or nearest neighbor) of the per-vertex attribute called "normal", or to the normal used for shading. The latter could differ from the previous definition because it could be altered by the fragment shader, e.g. via a normal map. The current implementation returns the interpolated vertex attribute transformed into the world coordinate system. To avoid any ambiguities and to keep the XML3D architecture's material model consistent (i.e. free of any predefined semantic for material parameters), we plan to add custom attribute picking to the XML3D architecture (see Future Work in Section 11.2).

6.2.1.10 Discussion

Using the Polyfill implementation has the advantage that it runs in every WebGL-capable browser, i.e. on every major browser on PCs and mobile systems. However, it also comes with some limitations.

[12] In fact, the 2D location of XML3D elements is derived from the element enclosing the <xml3d> element. However, by default these unknown elements have a zero extent and hence do not trigger any events.

One major challenge is the integration of CSS. We currently cannot introduce new CSS properties in the Polyfill implementation and monitoring existing CSS properties is more expensive than it should be if appropriate APIs would be available. Similar, the browser's debugging tools can be used to browse and modify the DOM. However, picking an object in the scene will not highlight the corresponding XML3D element in the debugger. An interface to control the debugger would make the emulation of the native functionality even more transparent to the user.

In contrast to the native implementations, the Polyfill implementation is tied to WebGL. With WebGL being a wrapper to OpenGL ES 2.0 (from 1997) it is not possible to exploit the latest features of modern graphics APIs. In particular MRTs, occlusion culling, and geometry shaders are often-missed features that would help to increase the rendering speed and quality. Many of the missing features are foreseen to be added in upcoming WebGL 2.0. Also, it is not yet possible to implement a software renderer (e.g. a ray tracer) with sufficient performance in JavaScript.

The xml3d.js Polyfill implementation is under active development. The source code of the library is publicly available under MIT license.[13] On the practical side, it is used in many research projects, industry projects, and increasingly in third-party projects. Additionally, it is a Generic Enabler in the EU Future Internet project. Finally, the xml3d.js library serves as an evaluation platform for the W3C Community Group *Declarative 3D for the Web Architecture (Dec3D WG)*, which deals with the standardization of declarative 3D approaches and related topics.[14] Based on the feedback that we got from internal and external projects and from the related community we were able to evolve our concepts and to evaluate the architecture based on non-trivial use cases. We will discuss its applicability and performance based on a selection of applications in Chapter 9.

6.2.2 Native Implementations

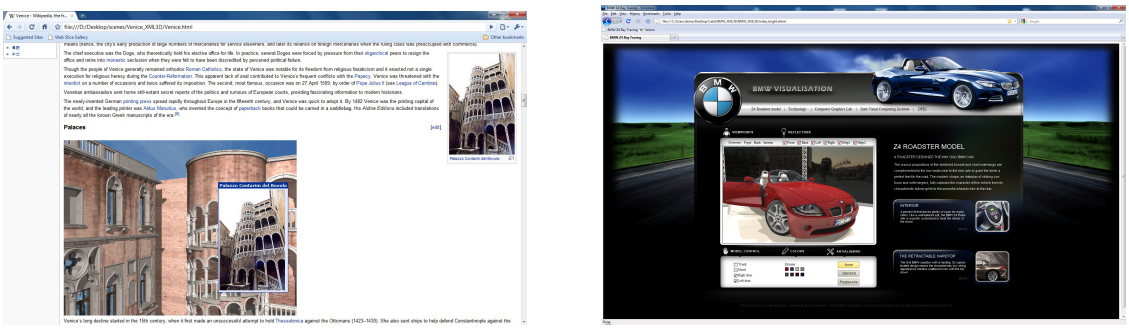


Figure 6.5: The modified version of Chromium browser with XML3D support showing an extended Wikipedia page about Venice with realtime navigation and interaction with a 3D model of a famous Venetian palace (left) and a car configurator demonstrating the tight integration of 2D and 3D content within the same web page inside the modified Firefox browser (right). Both examples use realtime ray tracing for rendering.

We have integrated support for early versions of XML3D natively into the Mozilla

[13] `https://github.com/xml3d/xml3d.js/`, accessed 18 December 2015.
[14] `https://www.w3.org/community/declarative3d/`, accessed 18 December 2015.

browser framework (the basis for the Firefox Web browser) and into Chromium (the open-source version of Google Chrome). These implementations target to eventually support all features of XML3D, different rendering algorithms, and large 3D scenes.

The native implementations of XML3D are composed of several components. Each browser is modified to support the new elements as well as their DOM interfaces. In our implementation the XML3D scenes of the DOM is implemented by an extended version of the RTSG [RGSS09] 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 (e.g. support for URI references).

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 we have to convert to and from this representation on-the-fly.

Thus, the RTSG structure can be considered as a DOM representation with typed data support. Similar to the architecture of the Polyfill implementation, RTSG allows attaching adapters that observe the data structure and implement specific aspects. The implementation is also split into data processing and rendering, with the possibility to attach different renderers.

The native implementations have several advantages compared to the Polyfill implementation:

1. The access to other native libraries and APIs is not restricted. Hence, we can, for instance, attach different renderers using **alternative rendering algorithms**. In our case, we attached two different renderers: A renderer using GPU rasterization based on OpenGL 4.1. Compared to WebGL (which is based on OpenGL ES 2.0), several additional features are available that we can exploit to render faster and with better quality (e.g. multiple rendering targets, uniform array buffer, etc.).

   Our second renderer is a ray tracer based on the RTfact library [GS08]. Figure 6.5 depicts XML3D scenes in the native browsers using RTfact. To achieve interactive ray tracing, RTfact uses SIMD packet ray tracing. On top of the RTfact functionality, the renderer uses fine-granular thread-management based on Intel's Threading Building Blocks [Rei07], which supports parallelism on task-level, including work stealing for balancing parallel workload. Despite providing simple task-parallelism with WebWorkers and SIMD.js being proposed to expose SIMD data-parallelism to the web, the access level a well as the performance of JavaScript is (not yet) sufficient to implement an interactive ray tracer.

2. With full access to the **CSS** implementation of the browser, we can easily introduce new CSS properties and monitor CSS changes efficiently. As discussed earlier, this is currently not possible from JavaScript and, hence, new CSS properties are not available in the Polyfill implementation.

3. In the native implementations all **debugging** facilities also work for 3D content. For

instance, clicking on a geometry while the browser's developer tools are open highlights the corresponding DOM element.

4. In the native implementation we have full control on **memory management**, which allows us to explicitly handle all involved resources. This renders it easier to handle large scenes and in particular use cases that require unloading of resources, e.g. tile-based GIS applications. In contrast, xml3d.js depends on the JavaScript engine's garbage collector and on the browser's caching implementation, which may differ drastically between different implementations.

On the downside, the native implementations require users to download a modified version of Chrome or Firefox to use a native XML3D implementation. Moreover, since browser frameworks evolve very quickly, it was virtually impossible to keep track with our XML3D-specific modifications. As a result, we developed the native implementations to prove the renderer-independence of XML3D, but discarded these implementations in favor of xml3d.js later on.

In addition to the two options implementing XML3D presented above, we consider using Node.js[15] to implement XML3D in a non-browser environment. In combination with jsdom[16], a JavaScript implementation of DOM and other HTML-related web technologies including a subset of CSS, we could run xml3d.js in a JavaScript runtime. The Node.js runtime is less constrained than the browser and allows calling native functions from dynamically-linked shared objects written in C or C++. This way we could use, for instance, distributed ray tracing implementations to render an XML3D scene, but still reuse large-parts of the functionality of xml3d.js, including configuration of DOM interfaces and data compositing. Also the implementation of Xflow could be (incrementally) replaced by a native implementation that could exploit data parallel APIs not exposed in browser-environments. This approach would be interesting for a server-based implementation of XML3D.

6.3 Extending XML3D

Extending the base functionality to implement domain-specific features is an inherent feature of 3D toolkit libraries. In turn, extending a format is critical as it requires all implementations to follow-up and some extensions might break backward compatibility. As discussed earlier, this was the reason we developed a more flexible concepts for data processing and materials. These allow creating functionality without extending the abstract model of XML3D.

Obviously, there is functionality that exceeds the capabilities, but also the scope of the abstract model. Hence, we discuss three approaches to extend XML3D on different levels. First, we discuss how to extend the functionality of XML3D with a rigid body physics simulation in the next sub-section. Then we briefly discuss how the approach for the physics simulation can be generalized and hence be used also for other simulations (Section 6.3.2).

[15] Node.js is an extended version of Google Chrome's JavaScript engine designed for developing server-side applications, `https://nodejs.org/`, last accessed 20 January 2016.

[16] `https://github.com/tmpvar/jsdom`, last accessed 20 January 2016.

Finally, we discuss the extension of the abstract model by a <volume> element that allows rendering volumetric data in Section 6.3.3.

6.3.1 XML3D Physics

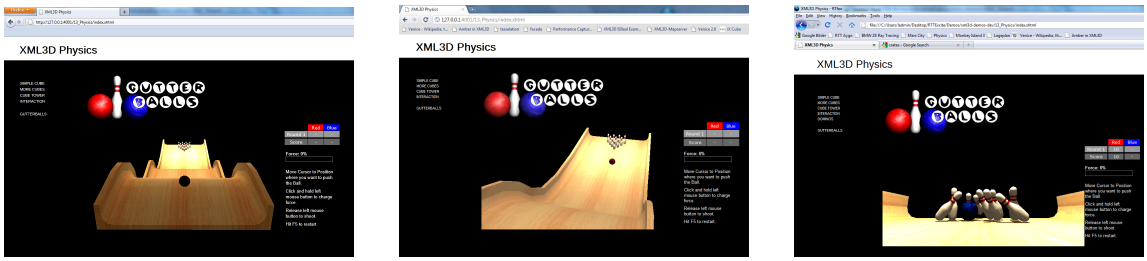


Figure 6.6: A bowling game using XML3D Physics. From left to right: Firefox 5 & WebGL, modified Chromium & OpenGL, modified Firefox & RTfact ray tracer. Invisible geometry objects along the track generate events on collision with the ball, used to control the camera position.

Physics simulations greatly improve the realism of and intuitive interaction with the scene and, hence, are an important component of many interactive 3D applications. In addition, its use can simplify the authoring of 3D scenes. In particular using dynamic animations is often easier than combining predefined animations in a convincing way.

However, in the past physics specifications have been very invasive to the design of 3D scene descriptions, e.g. it required many new nodes distributed across the entire scene graph (see next sub-section). This approach is in clear conflict with the design goals of XML3D and does not scale with the addition of other simulations such as acoustics, haptics, and others. In this section, we present a declarative and orthogonal physics annotation framework that nicely separates the generic 3D scene description from its physics properties. These annotations are ignored by the visual rendering and are instead operated on by a separate physics engine that communicates with the rendering part through modifications of the scene graph (e.g. motion of objects). Through events, callbacks, and an additional API object, physically annotated objects can be directly manipulated and interacted with through JavaScript.

Additionally, we present a plug-in based implementation that runs in all our three XML3D implementations and discuss the implementation based on a number of examples and performance evaluations. Our approach has been previously published in [SS11a].

6.3.1.1 Related work

There are few portable 3D graphics formats that allow describing physical properties in order to drive physics simulations. The COLLADA physics component (since version 1.4) includes the description of physical material and object parameters, proxy geometry for collision detection, constraints, and force fields. This set of functionality set was derived from the functionality of popular rigid body dynamics engines [CV07] and builds also the basis for the functionality of XML3D Physics.

As a main difference, the physics information in COLLADA is kept inside a physics scene separate from the visual scene. The physics scene consists of rigid body instances.

These reference a collection of shapes used as simplified geometry representation, a physics material, and the object that actually represents the rigid body in the visual scene. Having objects living in separate physics and visual scenes reflects the typical system architecture of rendering systems where objects have representations in the physics engine as well as in the rendering engine. However, we think that exposing this implementation-driven approach to the user is not intuitive and too complex: It would require the user to update all the references in the physics scene every time there is a structural change in the visual scene. Also, the inclusion of physics into COLLADA led to a significant extension of the COLLADA format. We wanted to avoid this for XML3D.

X3D 3.2 introduced a *Rigid Body Physics* component [Web08]. In terms of functionality, this component is comparable with COLLADA's physics component. Similar to COLLADA, physics parameters and objects are defined in a dedicated set of X3D nodes. Basis is the newly introduced non-hierarchical <RigidBodyCollection>. <RigidBody> nodes in this collection define the physics parameters of objects participating in the physics simulation. In contrast to COLLADA, the RigidBody objects do not reference their visual representation directly, but a <CollidableShape> node that wraps a single <Shape> node in the scene graph. This invasive design causes, that X3D implementations without support for the Rigid Body Physics component are not even able to display the visual part of the physics-enabled X3D scene.

To detect collisions, the author has to group objects of interest (again the <CollidableShape>s) into a <CollisionCollection>. Placing a <CollisionSensor> in the scene graph that references the <CollisionCollection>, allows for listening to events generated when those objects collide.

The design of X3D's physics component requires a whole range of new nodes, two collections, and references between these collections that must be maintained alongside the actual scene graph. The worst property of this design is, however, that the physics component imposes a specific structure of the scene. We discussed earlier that it is important that the structure of the scene is driven by the application and not by the functionality of the scene graph (cf. Section 4.2).

Although X3D allows a granular configuration on object participation in collision detection and rigid body simulation, the complexity is unreasonably high and prevents non-experts from using it. [SMZB14] proposes an implementation of the X3D Rigid Body Physics component in X3DOM.

Few 3D games can manage without physics simulated by some means or another. There are a number of commercial and non-commercial physics engines available: Havok Physics[17], PhysX[18], Bullet[19], and ODE[20], just to name a few. Additionally, there are frameworks available, that abstract away implementation details of the various physics engines, provide a uniform API, and make implementations comparable [BB07]. Although these API solutions are not directly related to our declarative approach, they give a good overview on the common functionality and parameter sets of recent physics engines.

Most DCC tools offer functionality to rig graphical objects with physics parameters and

[17] `http://www.havok.com/`, accessed 18 December 2015.
[18] `http://physxinfo.com/`, accessed 18 December 2015.
[19] `http://www.bulletphysics.org/`, accessed 18 December 2015.
[20] `http://www.ode.org/`, accessed 18 December 2015.

to run a physics simulation during rendering. Again, this demonstrates how helpful even simple rigid body physics and collision detection can be to support the creation of realistic appearing virtual environments.

6.3.1.2 Design of XML3D Physics

This section explains the requirements for the XML3D Physics extension, gives an overview of the syntax of the physics annotations, and shows possibilities how to interact with the physics objects.

Requirements We had the same requirements on the design of the physics component that we also posed for the XML3D architecture, in particular with respect to functionality (comparable with non-declarative approaches) and usability.

Additionally, we wanted the extension to be orthogonal and robust, i.e. XML3D's abstract model should not be changed because of the physics extension. Also, the interpretation of physics related information should be independent from other simulations such as visual rendering. Adding physics information should not break rendering or other components in case the available runtime environment cannot deal with the physics information.

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

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

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

The following section describes a selection of parameters and how they are connected to corresponding XML3D elements (all references to line numbers refer to Listing 2): The physics parameters need to be defined in a dedicated namespace, identified with a specific URL identifier (Line 3). Typically the namespace is bound to a prefix. This prefix indicates the namespace for an element, its attributes, and all its children (if those are not prefixed otherwise).

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

Material parameters for a physics simulation correspond very much to visual surface parameters for rendering. Thus we decided to introduce another <material> element for physics, that lives in the physics namespace. It leverages XML3D's generic data model to define and compose parameters. Lines 14-17 show the definition of a physics material and its parameter set.

Additionally, the material defines the simulation model of the object within the physics world. There are three kind of physical models in the scene. Objects marked as *static* contribute as collision objects but are neither moved nor modified by the physics simulation. *Kinematic* objects are like static objects but can be moved within the physics world. Forces are applied to all objects that are marked as *dynamic*. Objects that do not have a physics material associated, are not taken into account by the physics simulation at all. Currently supported material parameters are friction, restitution, plus linear and angular damping.

Like visual materials, physics materials can be shared among multiple objects in the scene and are a good candidate for assignment via CSS. Again, the definition of custom CSS properties is not possible in the Polyfill implementation, thus we provide the option to assign physics description via attribute as a fallback (Line 35).

The generic data model can be used as-is to describe physical parameters of geometry objects, e.g. <mesh> elements. In line 28, the *mass* parameter is defined for the mesh and accessible from both, the visual and physical simulation. To prevent clashes of parameter names, it is still possible to define these parameters in a material. Another typical per object parameter is the *centerOfMass*. If not given, a constant density is assumed and the center is derived from the geometry.

The definition of proxy geometries requires structured data, thus we introduced a <shape> element for XML3D Physics (see Line 29). Again, the generic data model is used to define the parameters of the proxy geometry. Similar to the <mesh>, the *type* parameter defines the type of the proxy geometry. If no proxy geometry is given or if the given proxy type is not supported by the physics interpreter, the mesh geometry is used.

A somehow little more complex task is to describe constraints between objects. While the annotation of parameters and assignment of physics materials was straightforward, a constraint defines a relationships between objects that (in most cases) differs from the parent-child relationship within the DOM. If one considers for instance a hinge constraint, the author might not want to structure the scene so that the two parts of the hinge are children of the constraint but rather wants to follow some other application-specific logic. As a result, it is necessary to define the objects of a constraint using URI references. If such

```
<xml3d style="width: 300px; height: 200px;"
       xmlns="http://www.xml3d.org/2009/xml3d"
       xmlns:physics="http://www.xml3d.org/2010/physics"
       physics:gravity="0 -9.81 0">
  <defs>
    <material id="visual-material" model="urn:xml3d:material:phong">
      <float3 name="diffuseColor">1.0 0.9 0.8</float3>
      <float name="ambientIntensity">0.4</float>
      <texture name="diffuseTexture">
          <img src="textures/crate.png"/>
      </texture>
    </material>

    <physics:material id="cube-physics-mat" model="urn:physics:material:dynamic">
      <float name="friction">0.5</float>
      <float name="restitution">0.2</float>
    </physics:material>
    <physics:material id="ground-physics-mat" model="urn:physics:material:static">
      <float name="friction">1.0</float>
      <float name="restitution">1.0</float>
    </physics:material>

    <data id="box">
      <int name="index">0 1 2 2 ...</int>
      <float3 name="position">-1.0 1.0 1.0 ...</float3>
      <float3 name="normal">0.0 0.0 -1.0 ...</float3>
      <float2 name="texcoord">1.0 0.0 1.0 ...</float2>
      <float name="mass">10.0</float>
      <physics:shape type="urn:physics:geometry:box">
          <float3 name="extends">2 2 2</float3>
      </physics:shape>
    </data>
  </defs>
  <group material="#visual-material"
         physics:material="#cube-physics-mat">
      <mesh type="triangles" src="#box"/>
  </group>
  <group material="materials.xml#ground-mat"
         physics:material="#ground-physics-mat"
         physics:oncollision="alert('Object dropped on the floor')">
      <mesh type="triangles" src="./ground.json"/>
  </group>
</xml3d>
```

Listing 2: An XML3D scene containing physics annotations.

a reference cannot be resolved or points to a unsupported element type, the reference is dangling and the constraint is not taken into account by the physics simulation. Listing 3 shows the definition of a constraint and the references to the involved objects. XML3D Physics currently supports following constraint types: *point2point*, *hinge*, *slider*, *conetwist*, *generic6dof*, and *generic6dofspring* [CO15a].

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

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

6.3.1.3 Interaction

Figure 6.7: Scene demonstrating the interaction capabilities of XML3D Physics. It includes pushing of objects (implemented as shown in Listing 4), several proxy geometries, and dynamic addition of new objects by clicking on an HTML button.

There are two ways of interacting with the physics scene: Events and JavaScript calls. When two physics objects collide, the system emits an *collision* event. It is necessary to register a listener to the object of interest to receive collision events. This is done in exactly the same way as for other DOM events: Developers can either use the new event attribute *oncollision* as shown in Line 40, or use the element's `addEventListener` method to register a JavaScript callback function. The received collision event contains information about the involved objects. XML3D geometry objects that have the CSS property *display* set to *none* are not considered during rendering, but can still be used to detect collision events as explained above. This is very useful to detect when objects enter a certain area in a scene.

The second way to interact with the physics world is via JavaScript. Each object that is marked as dynamic implicitly has an additional DOM API property called *physics*. This property contains a *PhysicsObject* with an interface that allows applying forces or an impulse to the center of the object or to a specific position. This makes it very simple for the user to interact with the physics engine. Listing 4 shows the few lines of code required to push an object with a mouse click. Figure 6.7 shows a scene demonstrating the interaction capabilities of XML3D Physics.

```
<script type="text/javascript">

  function pushObject(event) {
     var po = event.target.physics;
     if (po) {
       po.applyImpulse(event.normal.scale(-20), event.position);
     }
  }

</script>
...
<mesh src="#box" onclick="pushObject(evt);"/>
```

Listing 4: Applying an impulse to an object using the physics interface. The impulse goes along the negative normal at the position where the object was clicked.

6.3.1.4 Implementation

For prototyping we implemented the physics interpreter for XML3D as a NPAPI browser plug-in. This decision was mainly based on the fact, that this way we forced ourselves to keep renderer and physics engine separate. The communication is only possible through the DOM and DOM-API calls. As a result, this guaranteed us an orthogonal design. Of course, the implementation could easily be integrated into the core of the different XML3D implementations. For the physics simulation we have chosen the Bullet engine [CO15b], because it was free, open-source, and known to support all needed features robustly.

There are several ways of communication between the DOM and the physics plug-in. During the initialization phase the plug-in sets up the physics simulation and creates all physics objects with references to their corresponding objects in the DOM. It parses all the required information through the standard DOM API. This includes all the attributes and elements from the physics namespace as well as necessary data from the reference 3D scene, such as the structure of the tree, transformations, and geometry where no proxy geometry is given. The physics simulation can be started (and stopped) explicitly calling a corresponding JavaScript method on the <xml3d> element. As soon as the physics simulation has started, the calculations for the objects in the physics world are performed and changes get applied on the corresponding objects in the DOM. This involves mainly changes on the dynamic objects' transformations and firing of collision events. The plug-in monitors the scene in the DOM for:

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

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

Performance The major concern in using the HTML DOM as a physics scene graph together with the browser's plug-in API (NPAPI [Moz15]) was the overhead generated by the need to update the DOM from the plug-in. The DOM is a very generic data structure, primary designed for string operations. Browser vendors recently improved the performance of DOM operations, but the basic problem still remains.

Due to the implementation as a browser plug-in, there are three threads involved performing the physics simulation: The physics simulation thread, the plug-in thread, and the browser's main thread. The simulation thread runs the bullet engine's simulation loop in an arbitrary and configurable update frequency. The plug-in thread interfaces between the simulation and the browser: It collects the results of the simulation thread and sends messages to change the DOM accordingly. These DOM manipulation calls are executed in the browser's main thread.

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

We measured the time needed until the information is available in the DOM and compared them to the raw output of the physics engine. For the tests, we run the simulation at ∼60Hz, resulting in a maximum of 60 pose updates per object per second. As it can be seen in Table 6.1, the overhead for small and medium scenes is negligible. However, starting at ∼1000 updates per second, the DOM updates start lagging behind. We tested four variants passing the pose update information to the DOM with different results:

1. V1: Two *setAttribute* calls, one for rotation, one for translation. This method is a generic string based DOM setter method, thus the arguments need to be parsed after assignment.

2. V2: We gathered all pose updates including their target and transfered them to the browser in a single string. The string get parsed in JavaScript to reconstruct the assignments. Each assignment consists of a *getElementById* call to get the targeted element in the DOM and the two *setAttribute* calls as explained in V1.

3. V3: The XML3D specification defines *rotation* and *translation* properties of the <transform> element. These are of type *XML3DRotation* and *XML3DVec3*, which exist as native or JavaScript objects in the browser depending on the XML3D implementation. It's possible to create new instances of these types from within the plug-in. Then one DOM call is needed to assign those objects to the rotation property and translation property respectively.

4. V4: We used a *setPose(translation, rotation)* call to assign rotation and translation in a single step. As this method is not defined in the XML3D specification we simulated

Scene	1 crate	10 crates	100 crates
Sim. time (ms)	474	521	1991
Pose updates (PU)	298	2523	80851
Bullet PU/s	63.1	484.3	4060.8
DOM V1 PU/s	63.0	482.6	630.6
DOM V2 PU/s	62.9	480.6	785.6
DOM V3 PU/s	62.9	480.2	1233.5
DOM V4 PU/s	62.9	481.5	2270.3

Table 6.1: XML3D Physics performance in comparison to the raw physics simulation. The test scenes consist of 1, 10, and 100 falling boxes. The simulation steps, time, and pose updates (PU) are measured until no further updates appeared due to convergence of the simulation. DOM V1-V4 refers to the different options binding the PUs to the DOM representation.

it in JavaScript. Just as in V3, we used native or JavaScript objects as parameters for the function.

As it can be seen from Table 6.1, the performance varies depending on the type of DOM assignment. It can be said, that an asynchronous DOM call from the plug-in is expensive and should be avoided. The JavaScript engine seems to have better access to the DOM: In V2 the instructions need to be parsed before execution and three DOM-API calls are performed. Nevertheless, this solution is faster than calling just two methods from within the plug-in. Introducing a `setPose` method to the XML3D <transform> element allows setting the translation and rotation in single call. Using this method we can achieve up to 2300 DOM updates per second, which seems to be the limit for the browser frameworks with DOM updates running in a single thread. The simulation produces approximately 4000 updates per second running at 60 simulation steps per second with 100 objects and lots of intra-object collisions. Figure 6.8 shows the rendering of a similar scene as used for the performance tests, other scenes produced similar results.

To narrow down the bottleneck between the physics simulation and DOM updates, we recorded the pose updates and executed them from the JavaScript environment alone without the plug-in overhead. Using the same kind of assignment as in V2, we can achieve 12500 pose updates per seconds. Using XML3DRotation and XML3DVec objects for the assignment (similar to V3), we can update the poses even 54,975 times per second. The results from these tests show clearly that the current bottleneck is the asynchronous DOM update from within the plug-in. Avoiding the NPAPI interface to access the DOM would gain a significant performance improvement. Nevertheless, the implementation is fast enough for a first prototype revision that has to deal with the overhead coming from the plug-in API.

6.3.1.5 Conclusions

We presented a extension mechanism for XML3D that allows running rigid body physics simulation and collision detection without extending the abstract model: We added the required information using annotations instead of adding extra nodes to the XML3D specification. Thus, we proposed a fully orthogonal design, that for the first time allows annotating the 3D scene directly, but keeps the visual rendering and physics simulation

independent and exchangeable. We demonstrated the orthogonality of our approach with a physics simulation plug-in that works for all XML3D implementations, the two native XML3D browsers and the WebGL/JavaScript renderer.

The physics annotations can be easily added by hand or – more convenient – through our XML3D Blender exporter that we extended for that purpose. The physics simulation combined with the JavaScript API and the collision events greatly simplifies the creation of interactive 3D web applications. It enabled bachelor students with basic XML3D knowledge to create an interactive 3D bowling game within two days, including modeling, game logic development, and design of the webpage (Figure 6.6).

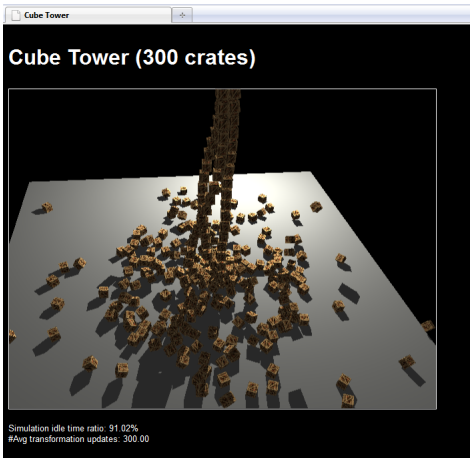


Figure 6.8: Performance test scene. 300 crates in a colliding tower, rendered using realtime ray tracing.

From our measurements we see, that the system is running reasonable fast for small and medium sized scenes. We found that the communication between browser and plug-in is not yet fast enough to fully reach the performance level of native physics engines. In this first revision we preferred the guaranteed orthogonality over performance.

At the time of implementation, integrating the physics engine using the plug-in API was the only option to add physics to all XML3D implementations. Today, multiple JavaScript physics engines are available, most prominent *ammo.js*[21], which is the C++ Bullet engine compiled to JavaScript via Emscripten [Zak11]. Although still slower than the native Bullet engine[22], this implementation would not suffer from the slow browser to plug-in communication and would finally run without any plug-in.

6.3.2 Extending XML3D via Semantic Annotations

The approach we used for physics annotations is applicable to a number of other simulation domains such as audio and haptics by simply adding other annotations to the scene description. In particular rendering spatial audio is a functionality than can be found in many rendering system. It would be possible to exploit the spatial description of XML3D and to extend the scene description with audio-related entities such as sound sources and

[21] `https://github.com/kripken/ammo.js/`, accessed 18 December 2015.

[22] `https://hacks.mozilla.org/2013/12/gap-between-asm-js-and-native-performance-gets-even-narrower-with-float32-optimizations/`, accessed 18 December 2015.

sound characteristics of surfaces. The Web Audio API [W3C12b] allows modifying sounds, e.g. based on the position and velocity of the sound source, and hence could be used to implement such an extension.

The concept of annotating additional scene information for various simulations can be generalized by adding semantics using a generic metadata approach. This is a common approach for the Semantic Web [BLHL01] that promotes common formats and protocols, for instance the Resource Description Framework (RDF) [W3C04] and its HTML-compliant notation RDFa [W3C12a]. Consequently, we can use these technologies to also annotated XML3D with additional information that can be used in specific domains. For instance, an object in the XML3D scene could be annotated as a car that is controlled by a traffic simulation. A similar approach has been used in [KLN*10], where an AI engine is identifying semantic entities in the 3D scene from annotations and applies an agent simulation on the XML3D scene (see also Section 9.3).

In general, XML3D can be used as the central 3D model reference, while simulations monitor the scene and expose their results to the scene. The generality of the DOM, the fast scripting engines and the ubiquity of browsers give good reason to run such simulations in the browser context for making them broadly available. Issues that may appear if multiple simulations try modifying a scene concurrently and other challenges involved in running distributed multi-user and multi-simulation scenes (also known as virtual worlds) are not addressed by this thesis and we leave this for future research.

6.3.3 Volume Rendering

The current abstract model of XML3D considers geometry represented as geometric *surface* primitives such as points, lines, and triangles only. This is a natural choice since these are the primitives supported by recent 3D graphics APIs. Most other representations can be (or need to be) transformed to these primitives before rendering. This can be done using the Xflow data processing functionality. Hence, we refrained from adding additional surface-geometry elements to the abstract model.

Volume rendering allows users to see, explore, and analyze the inner structures of real and virtual objects. In addition to its use in industry fields such as meteorology, medicine, oil prospecting, and nondestructive testing, volume rendering is increasingly used in computer games to visualize effects such as fire, smoke, and clouds. A 3D volumetric data set is a set of samples, which represents 3D spatially varying properties of a source object, e.g. density, heat, pressure, color, etc. Typically, volumetric data sets are acquired from measuring properties of real objects using techniques such as computerized tomography (CT), magnetic resonance imaging (MRI), seismic imaging, radar, etc. Another common source of volumetric data sets are numerical simulations, for instance computational fluid dynamics or FEM simulations.

In most cases data samples are acquired on a regular 3D grid or, if other patterns are used, get resampled to a regular grid before rendering. Since the grid contains discreet samples, interpolation techniques need to be applied during rendering of a volumetric data set.

A volumetric data set may be rendered indirectly by detecting surfaces in the volumetric data set and fitting geometric primitives to these surfaces (*Indirect Volume Rendering*). For instance, a common technique is the detection of isosurfaces, i.e. surfaces that represent

points with a constant value inside the data set. A well-known approach for extracting isosurfaces from volumetric data is the marching cubes algorithm [LC87], which can be implemented as an Xflow operator. Such extracted surfaces can be represented as triangle meshes and, hence, can be rendered with XML3D without any extensions.

In contrast, *Direct Volume Rendering (DVR)* operates on the original data omitting an intermediate geometric representation. Many techniques have been proposed for DVR (e.g. [Lev88, LH91, LL94]). A very common approach for direct rendering of volumetric data is volume ray casting, where samples are taken along a ray through the data set. Today, programmable graphics hardware allows for implementing volume ray casting in the fragment shader including early ray termination and empty-space skipping [KW03]. Thus, direct volume rendering can be achieved at interactive frame rates even on modest graphics hardware.

Obviously, it would be possible to render a cube using the <mesh> element, pass the volume field as a material parameter and implementing volume ray casting in the material description using the programmable material concept. However, this approach comes with some disadvantages: First, the rendering would be explicit. Hence, it prohibits the system from using an alternative rendering algorithm. Secondly, such a ray caster has no access to other scene objects during ray casting in the fragment shader. Consequently, in order to mix volume rendered objects with explicit geometry, the renderer has to organize the rendering in a way the results can be composed using z-buffer techniques. Since XML3D does not expose the z-buffer concept, the user has no means to control the compositing. Finally, this approach would abuse the semantic of the <mesh> element, which should represent a triangulated cube in the first place. As a result, it is worth considering to extend XML3D for supporting volumetric data sets as first class citizens.

Nina Istomina proposed exactly this in her master thesis [Ist13]. Please refer to this thesis for all details that concern implementing volume ray casting in a WebGL environment, optimizations, and mixing volumetric rendered objects with other objects. In this section, we will focus on the proposed extension of XML3D's abstract model necessary to support volume rendering.[23]

We extend the abstract model by a <volume> element. The <volume> element is very similar to a <mesh> element: it is a leaf element of the scene tree, its placement is influenced by the transformation hierarchy, it can be styled using CSS, and the material is derived from the material property. All events available for a <mesh> element are also available for the <volume> element.

The <volume> element is a sink of the dataflow graph with mandatory and optional parameters. The *field* parameter is mandatory and contains the volumetric data set as a one-dimensional array. It expects data as scalars of type unsigned byte.[24] The *gridSize* parameter contains 3 integers that hold the dimensions (width, height, depth) of the volumetric data set and are used to construct the three-dimensional array from the *field* parameter. The volumetric data set is rendered within a cube, which represents the volume in 3D-space. By default, this cube has coordinates from -0.5 to +0.5 in object space.

[23] The extension has been slightly adapted compared to the proposal of Istomina in order to reflect the latest changes in XML3D.

[24] This is a practical consideration imposed by the current implementation. Conceptually, other precisions would work as well.

The dimensions of the cube can be adjusted using the optional *size* parameter. Also, this cube is influenced by the transformation hierarchy.

With these parameters, a naive definition of a volume would look like this:

```
<volume>
  <!-- The volumetric data set -->
  <ubyte name="field">0 0 33 ...</ubyte>
  <!-- The dimensions of the regular grid -->
  <int name="gridSize">53 83 70</int>
  <!-- The dimensions of the volume in the scene -->
  <float name="size">2 2.5 2</int>
</volume>
```

Due to the size of the scalar field, the volume data is typically stored in an external resource:

```
<volume>
   <data src="binaries/ndt_volumes.bvd" />
   <float3 name="size">5 10 6</float3>
</volume>
```

In this example an external binary file contains the volumetric data. Istomina proposes a binary compressed format as delivery format for volumetric data, which could also be encoded in a Blast container (see Section 8).

We can also leverage the concepts of the XML3D architecture for the construction of volumes. With Xflow, we can support several common representions for volumetric data. For instance, an Xflow operator can be used to construct the data field from an array of images:

```
<volume compute="field, gridSize=xflow.volumeDataFromImages(slice)">
   <texture name="slice">
    <img src="../sirona_implant_planning/0000.png">
    <img src="../sirona_implant_planning/0001.png">
    <img src="../sirona_implant_planning/0002.png">
    <img src="../sirona_implant_planning/0003.png">
    ...
    <img src="../sirona_implant_planning/0255.png">
   </texture>
</volume>
```

In fact, a GPU-based renderer stores the data field in a 2D or – if available – in a 3D texture and exploits hardware assisted texture mapping for the interpolation of samples. However, since this is a renderer-specific optimization the basis for the <volume> element is a generic regular 3D field and it is up to the implementation to store the data in textures or, in the example above, to use the provided image slices as-is.

In addition to the volumetric data, the user needs to provide information on how the volumetric dataset should be rendered. We considered the definition of the rendering style similar enough to materials for surface geometry, thus we decided to reuse the <material> element also for volumes. Similar to geometry materials, we have a predefined material models for volumes that can be configured with a set of parameters. Currently, the

implementation offers a single predefined material model based on a simplified emission-absorption model. It integrates colors and extinction coefficients along each viewing ray based on the properties of each cell in the 3D data field. It does not take lighting, shadowing and scattering into account, i.e. lights placed in the XML3D scene are not considered. It has a *transferFunction* parameter that takes a look-up table represented as a 1D texture:

```
<defs>
    <material id="volume-material" model="urn:xml3d:volume:transfer">
        <texture name="transferFunction">
            <img src="transferfunctions/implant_planning.png">
        </texture>
        <float name="baseIntensity">30.0</float>
    </material>
</defs>
```

The *baseIntensity* parameter can be used to fine-tune the global volume opacity without changing the transfer function. A number of renderer-dependent parameters allow configuring the ray casting, including *stepSize*, which will determine how many samples are take during ray traversal and *rayTerminationAccuracy* for early traversal termination before the full opacity has been reached. The material overrides mechanism discussed in Section 6.1.1 also applies to volumes: All parameters can be defined inside the <material> element and thus be shared across multiple volumes, or alternatively, be defined per object as parameter of the <volume> element.

Again, Xflow can be used to extend the functionality of the material model. For instance, we provide an Xflow operator that supports a parameterizable color ramp:

```
<material model="urn:xml3d:volume:transfer"
          compute="transferFunction=xflow.calcTFFromColorTable(density,color)">
    <float name="density"> 0.0 0.4 ...</float>
    <float4 name="color"> 0.1 0.4 0.8 0.1 0.5 0.8 0.3 0.2 ...</float4>
</material>
```

The output of this operator is – similar to the example above – a look-up texture. Hence it offers the same fast evaluation time in the shader. In contrast to the operator above, the user is still able to adapt the rendering style during runtime.

Conclusion In this section, we proposed an extension of XML3D's abstract model that allows describing volumetric data sets in XML3D. Figure 6.10 shows a web page with multiple volumetric data sets in a single XML3D scene. The proposed extension also allows rendering volumetric data sets mixed with triangle meshes (see Figure 6.9).

We showed that we can reuse the concepts of the XML3D architecture when extending the abstract model. This includes the HTML integration including CSS and events, the generic data model, dataflow processing, and delivery of the data using Blast.

The only concept of the XML3D that has not (yet) been applied to volumes is the use of programmable materials. While we leave this to future work, we are convinced it would be possible to leverage shade.js for exposing the configuration of the rendering style and lighting to the user without tying the rendering to a particular algorithm.

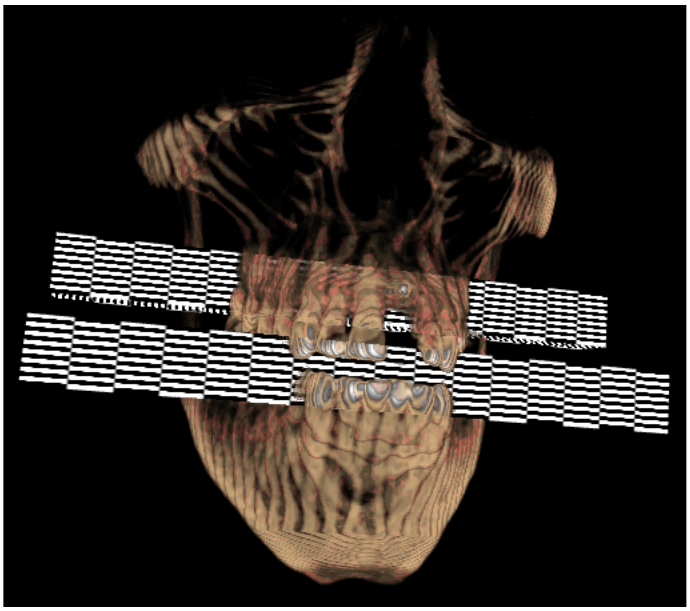


Figure 6.9: Volume object with two intersecting geometries. The geometry is visible on transparent volume regions, the volume overlaps the geometry on its opaque region (from [Ist13]).

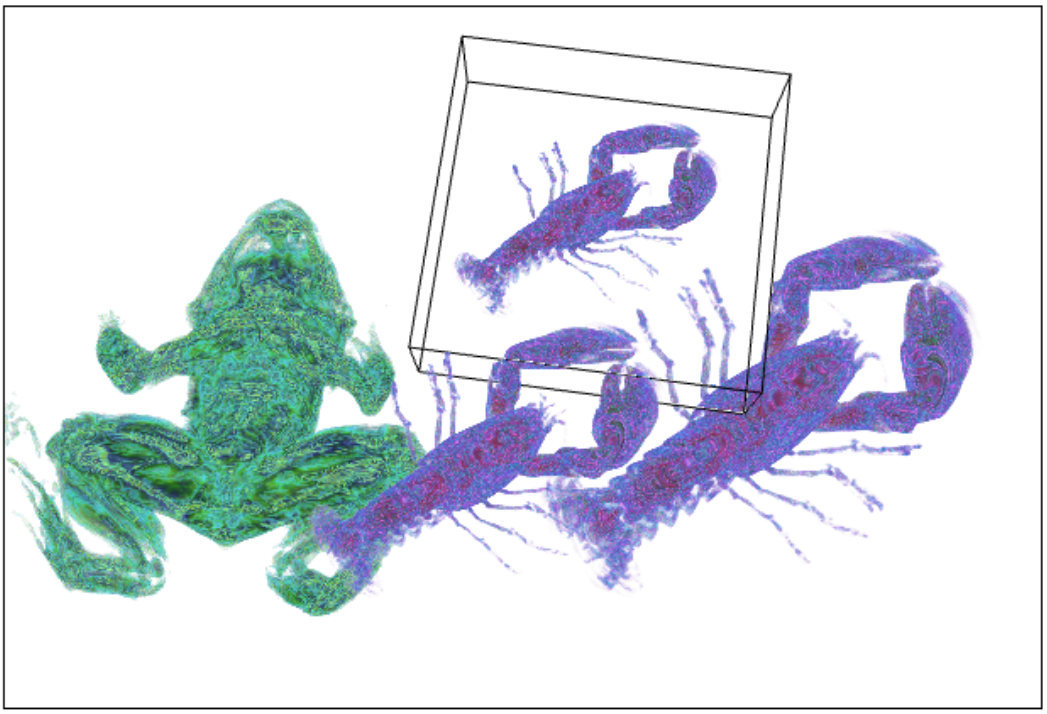


Figure 6.10: An example of XML3D scene with four <volume> objects. Using the XML3D event attributes, a bounding box is drawn for the object below the mouse cursor.

7 Programmable Materials

In most games specialized and optimized GPU shaders are closely intertwined with game assets that are prepared especially for these predefined shaders as well as for the game engine and renderer. Such a shader will not work on its own and cannot be applied in other scenes unless the exact same environment is used. Most other applications, however, would benefit strongly from the concept of more general *materials* that can easily be reused in different scenes and with arbitrary rendering systems. Such materials could be exchanged, refined, verified, organized in reusable libraries, and shared across the Internet. For this to happen, a material needs to be self-contained and independent of 3D scene descriptions and rendering algorithms. The representation and efficient rendering of such generic material descriptions is still an unsolved issue.

All applications that try to offer such general material concepts require an adaptation logic and a process for generating shaders specialized for a concrete but yet unknown *execution environment*. This includes adapting to available uniform and per-vertex parameters, supplied textures, the capabilities of the current renderer, and the optional functionality of the GPU. As a simple example, consider a material that takes its diffuse coefficient either from a default constant value, a per-object uniform variable, an interpolated per-vertex attribute, or from a supplied texture map, such as:

```
kd = Color(0.8); // default value
if (exists(diffuseMap) && isTexture(diffuseMap) && exists(texcoords)) {
  kd = sample(diffuseMap, texcoords);
}
else if (exists(vertexColor)) {
  kd = vertexColor;
}
else if (exists(uniformColor)) {
  kd = uniformColor;
}
```

Listing 5: Simple material logic

Surprisingly, describing such a logic is not possible in current shading languages, because none of them allows introspecting the execution environment, i.e. to query the type and existence of parameters.

Such adaptivity is often implemented in the application through Ubershaders [McG05, SJBF10]. Obviously, the features of an Ubershader are never complete and customizing the logic requires changing the application itself. The same restriction applies to approaches exploiting the metaprogramming facilities of the host-language [MQP02], where the adaptation logic is still determined at compile time of the application and cannot adapt to arbitrary scenes and contexts as required for the Web.

A material can only be portable between applications if the adaptation logic is part of the material description itself. Hence, we need to shift this logic from the application to the material description.

In this chapter we present *shade.js*, a system for specifying adaptive, self-contained, and portable material descriptions. We make the following key contributions:

- We propose a novel shading language that offers introspection of its execution environment as an integral part of the language (Section 7.3). This gives authors the possibility to write materials that adapt to their current execution environment independent of the application.
- shade.js is the first shading language to exploit the polymorphism of a dynamic language. It deliberately refrains from declaring types and other qualifiers (replacing it with type inference). As a result, we achieve more generic material descriptions that work for a wider range of input parameters and renderers.
- We present an accompanying compiler that performs the required analysis to specialize the generic material description to a specific execution environment at runtime (Section 7.4). During this analysis, the compiler infers types (including type checking), semantics, and compute frequencies. Then it optimizes the shader code accordingly.
- We generate code and present results for four different rendering algorithms and three renderers: GLSL shaders for forward and deferred rasterization via WebGL (Section 7.4.5) as well as OSL shaders for Whitted-style ray tracing and path tracing using Blender's Cycles render engine (Section 7.4.5).

This part of the thesis has been previously published in [SKSS14].

7.1 Related Work

In this section we discuss the limited means of specialized shaders for adapting to their execution environment and the proposed approaches to tackle those shortcomings.

7.1.1 Specialized Shaders

There are numerous of "little languages" [Ben86] that focus on the problem domain of shading in 3D computer graphics. Such shading languages include the RenderMan Shading Language (RSL) [HL90, Ups89] and the Open Shading Language (OSL) [GSKC10]. Both, RSL and OSL are fully procedural languages inspired by C with additional domain-specific data types, e.g. for colors and vectors. Both languages provide means to define values for parameters that are not available in the execution environment (default values) and ad hoc polymorphism using function overloading. However, all parameters and variables are strictly typed and thus fixed at authoring time. Moreover, there is no possibility to alter the control flow based on the type or existence of supplied scene parameters (e.g. vertex attributes). Thus, the adaptivity of these shaders to new scenes and renderers remains limited.

GPU languages such as Cg [MGAK03], HLSL [Mic02], and GLSL [KBR03] are highly specialized. They are specifically designed to run efficiently on the programmable pipelines of recent graphics cards. Similar to RSL and OSL, these languages are inspired by C. They come with a static type system and provide no means to introspect their execution environment. Their polymorphism is restricted to operator overloading and default values. Even worse, authors have to declare the sources of parameters (uniform or per-vertex) at authoring time using type qualifiers.

The compiler of GPU languages has very limited means to communicate with the execution environment. For instance, the compiler cannot report expressions that need only be evaluated once per geometry patch back to the execution environment in order to avoid unnecessary per-fragment operations. Thus "a folklore has developed over hand coding these types of optimizations" [HL90]. Unfortunately, this has not changed for the last 25 years.

Preprocessor directives are a common way to change GPU shader programs to compile in different execution environments. Some extensions in GLSL implicitly define macros for the preprocessor, making it possible to adapt the shader code based on optional hardware capabilities. However, all other code changes performed by the preprocessor need to be controlled by the application, which implements the necessary adaptation logic. As a result, the shader is tied to a specific application.

HLSL Shader Model 5 [Mic08] adds *Dynamic Linking*[1] as a more sophisticated approach to shader permutations: It supports using virtual functions in a shader with multiple concrete implementations compiled into a single shader program. The application can then choose the concrete implementations to be used at runtime. Compared to Ubershaders with dynamic adaptation based on setting uniforms, this approach yields better performance. However, the adaptation logic, i.e. choosing the appropriate implementation, remains in the application.

7.1.2 Meta-Systems

A number of approaches have been proposed to solve issues related to specialized shading languages. In general, these approaches develop meta-systems that create code for one or more specialized languages in the end.

McCool et al. [MQP02] propose shader metaprogramming within the host language. The proposed API allows for using generic types at authoring time by exploiting C++ templates. The types are checked and determined at compile time. Additionally, the API offers specialization during runtime, e.g. branching based on a query for the type of a parameter. However, this approach has some disadvantages: The C++ polymorphism is only available for those parameters that can be determined during compile time of the application. Also the adaptation logic is determined already at application compile time, resulting in predefined specializations similar to Übershaders. Additionally, shader metaprogramming offers no means to query the existence of parameters in the execution environment.

Similarly, Kuck et al. [KW09] specialize shaders at runtime using C++ classes as an

[1] Not to be confused with flexible linking of shaders for multiple pipeline stages, e.g. within a GLSL Program pipeline object.

abstraction layer. Again, the adaptation logic is fixed at compile time of the application and cannot be altered by the shader author.

Vertigo [Ell04] is a metaprogramming system using the functional language Haskell as host language. It supports shading and expression rewriting for optimizations, but offers no adaptivity based on the execution environment.

7.1.3 Declarative Approaches

Declarative approaches such as Shade Trees [Coo84, AW90, MSPK06] do not expose the program's control flow to the user. As a result, the author cannot adapt the control flow based on the current execution environment. For instance, it is not possible to branch a shading program based on the existence of a specific parameter.

Additionally, Shade Tree approaches come with the potential for type mismatches caused by heterogeneous execution environments [AW90]. Abstract Shade Trees (AST) [MSPK06] extends Cook's Shade Trees with semantically rich vector types that include the vector's coordinate space and the interpretation (e.g. normal, direction, point). It offers a type matching algorithm that detects and tries to automatically resolve type mismatches by adding conversion nodes into the tree. This results in a higher degree of type polymorphism compared to previous approaches.

More recent approaches mix declarative and imperative elements: MetaSL [Men10] offers authoring of procedural building blocks with arbitrary input and output parameters. The Material Definition Language (MDL) [Nvi12] allows describing physically-based materials independent of the used rendering algorithm. Similar to OSL, MDL provides predefined radiance functions to describe reflection, transmission and emission of light at a surface. While the parameters of these radiance functions can be customized fully procedurally (e.g. for multi-texturing), the material structure, i.e. the used combination of radiance functions, cannot alter based on surface parameters. This makes MDL easier to analyze compared to a procedural approach such as OSL. While this limitation guarantees physical-plausible materials, it neglects the freedom authors might want to have to achieve a desired effect. In our approach, we can detect if a material is physically plausible or not and generate warnings if desired. However, we do not restrict the expressiveness of the material description.

Both, MetaSL and MDL, use a language inspired by the C for the procedural parts. Hence, the resulting materials come with limited adaptivity due to fixed types and lack of introspection.

In GPU languages, logically related computations are spread over multiple stages in the rendering pipeline.[2] RTSL [PMTH01], Renaissance [Aus05], and Spark [FH11] tackle this issue using compiler technology to partition one shader description to the different stages of a multi-stage pipeline. In shade.js, we use a similar approach to partition computation from a single material description to the CPU, vertex shader, and fragment shader stage. However, the previous approaches require typed input parameters and provide no mechanism to introspect their execution environment.

Other systems address partitioning shaders into multiple passes [CNS*02, RLV*04] and shader simplification [Pel05, SAMWL11, WYY*14]. These techniques are related but or-

[2] For a in-depth discussion on this issue refer to Section 4.1.

thogonal to *shade.js* and could eventually be included.

A type of inverse approach to adaptation is described in [LS02], where the geometric data is specialized by a compiler based on the knowledge of the applied shading program.

7.2 Our Approach

In order to achieve more adaptive materials, we propose a new domain-specific language based on a subset of JavaScript. We have chosen JavaScript mainly for its polymorphic features: It is not explicitly typed, an algorithm described in JavaScript can be used with any types that provide the necessary methods used in the algorithm. This allows writing procedures more generically and increases the expressiveness of the material description.

Additionally, the language offers mechanisms to introspect its execution environment. We provide three different kinds of introspection: Authors can query the environment for i) the existence of parameters; ii) the type of parameters; and iii) the availability of optional functions. The JavaScript language provides natural ways to query types and the existence of properties of parameters that we exploit for shade.js. As a result, authors of a material can adapt its control flow based on information about the execution environment (see examples below).

We integrated shade.js into xml3d.js (see Section 7.5). Thus we are able to describe the material in the host language. Moreover, shade.js can easily be ported to other WebGL engines and – with little extra effort – also to native renderers. Since a shade.js shader is written in legal JavaScript, it would even be possible to run it in a dynamically interpreted environment. Although this feature is interesting – for instance for debugging – we aim to generate specialized shader code for common renderers and platforms.

To this end, shade.js comes with a JIT compiler that performs static code analysis based on the concrete execution environment at runtime. At this point in time the compiler can evaluate all queries into the (current) execution environment statically. Depending on the result of the analysis, the compiler changes the control flow of the shader and eliminates dead code. Additionally, the compiler infers all variable types based on the types of the parameters provided by the execution environment. As a result, the cross-compiled code is adapted and highly specialized to its execution environment, i.e. it has static types and the resulting control flow only depends on the values of parameters, not on their types. If the description fails to compile, the compiler can produce meaningful error messages based on the results of the code analysis.[3]

In addition to these novel concepts, we cherry-pick a few of the most useful and important concepts from existing approaches and integrated them into shade.js:

- For portability across different lighting systems we adopt interfaces based on radiance closures similar to OSL [GSKC10]. The compiler resolves the radiance closures and generates appropriate code for the target rendering system.
- Similar to RTSL [PMTH01] and Renaissance [Aus05], our compiler partitions the shader program by computation frequencies in order to optimize the runtime per-

[3]This is in stark contrast to e.g. GLSL: Here, using a variable or texture that is not defined in the execution environment will lead to a memory access with arbitrary values and undefined results.

formance, i.e. it extracts parts of the code that can be executed on the CPU or on the vertex shader stage, respectively.

- Similar to [MSPK06], we have a semantically rich type system that has additional type properties such as computation frequency (constant, uniform, or per-vertex), interpretation (e.g. color, normal, point), and the coordinate space (e.g. object space, world space, etc.) of a vector.

In contrast to previous approaches, we deliberately refrain the authors from declaring these properties. Instead, the compiler automatically derives them from the execution environment and interfaces used in the shader code.

7.3 Language

Shade.js supports a subset of JavaScript that includes all arithmetic, logical, and assignment operators, conditional statements, loops, break, and continue statements.[4] We support the built-in data types `undefined`, `number`, `boolean`, `string`, `object`, and `function`.[5]

However, we do not support the entire functionality of JavaScript: Array sizes may not change and array elements need to have homogeneous types. Also, we have restricted support for JavaScript objects. We support functions as an abstraction mechanism, but not the JavaScript prototyping functionality. These restrictions are not of a conceptual nature but we enforce them for now to simplify the mapping to the target languages.

We have added predefined objects for vectors with two, three, and four components, and for 3×3 and 4×4 matrices. Similar to GLSL, vectors have methods to swizzle components. The vector and matrix objects are immutable, since this allows for a better mapping to target languages that provide vectors and matrices as built-in primitive data types.

7.3.1 Polymorphism and Introspection

In shade.js, we exploit the properties of the JavaScript *object* to query the existence of parameters and optional renderer capabilities and the **`instanceof`** and **`typeof`** operators to query the type of parameters. Recall the pseudo code from Listing 5. With shade.js one can implement this logic in straight-forward JavaScript syntax:

```
function shade(env) {
  var kd = new Vec3(0.8);

  if (env.texcoords && env.diffuseMap instanceof Texture) {
    kd = env.diffuseMap.sample2d(env.texcoords).rgb();
  } else if (env.vertexColor) {
    kd = env.vertexColor;
  } else if (env.uniformColor) {
    kd = env.uniformColor;
  }
  ...
}
```

[4] Break and continue are only supported without labeling.

[5] For a reference on supported functionality, additional types and available radiance closure refer to: `https://github.com/xml3d/shade.js`.

The first parameter of the shade function is an object that provides all parameters of the execution environment as its properties. If an input parameter does not exist, accessing the property returns *undefined*, which evaluates to *false* in logical expressions as well as in expressions with the *instanceof* operator. This mechanism cannot only be used to query for the existence of shader parameters but also for the availability of optional functions, accessible through the *this* keyword:

```
if (this.fwidth) {
  // The execution environment supports derivatives
  var fw = this.fwidth(env.texcoord);
  ...
}
```

Since shade.js uses implicit types and sources for all input parameters, we can further simplify the logic of Listing 5 by replacing *diffuseMap*, *vertexColor*, and *uniformColor* with a single input parameter *color*:

```
function shade(env) {
  var kd = new Vec3(0.8);

  if (env.color && env.color.sample2d && env.texcoords){
    kd = env.color.sample2d(env.texcoords).rgb();
  } else if(env.color && env.color.rgb){
    kd = env.color.rgb();
  }

  ...
}
```

In this code snippet, instead of using the *instanceof* operator, we check for available methods of the *color* parameter to determine its type: if *sample2d* is defined, it can be used as a texture and if *rgb* is available, it can be converted to a color. By checking for available methods instead of explicit types, we further expand the range for supported input parameters. For instance, both *Vec3* and *Vec4* implement the *rgb()* method and are therefore candidates for a color. Additionally, since the source of input parameters is implicitly determined, *color* can be provided either as a uniform parameter or as a vertex attribute. In cases where the execution environment provides both – a vertex attribute and a uniform parameter with the same name – the more specific per-vertex definition is used. See Figure 7.1 for a number of materials adapted to various execution environments.

Another step towards a more adaptable shader is the support of multiple, semantically equivalent input parameters of different names. The following example shader accepts the diffuse color according to the naming conventions of Wavefront OBJ (Kd), COLLADA (diffuse) and XML3D (diffuseColor):

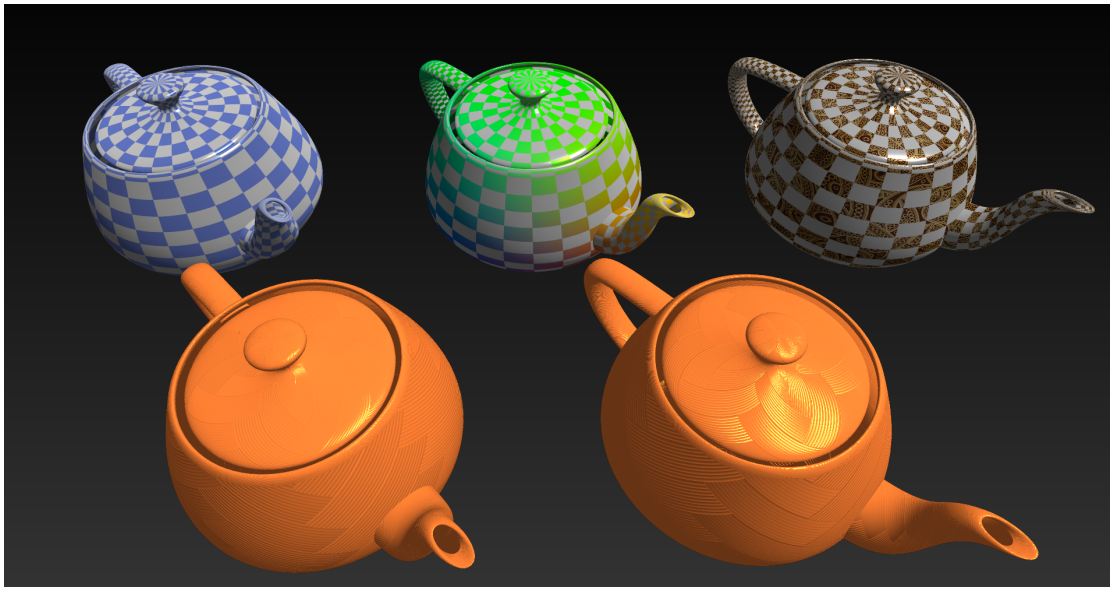


Figure 7.1: Example of adaptable shaders using polymorphism and introspection. The checkerboard shader used in the rear of the image works for a uniform color (left), vertex colors (middle), and a texture (right). The orange teapots in the front feature normal mapping, using the tangent vertex attribute if available (left) and otherwise an approximated tangent based on the derivative of the vertex position (right).

```
function shade(env) {
  var kd = env.Kd || env.diffuse || env.diffuseColor;
  var finalKd = new Vec3(0);

  if(kd && kd.sample2d && env.texcoord){
    finalKd = kd.sample2d(env.texcoord).rgb();
  } else if(kd && kd.rgb){
    finalKd = kd.rgb();
  }
  ...
}
```

Again we interpret all input parameters either as texture or color, which is required e.g. to properly support the COLLADA convention. Converting this 8-line code snippet to an equivalent GLSL shader using preprocessor directives to accept input arguments for the diffuse color with different names and varying types, results in over 40 lines of code.

Similarly, we can handle input arguments with slightly varying semantics. For example, the following code accepts both *transparency* (with 1 being fully transparent) and *opacity* (with 1 being fully opaque):

```
function shade(env) {
  var alpha = 1;
  if(env.transparency != undefined)
    alpha = 1 - env.transparency;
  else if(env.opacity != undefined)
    alpha = env.opacity;
  ...
}
```

7.3.2 Radiance Closures

Similar to OSL, we provide a set of radiance closures to interface with the lighting system. A radiance closure enables the evaluation of interactions between lights and the material surface without providing an explicit viewing direction. Radiance closures are parameterizable. The return value of the *shade* procedure is either a radiance closure, a linear combination of radiance closures, or *undefined*. The latter discards the current fragment.

We implemented a set of radiance closures for shade.js, including the Oren-Nayar [ON94], Phong [Pho75], Cook-Torrance [CT82], and Ward [War92] reflection models, mirror-like reflection, and others. A checkered material that discards half of the squares is shown in Listing 6, and the resulting shader applied to a cube and ray traced using the Cycles renderer in Figure 7.2.

```
function shade(env) {
  var smod = (env.texcoord.x() * env.frequency) % 1.0,
      tmod = (env.texcoord.y() * env.frequency) % 1.0;

  if((smod < 0.5 && tmod < 0.5) || (smod >= 0.5 && tmod >= 0.5)) {
       return; // Discards this fragment
  }

  return new Shade().diffuse(env.kd, env.normal)
                    .phong(env.ks, env.normal, env.shininess);
}
```

Listing 6: A checkered material described in shade.js.

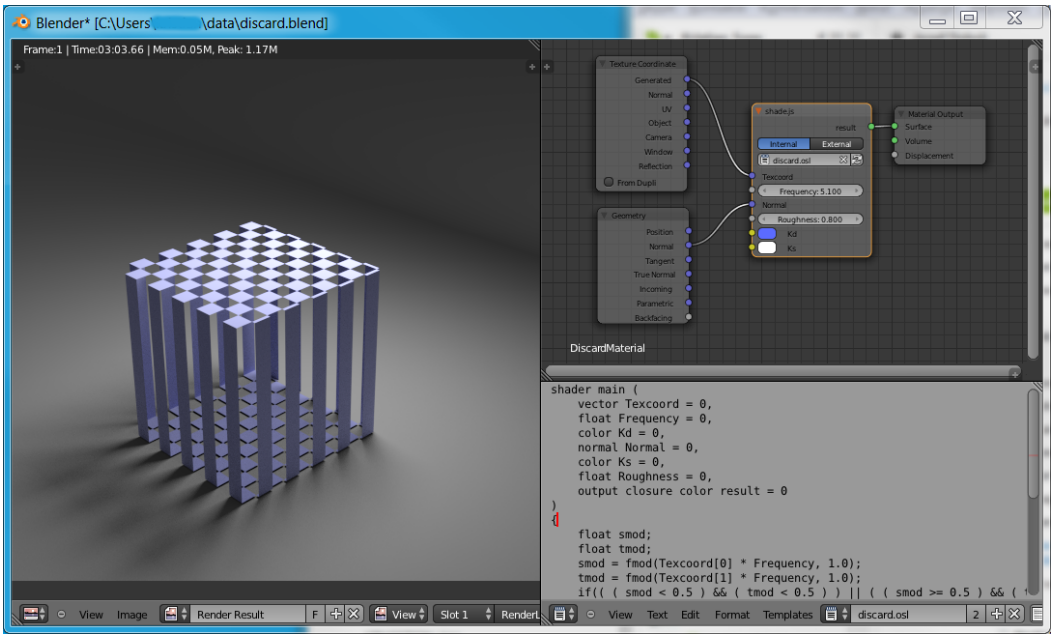


Figure 7.2: shade.js shader from Listing 6 compiled to OSL and rendered with Cycles in Blender.

7.4 Compiler

shade.js comes with a compiler framework that

- integrates the light computations as defined by the radiance closures,
- performs type inference and dead code elimination based on the input types in order to specialize the shader by removing polymorphism and resolving introspection,
- computes all type qualifiers that are required for the target language. This includes the source (e.g. uniform or vertex attribute) and the semantic (e.g. vector, color, or normal) of parameters,
- performs optimizations usually done manually by the shader author, and
- generates code for the target shading language.

We allow loops and conditional branching in shade.js code. Therefore, we have to formulate all our program analyses as monotone data flow frameworks [NNH99].

7.4.1 Specialization

As part of the compiler-based specialization, we have to derive the concrete types of all used variables. This is necessary because the target languages require explicit types.

JavaScript provides the built-in data types undefined, number, boolean, string, object, and function. The compiler infers these types based on the literals in the program and the type information of the parameters received from the renderer. In JavaScript, the number type does not distinguish between integers and doubles. Our type inference tries to represent numbers as integers wherever possible. This is a common approach in commercial JavaScript compilers [HG12] and necessary, for instance, for loops and dynamic array access in GLSL and OSL.

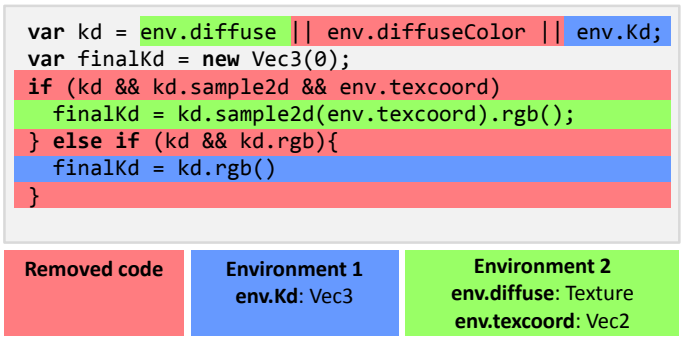


Figure 7.3: Example on how dead code elimination is applied to remove the use of introspection features. Two execution environments are described at the bottom. Depending on the execution environment, either the code highlighted in blue or in green remains. The code marked red is removed for both execution environments. The superfluous assignment of a default value to *finalKd* is removed by the target language compiler.

Introspection is only available in shade.js and must not appear in the target languages. In contrast to GPU shader compilers, which may perform constant propagation and dead

code elimination up to an arbitrary extent in order to optimize the performance of a shader, a thorough analysis is essential in shade.js to guarantee all introspection features (including branches that depend on the result of the introspection) are removed. See Figure 7.3 for an example.

It is not sufficient to only evaluate constant expressions (constant folding). The compiler has to also perform an analysis to substitute variables by a constant value if the variable holds this constant whenever execution reaches a program point (constant propagation). We eliminate unreachable branches in the same analysis.

Note that the result of the specialization does not depend on the values of input parameters, but only on types and computation frequency. Thus, changing the value of uniforms and attributes does not invoke recompilation.

7.4.2 Error Reporting

Being a dynamically typed language, JavaScript reports basic syntax errors immediately but type related errors only at runtime. In addition the error reporting of JavaScript is very relaxed. For instance, it tolerates the use of undefined variables as long as no property is accessed. This becomes especially confusing when undefined variables are used in arithmetic expressions: *NaN* (Not a number) is used to represent the result of a mathematical calculation that cannot be represented as a meaningful number. This special value propagates and may (or may not) lead to an unexpected result in a very different location.

Despite being a valid subset of JavaScript, shade.js has stricter error checking. Once the type inference and dead code elimination have been performed, the types of all variables need to be determined. Consequently, the use of undefined values or invalid types is not supported and will be reported immediately during compilation and prior to execution. These errors usually relate directly to missing or incorrect input arguments, thus the compiler can create descriptive error messages. For instance, the compiler generates an error when a non-existent parameter is accessed (without testing for its existence first):

```
NotANumberError: env.roughness is undefined: env.roughness * 2 (Line 4)
```

7.4.3 Semantics

The built-in vector data types of shade.js come without semantics such as *normal* or *color* and we consciously omit a mechanism to annotate these semantics. However, some shading languages (e.g. OSL) define colors and normals as basic data types that do not cast to each other implicitly. Also, knowing the semantics of input parameters is useful, for instance, to populate user interfaces in the application.

Instead, we annotate the arguments of the predefined radiance closures with their semantics. Based on this information, the compiler can derive the semantics of the input parameters by doing a data flow analysis along the reversed control flow graph. The analysis supports generic vectors, colors, and normals. Since a parameter could be used as color *and* normal (e.g. for debugging) the analysis detects this ambiguity and delivers both semantics as the result.

7.4.4 Optimizations

When GLSL is the target language, the shade.js system generates a GLSL fragment shader. However, some computations only change per vertex or are even uniform for a whole batch of geometry. Typically, shader authors manually move per vertex calculations into the vertex shader and uniform expressions to the application in order to improve performance. The shade.js compiler automatically performs these analyses and transformations to move these calculations to where they are most efficiently executed.

Uniform Expressions The idea of extracting uniform expressions to compute them only per-geometry patch was first introduced in [HL90]. In shade.js, the compiler does not only identify single uniform expressions in the shader program, but propagates these expressions to find more complex uniform expressions within the control flow. To avoid the extraction of expressions that are essentially free on graphics hardware (e.g. swizzling of components) we additionally introduced a cost estimation. The same estimation can be used to extract only the most expensive expressions when we are at risk of exceeding the maximum number of uniform values supported by the hardware.

A uniform expression gets replaced by a new uniform parameter whose value is computed by the application. In case of the WebGL-based renderer that is written in JavaScript, the extracted expression can be evaluated in the application as is. If the compiler extracts interdependent expressions it provides a method to set the original uniforms, transparently handling the update of all generated uniforms.

Coordinate Space Transformations Expressions which only depend on vertex attributes can be moved to the vertex shader stage. Additionally, operations depending linearly on these values can also be factored out into the vertex shader. A major use case for this is the transformation of vertex attributes into different coordinate spaces. Our type system stores information on requested coordinate spaces for vectors and points. Some parameters of the radiance closures (e.g. normals) are requested in a predefined space; The requested space depends on the renderer. Additionally, the material author can request transformations explicitly via provided functions, e.g.:

```
var P = Space.transformPosition(Space.VIEW, env.pos);
var N = Space.transformDirection(Space.WORLD, env.normal);
```

Based on these information we can propagate requested spaces along the reversed control flow. If a non-linear operation takes place, the compiler injects the transformation into the fragment shader. If up to the entry point of the shader only linear operations get applied, it is safe to move the transformation into the vertex shader. Since transformations of directions are linear operations, we can propagate them through any vector addition and scalar multiplication. This is sufficient, for instance, to shift all coordinate transformations involved in normal mapping [COM98] to the vertex shader stage.

7.4.5 Code Generation

In this section, we describe how shade.js code is converted to GLSL code for forward shading and deferred shading, as well as to Open Shading Language code.

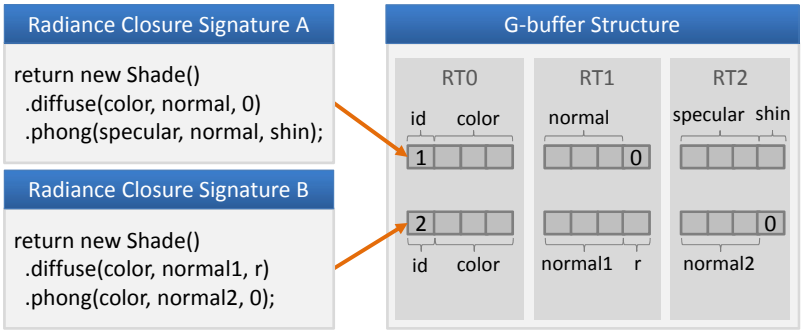


Figure 7.4: Example on how different combinations of radiance closure invocations are stored inside the G-buffers. An id is assigned for each unique combination of radiance closure invocations and used in the light pass to correctly interpret the parameters.

GLSL Due to the design of the shade.js data types, the code generation for GLSL is fairly straight forward. However, we have to explicitly add the code for lighting calculations. This code comes from the renderer and is written in shade.js as well. It is integrated at the beginning of the compilation process, thus the specialization and optimizations are not only applied to the material description provided by the user but also to the renderer-specific light calculations.

Each exit point of the shade routine that results in a (linear combination of) radiance closure is replaced with a function call that internally iterates over all available light sources. This function is specifically generated to reverse the looping: Each light is evaluated only once before the light material interaction described by the radiance closures is applied.

From a shade.js material description we can generate code for a forward renderer as well as for a deferred renderer [ST90]. As a result, switching between and combining both rendering techniques is simple and does not require having multiple shader versions for the same material. This is very useful, for instance, if one wants to solve the common limitations of deferred shading with respect to semi-transparent objects. Since rendering semi-transparent objects is not easily possible with deferred shading, these objects are typically rendered in a separate pass using forward shading. To our knowledge, our system is the only one that supports custom materials to be rendered with both approaches.

For the deferred shading pipeline we generate two types of shaders: An *object shader* that writes the parameters into the geometry buffers (G-buffers), and a *light pass shader* that reads the properties of the G-buffers and performs the lighting. The object shader includes all code of the original shade.js program, but instead of invoking the radiance closures, it writes their parameters into the G-buffers. To handle different combinations of radiance closures (which we call radiance closure signatures) across all object shaders using one global G-buffer structure, we assign a unique id to each radiance closure signature and include this in a G-buffer in addition to all other parameters (see Figure 7.4). This id is then read by the light pass shader in order to properly extract all parameters and perform the lighting with the selected radiance closures.

Since we know the semantic of the radiance closure's arguments, we can easily apply compression techniques to the arguments, e.g. quantize normals, in order to automatically minimize the number of required G-buffers. The light pass shader is generated from the list of all used radiance closures and reuses the light integrators of the forward shading

algorithm to implement the lighting.

Open Shading Language Our third compiler back end generates Open Shading Language code from the shade.js material. Here, we can omit the step to inject code for the lighting, because in OSL the light integration is not implemented in the shader but in the renderer. We map the shade.js radiance closures to the closures defined for the Cycles renderer provided by Blender. In addition to the common specialization we must compute the semantics of the generic vectors, because OSL requires using different types for colors and normals.

7.5 System Experience

The compiler framework itself is written entirely in JavaScript as well. This allowed direct integration of shade.js into xml3d.js (see Section 6.2.1). Additionally, we implemented an HTML5 material editor that allows for interactive authoring of shade.js material (Figure 7.6). Outside the browser, we can use the compiler framework to generate OSL shaders via the command line.

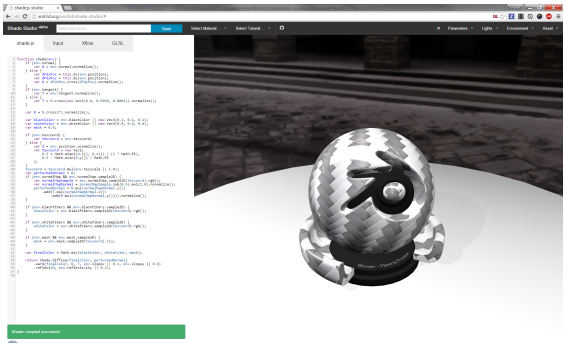


Figure 7.6: Screenshot of an interactive editor for shade.js running in the browser. The user can change source code and input parameters interactively and receives immediate feedback from the system, including warnings and error messages. Additionally, one can review the resulting GLSL shaders.

Figure 7.5 shows a scene we created to evaluate and demonstrate the capabilities of shade.js. The scene consists of 9 non-trivial shaders, ported from RSL, OSL, and other sources. This includes shaders with procedural textures, fractals, layered BRDFs, reflections, and normal mapping. We generate shaders for forward and deferred rasterization in WebGL, as well as OSL shaders for ray tracing and global illumination in Blender's Cycle renderer. As the figure shows, we achieve conceptually equal materials for all four rendering techniques. The resulting images differ only with respect to the lighting as every renderer takes a different approach to approximate the lighting equation.

We measured the performance of shade.js in terms of the generated GLSL shaders and the impact of optimizations. To do this, we translated nine GLSL shaders to shade.js and compared the results. Since we designed our data types to map very well to GLSL data types, there was no significant difference between the original shaders and the shaders generated by our compiler. For instance, translating the "Beating Circles"[6] shader from Shadertoy, results in 65 instructions versus 64 instructions for the original version.

In addition, we implemented the predefined Phong material (urn:xml3d:material:phong) from XML3D. This material is an Ubershader that is specialized by xml3d.js using preprocessor macros. Using the same execution context, our compiler generates GLSL code

[6] `http://www.shadertoy.com/view/4d23Ww`, accessed 18 December 2015.

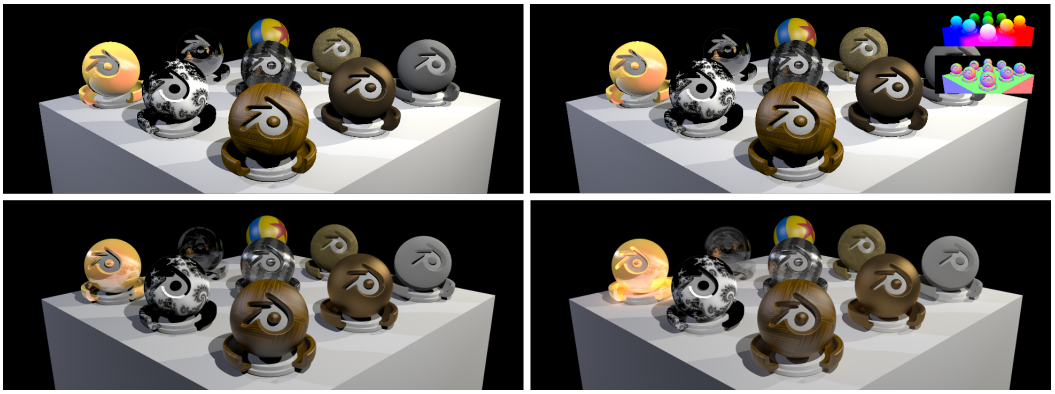


Figure 7.5: A collection of procedural materials written in shade.js and rendered in WebGL using forward shading (upper left), deferred shading (upper right), Blender Cycles ray traced (lower left), and with global illumination (lower right). We achieve conceptually equal materials for all four rendering techniques.

	Tablet fps			Laptop fps			Desktop fps			Instructions		
Uniform Extraction	off	on	Δ	off	on	Δ	off	on	Δ	off	on	Δ
Glowing dots shader	6.9	17.0	2.4x	27.4	53.2	1.9x	222	366	1.6x	83	24	29%
Beating Circles Shader	15.6	19.7	1.3x	37.1	48.5	1.3x	270	359	1.3x	37	31	84%
BPM shader	12.4	20.7	1.7x	34.2	56.9	1.7x	271	371	1.4x	69	27	39%

Table 7.1: Performance gain achieved by uniform expression extraction. We tested uniform extraction on a Tablet (Qualcomm Adreno 330), a Laptop (Nvidia NVS 5400M), and a Desktop computer (Nvidia GeForce GTX 660 Ti). All shaders were applied in our WebGL-based XML3D implementation, running in Google Chrome in Full HD resolution. In order to increase the shader workload and reduce other overhead, each shader was drawn 20 times over the whole screen. The instruction count was determined by Nvidia's cgc compiler. All 3 example shader include procedural animations based on a uniform time value. The *Glowing dots* shader was created by us, while *Beating Circles* and *BMP* are shaders from Shadertoy[7] translated to shade.js. The extracted uniform expressions were executed exactly once per frame in JavaScript, which had no measurable impact on the performance.

that results in 52 instructions whereas the specialized shader of xml3d.js results in 58 instructions with negligible differences in rendering performance.

In cases where the compiler can extract uniform expressions out of the fragment shader we are able to measure significant speed-ups in rendering performance. Table 7.1 shows that the performance impact is especially apparent for platforms with less powerful GPUs such as laptops and tablets.

The interface to the lighting system based on radiance closures is less flexible when compared to traditional approaches such as RSL [HL90], which allows for actively shooting rays and for implementing custom shading models. On the other hand, the higher abstraction level of radiance closures gives the renderer more freedom of choice in the approximations and light integration algorithms to solve the rendering equation, e.g. using importance sampling. In particular for GPU-based renderers with their limited access to scene information, we found it useful to be able to hide all the necessary "tricks" be-

hind the predefined shading models. For instance, it is possible to apply screen-space ambient occlusion (SSAO) as part of the *diffuse* radiance closure. For the *reflection* closure, the renderer could generate dynamic reflection maps. The computational effort and the techniques used to approximate these effects do not require changes in the material description. Even more importantly, such approximations can be turned on and off, either by user request or as a result of insufficient hardware or software capabilities, or based on performance criteria, again without rewriting existing shaders. Overall, the interface based on radiance closures greatly increases the portability of our material shaders.

7.5.1 Limitations

When creating adaptive shaders, there is a natural trade-off that type errors can only be determined once all input types are known. In contrast, specialized languages with static types and fixed input signature allow type errors to be detected independently of runtime input values. However, if an application offers adaptivity by supporting multiple input signatures e.g. via preprocessors, it runs into the same issues: One approach is to generate and validate all possible permutations in a preprocess (which would be possible for shade.js shaders as well). Otherwise, the correctness of a permutation is only guaranteed at runtime and it is left to the application to analyze and report errors that might come with a specific input signature.

Specialization in general can result in a combinatorial increase in the number of generated shaders. shade.js generates one shader per unique signature given by name, type and source of input parameters. Using a signature cache, the system keeps the number of compilation passes to a minimum. However, many different signatures can potentially increase the number of generated shaders. It would be possible to specify parameters to be excluded from the specialization. In that case the compiler would introduce an additional uniform variable that is automatically set within the shader's setter method and which is dynamically tested in the shader.

7.6 Conclusion

In this chapter we presented shade.js, a system for writing adaptive material descriptions that are portable between different scenes, hardware architectures, and rendering approaches.

A key element of shade.js is its ability to describe all aspects of materials in a single place and the ability to automatically adapt and specialize the shader to the specific execution environment being used. The use of radiance closures allows supporting procedural material descriptions even with global lighting algorithms.

Directly porting a significant number of non-trivial GPU and other shaders showed that shade.js can achieve essentially the same performance despite the much higher level of abstraction due to the compiler-based analysis. Exploiting some of the high-level features of shade.js allows for improving the performance even further via optimizations made possible by code analyses and transformations in the shade.js compiler.

Although any implicitly typed programming language would be suitable to implement the shade.js concepts, using JavaScript for shading, compilation, and as the host language, allows it to be used directly as part of the XML3D architecture. The ability to adapt

to heterogeneous scene data and GPU capabilities is of utmost importance in the Web context, with its high variability of scenes and devices. The provided optimizations proved particularly valuable for mobile devices (see Table 7.1).

The approach presented here is not restricted to materials. In Section 11.2 we will discuss ideas applying shade.js concepts to programmable light descriptions, portable specifications of atmospheric and volume effects, image-space effects like post-processing and tone-mapping, and others.

8 Data Delivery

With the advent of rich multimedia on the Web, the demand for efficient transmission of large sets of binary data is increasing. This includes HTML5 video and audio streaming, data to do audio processing using the Web Audio API [W3C12b], and 3D related data to be used by the WebGL API [Khr09]. Images, audio and video have respective HTML tags and accompanying data formats that transparently handle binary transmission and decompression. 3D data, on the other hand, has to be handled explicitly by the client application. However, unlike video, audio, or images, 3D data is structured and not homogeneous. Thus, it is more challenging to establish a common 3D transmission format for the web.

Beside binary XHR [W3C09f] based on ArrayBuffers, which is commonly used but insufficient for many use cases, developers found creative solutions to transmit 3D data to the browser. This includes exploitation of fast native parsing of JSON [Cro06] or UTF-8 [UA06], and exploiting 2D images to transfer 3D data [BJFS12]. However, all approaches that can be found in scientific literature or "in the wild" address only a subset of the challenges that need to be solved for a general, common, and efficient transmission of 3D data. The existing approaches do not provide a general binary format for *all* kinds of 3D data including meshes, textures, animations, and materials. Existing formats are domain-specific and fixed on a certain set of input data and thus too specific to handle other types of data.

Consequently, we propose *Blast*, a general container format for structured binary transmission on the Web that can be used for all types of 3D scene data and beyond. Instead of defining a fixed set of encodings and compression algorithms Blast exploits the *Code on Demand* paradigm to provide a simple yet powerful encoder-agnostic basis to leverage existing domain-specific compression techniques. Since streaming is of primary importance for a good user experience, Blast is designed on the basis of self-contained chunks to enable JavaScript clients to utilize Web Workers for parallel decoding and to provide early feedback to the user. This part of the thesis has been previously published in [SSS14] and is joint work with Jan Sutter.

8.1 Factors

As outlined in [SS13], every binary transmission format for the Web has to be designed to handle network characteristics such as bandwidth and latency and facilitate client applications in providing a good user experience.

In order to design an efficient 3D transmission format for the web, one has to consider following decisive factors:

Network bandwidth Although steadily increasing, network bandwidth can still be the bottleneck for the transmission of large amounts of data in particular in mobile

contexts. Strategies to overcome bandwidth issues include efficient representation, quantization, and lossless or lossy compression.

Network latency Sending a request to the server involves communication overhead which manifests itself in network latency. In fast networks, network latency can easily outweigh transmission time. Strategies to reduce network latency include use of structured or interleaved data to reduce the number of requests.

Decoding speed The data being transmitted needs to be decoded on the client in order to be used with Web APIs. Decoding time needs to be in balance with the other factors. Additional aspects that influence the decoding speed includes native decoding vs. decoding in JavaScript and whether the decoding process can be performed in parallel using technologies such as threads (Web Workers [W3C15b]) or data-parallel execution, e.g. through RiverTrail [HHSS12] or WebCL [Khr14b]. Obviously, also encoding speed is a factor. However, since in most use case the encoded data can be pre-calculated and cached, we consider this aspect as less decisive.

User experience Internet users are accustomed to responsive web applications and are sensitive to delays. Even 10 seconds of wait time result in a serious degradation of the user experience [Nie93]. It is, thus, crucial to provide users feedback on the progress of the transmission and the possibility to continuously interact with the scene while transmission may still be in progress. This is the behaviour Internet users are accustomed to from classical Web applications: Waiting times are accepted as long as feedback is provided and interaction can start before, for instance, all images are loaded.

User feedback can be achieved on application level (e.g. the application could display bounding boxes that get progressively replaced), but it can also be part of the transmission process (progressive loading of large data sets in usable pieces). To facilitate this, the format has to be streamable and must provide early access to usable chunks that can be handled in parallel to the main thread of the application.

Bandwidth and latency are the two major properties that depend on the current network connections (from intranet to mobile data services). Decoding speed largely depends on the used device (desktop to cell phone). The “soft factor” user experience is important in all of these settings.

8.2 Requirements

We derive the following set of requirements for binary transmissions as a consequence of these factors:

Structured To reduce network latency, a transmission format needs to be able to transport structured data. Requesting each resource of a complex structure individually can induce high latency. A binary format therefore has to be able to send multiple resources in a single request similar to XML. The client, thus, needs the ability to address individual resources inside the transmission, e.g. using URI fragments as identifiers.

Efficient Efficient handling in JavaScript is important and a reason why JSON has become so prominent as transmission format. Handling binary data in JavaScript is only efficiently possible if the format facilitates the use of Typed Arrays [Khr11]. They are required as input for APIs that support data-parallelism (e.g. WebGL, WebCL, or River Trail), but also the first choice for data processing in JavaScript [KSJ*12, Moz13]. Additionally, Typed Array data can be transfered with zero copying overhead between a Web Worker and the main thread.

Schemaless The format should neither impose a special schema on the data nor should it only support a specific set of attributes, e.g. certain 3D scene formats. Existing domain-specific transmission formats are not only unstructured but impose a particular signature on the input data. Even though, mesh data accounts for the majority of a 3D scene's data and are, thus, the most important part a binary transmission format has to handle, the contribution of animations, textures, and materials to the overall size of a scene should not be underestimated. A binary format for 3D data therefore has to be able to transport this data equally well. Generic parameters, as required for recent graphics APIs in general and in particular for XML3D's dataflow processing and programmable material concept implies a generic approach that no previous binary format provides.

Compression Data encoding is important to minimize the size of the transported data and to overcome bandwidth limitations. However, the encodings and compression algorithms must not be "baked" into the format. To obtain the best possible compression ratios with minimal amount of processing time, the algorithm has to be specifically designed for the data and the specific needs of an application. For a 3D transfer format this implies that encoders must be definable on a per-data basis.

Streamable Streaming is "a technique for transferring data, such that it can be processed as a steady and continous stream" [Pag08]. Designing the delivery format in a way we are able to stream the content to the client is a direct consequence of the requirement to send multiple resources at once and the necessity to provide early and continues feedback to support a good user experience. Without support for streaming, a single request containing all resources of a 3D scene would be required to be received entirely before data processing can start, ruining the user experience.

REST Compatible Even though not a direct consequence of the key factors we require a transmission format for 3D data to be compatible with web services following the REST paradigm [Fie00]. Delivering 3D content to the browser is in the transition from a using static files to more sophisticated service oriented approaches [DSR*13, Arn11]. REST APIs do not explicitly indicate the transmission format as part of the URL but use content-negotiation to determine the best suitable delivery format. To enable content-negotiation it is important that all possible formats use the same syntax to address subordinate resources based on fragment identifiers. The binary transmission format should support the established fragment identifier syntax of URIs to identify a resource subordinate to another.[1]

[1] In XML and HTML documents fragments usually correspond to a resource with an `id` attribute with the same value.

Based on these requirements we propose *Blast*, a streamable, encoder-agnostic container format for binary data transmissions on the web.

8.3 Blast

As a container format Blast supports both, a single binary resource per request and sending multiple resources in a structured transmission similar to XML. Moreover, Blast is streamable. Since streaming is currently not supported by XHR [W3C09f], Blast is designed to work with the upcoming Streams API [W3C14e], which allows for XHR based binary streaming. Additionally, Blast can be streamed using Web Sockets [FM11].

In fulfillment of the stated requirements for a binary transmission format Blast is build around the following key concepts:

Generic Approach Even though designed specifically with 3D data in mind, Blast does not impose any restriction on the transported data. Blast is typeless by design (see Section 8.3.1).

Code On Demand To support custom encoders on a per data basis Blast uses a code on demand [FPV98] approach to supply clients with the necessary decoding procedure. (see Section 8.3.2).

Identification and Addressability Structured transmission requires means to address resources within a transmission. To be compatible with XML Blast uses fragment identifiers based on a path description language to address resources in a transmission (see Section 8.3.3).

Chunked Based Streaming Binary streaming in Blast is achieved using self-contained chunks that can be independently processed in parallel (see Section 8.3.5).

Transparent Decoding Blast serves as a transparent layer for the transmission and decoding such that client and server can simply exchange their data structures disregarding any transmission details (see Section 8.3.6).

8.3.1 Generic Container

Blast is a generic container format and can be used for the binary transmission of any key-value data structure containing any types of data. Blast is type-agnostic and only handles byte data. The byte data generated by an encoder is an atomic value in a Blast transmission and will be passed as-is to the respective decoder. It can be a flat key-value structure describing a single mesh, a collection of such mesh definitions, a single vertex buffer of a mesh, or a complex hierarchical key-value structure. Every Blast transmission contains an arbitrary number of these values.

Each value addressable by the Blast addressing mechanism (see Section 8.3.3) can be encoded individually. Blast transports all necessary information for the reconstruction of the original key-value data structure.

Since Blast transmissions do only transport byte data the endianness of the encoded data is the responsibility of the sender and the used encoders that have to handle multi-byte

data properly. Endianness of a Blast transmission should be specified during content-negotiation as a parameter of the media-type, e.g. *application/blast;endianness=little*.

8.3.2 Custom Encoders – Code on Demand

Code on demand is a term that describes technologies that transfer source code to be executed on a remote end. All websites that use JavaScript use the code on demand idiom whenever they send JavaScript to the client.

Custom encoders are an important requirement for a general container format. Encoders in Blast can operate on any part of the key-value structure sent. There is only one requirement: they have to produce binary data that allow for a decoding procedure to reconstruct the original data. This encoded binary data is then handled as a value and the information where this value was located in the overall key-value structure is specified using a path expression (see Section 8.3.3). Additionally, for each value a unique identification of the corresponding decoding procedure has to be provided using a URL. We will show in Section 8.4 that for 3D data containing meshes as values the overhead induced by these URLs is negligible.

The client can provide an implementation-specific decoder utilizing technologies such as WebCL, RiverTrail, or native functionality and therefore use the URL for identification purposes only. For instance, an XML3D implementation can use the dataflow processing concept to describe the decoder and map the decoding to available hardware. In cases the client does not have an implementation available it can use the URL to request a default implementation on demand. This allows the use of any encoder without requiring the client to implement a decoding procedure (see Section 8.3.6).

The implementation requested on demand has to conform to a certain interface to be usable by the client. It has to expose a function `decode` taking two parameters, the encoded bytes and the endianness of the transmission. For this approach to function properly, it is the responsibility of the content provider to ensure that a suitable implementation is available at that specific URL. For our context, implementations are typically implemented in JavaScript. However, content negotiation mechanisms can be used to request implementations in other languages.

In contrast to the code on demand that web applications use, the approach in Blast may result in the execution of JavaScript from domains unrelated to the current web application. This imposes a potential security threat, even though JavaScript is already sand-boxed in the browser. Potentially malicious code can harm the client application by manipulating or reading sensible information from the DOM or flooding the console. To mitigate this potential we propose to exploit the secure execution context of Web Workers [W3C15b]. These worker threads, already proposed for parallel decoding, have no access to any sensible part of a web application and communication with the main thread is limited to a single channel, the message queue. Running worker threads can be monitored and watch dogs can be deployed to terminate them if necessary.

8.3.3 Structured Transmission

Structured transmission requires the client to be able to address individual resources inside the transmission for further processing. XML uses fragment identifiers. Other formats

require the client to explicitly know the enclosing structure to extract individual resources as they do not provide a standardized addressing mechanism.

Blast transmission are always structured. Hence, to be able to describe the structure of a transmission in Blast, we developed *JPath*, a query language that allows identifying the original position of the encoded value inside the original key-value data structure. JPath expressions only define the structure between values, the structure inside the values is in the responsibility of the encoder and decoder. The differentiation between a single resource and a collection of independent resources is determined by the server by generating the necessary fragment identifiers on the URLs.

JPath expressions are slash delimited property paths, or key paths respectively. JPath is inspired by JSONPath [Goe07] and JSONPointer [BZN13] and borrows its syntax from the latter, which is compatible with the XML fragment identifier syntax.

Consider the following key-value map representing an asset in JavaScript:[2]

```
{
  rootNode: {
    children: [{
      name: "Sample Node",
      mesh: 4
    }]
  },
  meshes: [... { // an array of meshes
    name: "firstMesh"
    position: new Float32Array([...]),
    normal: new Float32Array([...]),
    texcoord: new Float32Array([...]),
    index: new Uint16Array([...]),
  } ...]
  // ... possible other properties
}
```

The JPath expression *meshes* specifies the entire array of meshes. The path

```
/rootNode/children/0/name
```

points to the string `"Sample Node"`. JPath allows for both, property notation and bracket notation to specify array elements, for instance the entire first mesh of the array can be specified using either `/meshes/0` or `/meshes/[0]`. The property notation is used for compatibility with the XML fragment identifier syntax, while the bracket notation is used to explicitly indicate that the type of the parent element is an array. This is important for the transparent decoding that Blast provides (see Section 8.3.6).

[2] This example is a JSON representation of Assimp's C data structure: `http://assimp.sourceforge.net/`, accessed 18 January 2015.

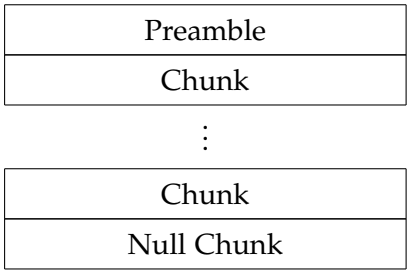


Figure 8.1: General structure of a Blast stream.

8.3.4 Structure of a Blast Transmission

Every Blast transmission consist of two major building blocks, the preamble and a sequence of chunks terminated by the null chunk. The entire structure of Blast, as depicted in Figure 8.1, is designed to be streamable, to facilitate parallel decoding on the client-side, and to reduce management overhead on both ends.

The preamble, shown in Figure 8.2, is the very first part of every Blast transmission and carries information for all following chunks. Its first 4 bytes (ASCII BLST) are for identification purposes and detection of the endianness of the stream. The following 4 bytes represent an unsigned integer that indicates the overall size of the preamble. The next 2 bytes each interpreted as an unsigned byte specify major and minor version number of the Blast stream format. The last part is a NUL terminated ASCII string for the URL that identifies the decoding procedure necessary to reconstruct the header information of each of the following chunks. Because this URL can be arbitrary long in theory the size of the preamble is not limited which is why its overall size has to be specified.

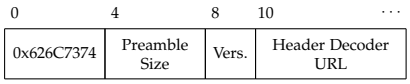


Figure 8.2: The preamble of a Blast stream. Size indications are given in bytes.

8.3.5 Streaming – Chunked Transmission

Blast can be used in three ways. First, sending a single resource per request. Second, sending multiple resources in a structured way within a single request to reduce the overall latency. It can transfer multiple binary resources ranging from a single mesh to all data of a scene in a single request. More importantly, Blast can be streamed. This enables a client to request all resources of a scene in a single request while still providing early user feedback and interaction possibilities. This flexibility is achieved using a chunk based approach (see Figure 8.3). Chunked container formats are widely used for multimedia data [MV94, Mat05, Ele85].

Chunks in Blast can transport any number of values. As long as XHR does not support streaming, a single chunk can be used to transfer a single resource or multiple resources. Web Socket connections, on the other hand, can use multiple chunks for streaming. The number of values, i.e. the size, of each chunk can be determined by the server depending on connection properties and the structure of the data.

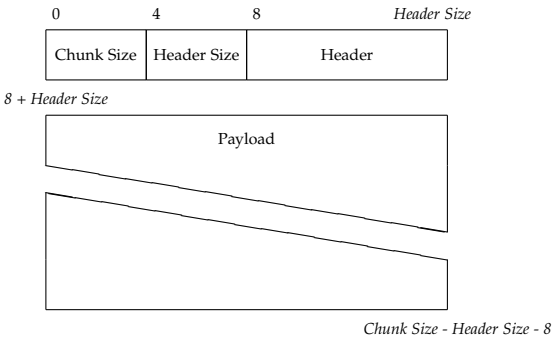


Figure 8.3: The structure of a Blast chunk. Size indications are given in bytes.

Instead of using a single description for the structure information of the entire data, Blast uses self-contained chunks. While a single description for the structure may allow for more efficient encoding of the structure information it imposes a specific order on the data transported and, moreover, makes it necessary that all data is already available to define the structure.

In Blast each chunk consists of two parts: a header and payload. Chunks in Blast are always self-contained and independently processable. They never transport only parts of the data necessary for reconstruction nor do they depend on other chunks in a stream. All information necessary is contained inside the chunk itself or was received in the preamble.

This does not only enable parallel decoding using Web Workers, but allows the receiver to discard already processed chunks immediately. Parallel decoding is important to not block the main thread during CPU intensive decoding. More importantly, autonomous chunks reduce management overhead, enable the remote end to send chunks as soon as possible or in any order the client requests, and to cache and reorder them during later requests.

Each chunk starts with 4 bytes to be interpreted as an unsigned integer to indicate its overall size. This allows the client to reserve the necessary memory for the chunk at once avoiding costly reallocations during processing. Following are 4 bytes to specify the size of the chunk header. The required header contains information about the structure of the data transported in the payload. It is an array of key-value maps, one for each value transported by this chunk. This header is encoded and has to be decoded using the procedure indicated in the preamble of the stream. The remainder of the chunk is its transported payload.

Each chunk header contains structure information about the values encoded in the payload. Since Blast is designed to allow every value to be encoded in a user definable way every entry inside the header contains a unique identification of the necessary decoding procedure to reconstruct the original data. Each entry additionally contains a JPath expression that uniquely identifies the value's original position in the key-value data structure sent. The following values are transported in a header entry:

offset The offset in bytes into the chunk's payload at which this value is located.

size The size in bytes of the encoded data.

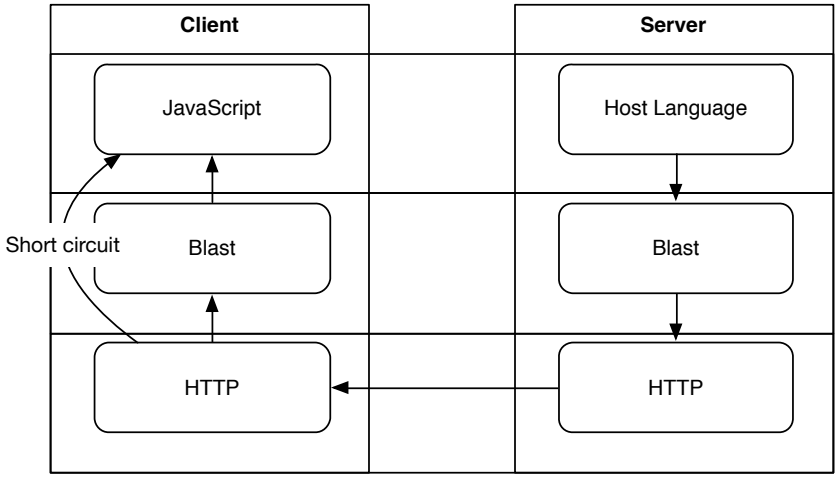


Figure 8.4: Blast serves as a transparent decoding and transmission layer while still providing the client ways to short circuit.

path The JPath expression that uniquely identifies the value (see Section 8.3.3).

metadata An optional key-value map that can hold any information of any size and can be used to transport additional information about a value. This meta information can, for instance, be used to specify additional semantics or application specific type information if necessary.

decoding specification The decoding specification indicates how the bytes specified by offset and size have to be decoded. This specification is always a URL that the client can use to uniquely identify the decoding procedure or to request an implementation on demand.

A special chunk in every Blast stream is the last one, the null chunk. Since every Blast transmission can contain an arbitrary and unspecified number of chunks this null chunk is used to signalize the end of the stream. The end chunk has no payload and no header information: It is just the 4 bytes that indicates a size of zero.

8.3.6 Transparent Decoding

The dichotomy between a resource's internal structure and the format in which the resource is transfered is an important aspect of the Web. Resources have a structure, e.g. meshes have buffers, materials have parameters and shader code. Content-negotiation defines the process in which a client and a server determine the used transmission format. This format, however, does not necessarily reflect that inherent structure of the data sent. A client's main interest is the data not the format in which it is transfered. For the client the transmission format is only of importance because it has to deserialize the actual data. One reason for JSON's prominence is that deserialization is natively supported in JavaScript and that encoding and compression using the HTTP intrinsic gzip compressor [FGM*99] is handled transparently.

As a consequence of the code on demand approach together with the unique identification and structure specification that a JPath identifier provides, Blast can be used as a transparent layer for data transmission (see Figure 8.4). This layer can parse incoming chunks, extract all transported values, download the necessary decoding functionality and reconstruct the sent object using the value's JPath expressions. The precise specification of a JPath expression enables the reconstruction of the complete structure of the original key-value map. The differentiation in JPath regarding array indices using bracket notation allows the receiver to recursively create non existing parent values along the path. As soon as the final chunk is received the Blast layer can inform the client that the object is completely reconstructed. Moreover, because decoders are specified uniquely using URLs the client can always identify a decoder and short-circuit the transparent decoding layer to use its own implementation.

The transparent decoding layer provided by Blast enables the client application and the server to disregard transmission details of the exchanged data structure. Moreover, it is now possible for the server to choose the encoding based on connection properties and utilize encoders that are unknown to the client. Intensive compression can outweigh transmission time on high bandwidth connections and increase latency while mobile connections may require such compression because of bandwidth and traffic limitations. Without the code on demand approach choosing an encoder based on these properties is only possible if the client has the required decoding procedure implemented.

8.4 Results

For the evaluation and benchmarking of Blast we integrated a server-side implementation into XML3DRepo [DSR*13] and a client-side implementation into xml3d.js. Furthermore, we integrated two existing encoders, OpenCTM [Gee09] and Open3DGC [Mam13].

The flexibility of Blast allows requesting scenes in three different ways: individual requests per resource, aggregated requests of multiple resources, and streaming of *all* resources within a single request.

We conducted benchmarks with respect to the transmission time of these usages simulating a broadband connection with 6 Mbit/s download and 512 Kbit/s upload bandwidth and a 1 Gbit/s local area network using a software bandwidth shaper. The bandwidths of the broadband connection are chosen to be conservative in regard to the average available bandwidth of todays connections.

The benchmark results for the transmission of two large scenes are shown are shown in Table 8.1: The "Atrium Sponza Palace" scene from Crytek[3] and the "London City" from Vertex Modelling[4] (see Figure 8.5).

The benchmarks show that the influence of the required number of requests is less significant for a 6Mbit/s connection: the limiting factor is the bandwidth itself, thus, the encoding becomes important. In contrast to a high bandwidth connection the reduction in the overall transmission time by aggregating and streaming resources is less significant. However, being able to stream all resources is important: it provides full control over the order of transmitted resources.

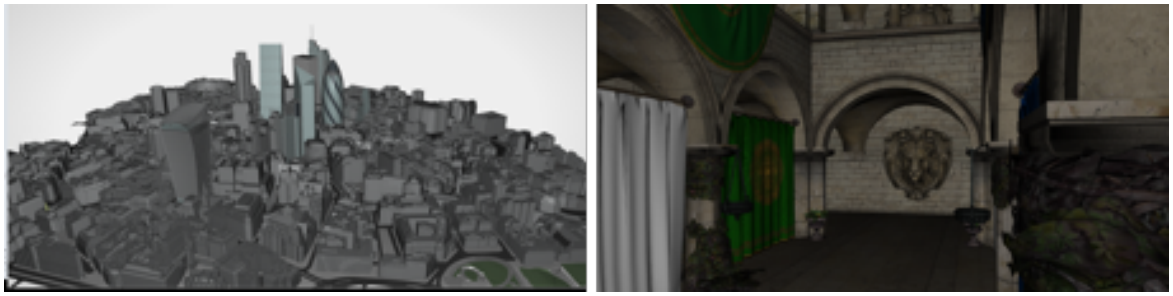


Figure 8.5: Left: "London City" (Model courtesy of Vertex Modelling). It consists of 4067k triangular faces, has no textures, and an overall size of 279 MiB. Right: The "Atrium Sponza Palace" (Model courtesy of Crytec). It consists of 394 meshes, 36 textures, and 279k triangular faces with an overall size of 81 MiB. Both rendered in XML3D and streamed with Blast in a single request using the Open3DGC mesh encoder. On a 6 Mbit/s connection the complete transmission took 31.7 seconds ("London City") and 89.5 seconds ("Atrium Sponza Palace") while the initial meshes became available already after ∼60ms.

[3] `http://www.crytek.com/cryengine/cryengine3/downloads`, accessed 18 December 2015.
[4] `http://vertexmodelling.co.uk/`, accessed 18 December 2015.

	Atrium Sponza Palace (81 MiB)				London City (279 MiB)			
			Transmission Time [s]				Transmission Time [s]	
	#Req.	Size [MiB]	6 Mbit/s	1 Gbit/s	#Req.	Size [MiB]	6 Mbit/s	1 Gbit/s
Individual								
JSON	418	72.2	104.7 (0.2, 84.6)	3.8 (0.02, 0.89)	328	136.2	194.9 (0.14)	4.4 (0.08)
OpenCTM		67.3	97.8 (0.04, 83.6)	3.7 (0.06, 0.84)		59.7	86.8 (0.09)	3.5 (0.04)
Blast (OpenCTM)		67.4	97.9 (0.05, 83.9)	3.7 (0.06, 0.85)		59.7	87.8 (0.10)	3.5 (0.04)
Blast (Open3DGC)		63.2	92.0 (0.02, 82.8)	3.6 (0.04, 0.81)		22.3	34.3 (0.05)	2.9 (0.02)
Aggregated								
XML	103	72.1	102.5 (1.2, 87.3)	1.2 (0.09, 0.93)	73	136.2	193.4 (0.27)	1.8 (0.12)
Blast (OpenCTM)	84	67.3	97.1 (0.9, 83.8)	1.0 (0.07, 0.79)		59.6	86.7 (0.23)	1.5 (0.10)
Blast (Open3DGC)		63.1	91.3 (0.8, 77.4)	1.0 (0.07, 0.74)		22.3	33.1 (0.13)	0.8 (0.09)
Streamed								
Blast (Open3DGC)	1	63.1	89.5 (0.06)	0.6 (0.03)	1	22.3	31.7 (0.06)	0.2 (0.03)

Table 8.1: Benchmark for the transmission of the "Atrium Sponza Palace" (left) and the "London City" (right) regarding a 6 Mbit/s and a 1 Gbit/s connection. All measurements were conducted 10 times and the median values are given. Except for OpenCTM transmissions, which already deploys an LZMA compression, all formats were further gzip compressed. Transmission times in parentheses indicates the times until the first mesh was received. For the "Atrium Sponza Palace", where textures contribute the major part of the size, two numbers are given: the first indicates the minimum time, the second the maximum time until the first mesh was received. The used mesh encoder for the Blast transmissions is given in parentheses. For the aggregated resources scenario five meshes are sent within a single request. Since XML does not support binary data textures are still requested separately. Concurrent requests were limited to two following the HTTP/1.1 standard [FGM*99]. In general, streaming Blast using Open3DGC results in the best performance for all benchmarked settings.

The time for the first mesh in the "Atrium Sponza Palace" scene clearly varies for all transmission types, except for streaming. This variation in time is a result of the browser's internal order and scheduling of the requests. There is no guarantee, for instance that meshes are always received before the textures without particular application logic. Blast, on the other hand, allows the server to control the order of the resources.

In contrast to the low bandwidth connection, the benchmark simulating a 1 Gbit/s connection clearly shows the impact that the number of requests have on the overall transmission time. Aggregation and streaming result in a significant reduction of the overall transmission time.

Because of its flexible design, Blast can be used in all three scenarios. Its encoder-agnostic design enables the use of existing domain-specific encoders to reduce the size of the transmission. The overhead of Blast for the transmission of individual resources per request is negligible in comparison to domain-specific formats such as OpenCTM. However, while these format lack the ability to transfer multiple resource in a single request, Blast provides this functionality still taking advantage of their compression abilities. The reduction in the number of required requests by aggregating resources considerably improves the transmission time for high bandwidth connections.

Because of the inability to stream binary data using XHR, resource aggregation is of great importance. However, using Web Sockets to stream Blast, further reduces the required request to a single one still providing the same early feedback and user experience as individual requests for each resource. In general, streaming Blast in combination with Open3DGC achieved the best performance in all tested settings. Furthermore, the server has full control over the order of the resources, which allows for a controlled transmission. This can be used to send meshes depending on the client's view into the scene or to interleave meshes and their respective textures to further increase the early visual quality of the rendering.

8.5 Related Work

The list of existing approaches regarding binary data transmission is huge. Many different formats exist in scientific literature and elsewhere. We will discuss three categories of current approaches, why they only address a subset of the stated requirements, and how they differ from Blast, yet are not mutually exclusive.

8.5.1 Document-Based Approaches

Textual transmission formats such as JSON [Cro06] and XML [W3C08] are widely used. Both are not bound to any set of predefined attributes and allow for arbitrary structured content. As textual formats, they do not supports embedding binary data directly, but only using *base64* [Jos06] encoded binary strings. The base64 encoding however increases the size of the original data and increases decoding time on the client.

JSON and XML both have binary representations. One very common binary JSON format is BSON [10G13]. Developed as the main storage format for mongoDB, BSON is driven by the design requirement for fast traversal. As a result, it treats arrays and documents equally and encodes array indices explicitly as properties. BSON provides data types not natively found in the JSON specification [Cro06], in particular binary blobs.

These blobs are the application's responsibility and can be utilized to transport arbitrary binary data inside a BSON document. Its inherent JSON like structure, however, imposes difficulties for streaming. Additional BSON has no support for custom and data specific encoders.

Several competing standards for binary representations of XML exist, for instance Efficient XML Interchange (EXI) [W3C11] and Fast Infoset (FI) [Tel05]. The latter is also used for binary encoding of X3D documents [Web07]. Binary XML representations utilize dictionary compression for element and attribute names. XMI additionally deploys a deflate compression on the data. FI, on the other hand, allows for custom encodings and domain-specific compression methods similar to Blast. As a result, FI can achieve high compression rates [SS11b]. This custom encoding support, however, uses a special schema for identification that does not support a code on demand approach.

The effort that both of these binary XML representations put in efficient encoding of the structure information, however, increases the complexity of the decoding process even though the size of the structure information is negligible compared to the actual payload in particular for 3D content.[5]

8.5.2 Domain-Specific Approaches

Numerous solutions for streaming 3D meshes exist [LCD13, BK02, LJBA13, Hop96]. Meshes, however, are only one aspect that has to be handled for a generic transmission format and their domain-specific design makes them not applicable to other kinds of 3D data. Moreover, most of these approaches can exploit Blast chunks for transmission and their implementation has to be done on the application level. Their usage is, thus, orthogonal to Blast.

The Open Compressed Triangle Mesh file format (OpenCTM) [Gee09] is a domain-specific encoding and compression format specifically design to handle triangulated 3D mesh data. OpenCTM can achieve high compression rates but cannot be used to transfer multiple meshes and is not applicable to other kinds of 3D data. However, OpenCTM, as shown in our results, can be leveraged in Blast to efficiently compress triangular mesh data.

Encoding geometry information inside images is an approach that allows for a very efficient reconstruction on the GPU. Moreover, image decoding is done natively in the browser, circumventing any JavaScript based decoding for the image data. Sequential Image Geometry (SIG) [BJFS12] extends this approach further. Every vertex attribute is encoded in chunks of 8 bit. These chunks are distributed into sequences of images ordered by relevance to allow simple quantization by omitting less relevant images. Progressive loading can also be achieved if images of a sequence are received in descending order of relevance. One of the issues of SIG is the large number of required requests, i.e. up to four request per vertex attribute if 32-bit precision is required.

Also, receiving the images in the correct order cannot be ensured as Image loading is is subject to the browser's internal resource handling and HTTP 1.1 has no support for prioritizing requests.

[5] According to [BJFS12], 95 percent of an X3D document are used to store vertex and vertex-related data (indices and attribtues) on average.

Similar to OpenCTM, Blast is able to exploit the efficiency of SIG while reducing the necessary requests to a single one and furthermore providing a way to send all images in a specific order to allow for progressive refinement.

8.5.3 JSON and Binary XHR Approaches

A common approach used for instance in X3DOM, three.js, and glTF are unstructured binary XMLHttpRequests using an additional document for structure information. This approach, despite its common usage, increases the number of requests by at least a factor of two as structure and data are separate resources.

glTF is the suggested approach of the Khronos Group to standardize a JSON based 3D scene description format. Typically, a glTF asset is comprised of at least two files: a JSON based description accompanied by referenced binary files for meshes, textures and animations, and text files for shader code. The JSON description contains all information necessary to extract the embedded information inside the binary files.

glTF defines external binary containers for meshes, textures and materials, however, are currently only referenced by URL or embedded as a base64 encoded string. There is no solution towards reducing the overhead in requesting each of these resources in separate request. Compression in glTF is defined using an extension mechanism comparable to those used for functionality in OpenGL and WebGL using a central registry. Clients either support an extension, in which case they must have an implementation of the decompression algorithm, or they cannot read the file. This approach would allow for different encoders and compressors but it requires the client to support all of those extensions. The current draft standard only specifies the Open 3D Graphics Compression (Open3DGC) extension [Mam13].

Compared to Blast, glTF follows a different design rational and intention. As an asset format glTF aims to provide a standardized format for 3D scenes. Blast and glTF can be combined to provide a standardized 3D scene format and an efficient streamable binary transmission format for all external resources.

8.6 Conclusion and Future Work

In this chapter, we decomposed the challenge of transmitting large binary data sets to the client into its decisive factors and, from those, derived requirements on a 3D transmission format. Based on these requirements, we propose Blast, a general container format for binary data transmission for the Web. Though we focused on its ability to transport 3D mesh data it is general enough to be used for any kind of 3D data.

Blast can leverage existing mesh encodings, e.g. Open3DGC and OpenCTM, to achieve high compression rates. The code on demand approach and the transparent decoding enables the server to employ the best possible encoding on the data without burdening the client with decoding and transmission details. Blast's streaming design, focused around chunked based transmission, allows a server to adapt to different features of a network connection by changing the number of values per chunk or the used encoders.

The benchmarks have shown that the flexibility of Blast comes with negligible overhead. Its design does not obsoletes existing efforts but provides a solid basis to overcome their inherent limitations while still taking advantage of their strengths. Blast can reduce

the number of required requests (up to a single request for the whole scene) while still providing the same early feedback compared to a single request per resource approach.

Based on Blast we can now start the evaluation of advanced transmission requirements based on spatial details of a scene including view-dependent and occlusion-dependent requests. In current approaches network characteristics are typically not incorporated into the decision nor the negotiation of the used encoding or how a resource is transfered. The code on demand approach and the flexible chunked encoding of Blast enables the incorporation of heuristics about connection bandwidth, round trip time, and user agent information in the choice of encoding, chunk size, and resource order.

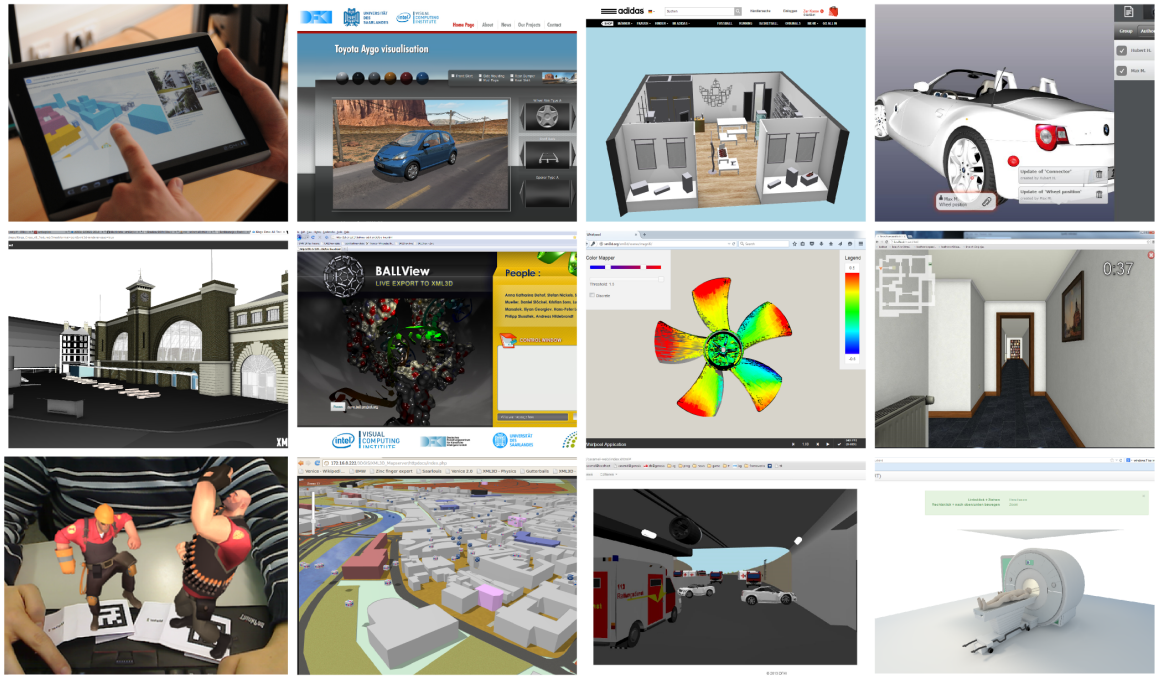


Figure 9.1: A selection of twelve projects using XML3D for visualization (project partners in parenthesis) from left to right and top to bottom: a) Interactive 3D campus map (Saarland University), b) interactive car configuration (RTT), c) collaborative shop planning (Scheer Management & Adidas), d) collaborative CAD review (BMW), e) building information modeling and stakeholder engagement (UCL & Arup), f) visualization and collaborative design of drugs (Intel VCI & Center for Bioinformatics Saar), g) visualization of derivations from CAD data (EU project FIT-MAN, Whirlpool), h) training game for Parkinson patients (EU project VERVE), i) Augmented Reality application using Xflow (Intel VCI), j) geospatial information visualization using OpenStreetMap data (BMWi RealGIS project, Caigos), k) disaster management training application (InSitu project), l) patient education (Saarland University Medical Center)

9 Application Examples

As of today, the XML3D architecture and its implementations have been used in at least 28 research and industry projects at DFKI and Saarland University including Intel Visual Computing Institute. Figure 9.1 shows a selection of applications that have been realized with XML3D in that context.

In this section, we will briefly discuss four XML3D-based applications in more detail: An interactive cultural heritage application running on a terminal in a museum (Section 9.1), an application from the automotive industry (Section 9.2), and two factory planning applications (Section 9.3). For a more comprehensive list of applications and the current

development status, visit the project's web page.[1]

9.1 The Saarlouis Terminal

Figure 9.2: Left: Setup of the Saarlouis terminal in the municipal museum of the city of Saarlouis. The image shows the touch-based interaction console running the XML3D Chromium browser and the 3D television running the VR system. Right: Screenshot of the Saarlouis XML3D scene embedded in the HTML-based interface for navigation and display of additional information.

The *Fortified City of Saarlouis* terminal project allows exploring the 3D city model of two different eras of the historic Saarlouis, a fortress city constructed by France's famous military engineer, Sébastien Le Prestre de Vauban. It was realized in collaboration with the municipal museum of the city of Saarlouis. The objective of the project was to induce awareness of the eventful history of the city and to provide information about the French, but also of the Prussian, era of the city. Additionally, the terminal provides an alternative to the existing physical diorama that shows the French era only, disallows including multimedia data, and is hard to maintain. In the end, the touch terminal is more fun, especially for younger visitors.

The Saarlouis terminal was the first larger project that used XML3D rather than a classical 3D realtime interactive system. The application was mainly developed by a software developer (with little graphics programming background) and a web designer. The 3D model of the city already existed for offline rendering in Cinema4D and was slightly prepared to be suitable for realtime rendering. The final city model consists of more than 6 million polygons in more than 20.000 objects. At the time the Saarlouis terminal project was running, WebGL and the WebGL-based implementation of XML3D were in an early development stage. Hence, the native implementation of XML3D (the modified Chromium browser discussed in Section 6.2.2) was used for the project. At the target resolution of 1920×1080 pixels, we achieve similar rendering times compared to the established Lighting VR system [BLRS98] from Fraunhofer IAO.

According to the authors of the application, the main motivation for using XML3D for this project was its tight HTML integration and, associated with this, the easy integration of additional multimedia material, the easy development of 2D and 3D GUI elements including their touch-based control, and not at least the easy implementation of the application logic [SDHS13].

[1] `http://www.xml3d.org`, accessed 18 December 2015.

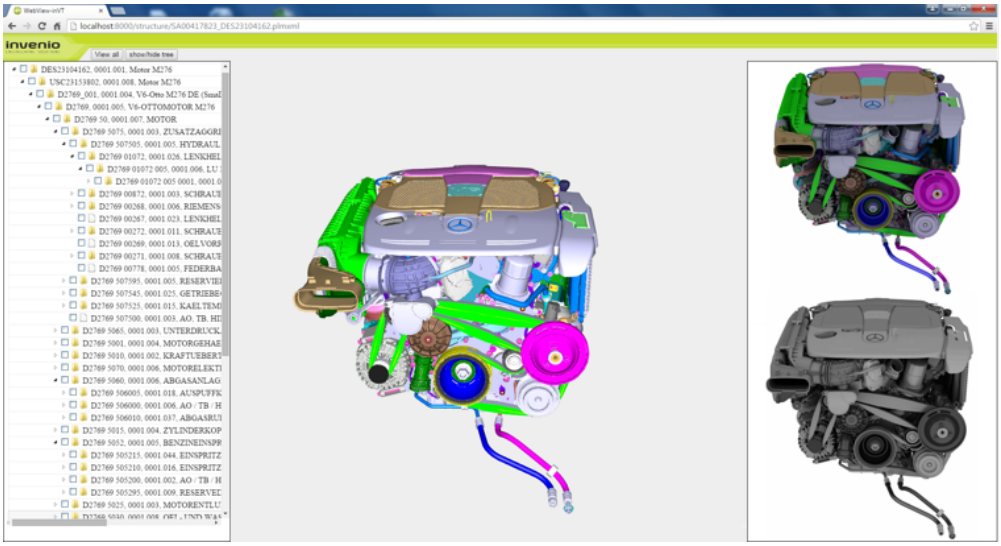


Figure 9.3: Screenshot of invenio WebView (courtesy by invenio VT GmbH). The product structure of the engine is displayed in a tree view on the left. In the center, the interactive XML3D scene is shown. It is synchronized with the tree view, i.e. clicking on parts of the engine not just highlights the part in the 3D scene, but also selects the same part in the product structure and vice versa. In addition, the user can request images of the current view rendered on the server using a ray tracer based on Intel Embree [WWB*14].

9.2 Invenio WebView

Invenio Virtual Technologies GmbH is a software and technology provider for virtual product development mainly for the automotive industry. Invenio developed the *WebView* component, which is an XML3D-based visualization module. In combination with Invenio's toolchain, it is possible to visualize arbitrary products from a product data management system (e.g. Siemens Teamcenter[2]) in the browser.

The WebView component is a web service and caching system. It represents product hierarchies in HTML and in XML3D. Product data is converted into XML3D assets, that are delivered to the client using Blast. Hence, the system makes extensive use of the asset instancing and the data delivery concepts of XML3D. Additionally, a custom material with simplified lighting leads to a better performance yet sufficient for the application. Figure 9.3 shows an engine exported from a PLM system to XML3D.

The component allows visualizing large data sets from the automotive industry. Obviously, the rendering performance largely depends on the target's hardware. For instance, a PDM export with 1580 objects and 21.7 million triangles can be rendered with xml3d.js in 12.6 ms (79.4 fps) on a Desktop PC (Intel i7-4790) with a dedicated GPU (NVIDIA GeForce GTX 970). The same scene renders in 272.5 ms (3.7 fps) on a Laptop (Intel i7-3630QM) with a mobile GPU (NVIDIA NVS 5400M).

Consequently, WebView can simplify the delivered data by either delivering the meshes with automatically reduced level of detail, or by just delivering hulls, which can be resolved to more detailed geometry on demand. This way the amount of delivered data can

[2] `http://www.plm.automation.siemens.com/de_de/products/teamcenter/`, accessed 18 December 2015.

be adapted to the application's needs and the client-device's performance and eventually displayed even on a tablet computer.

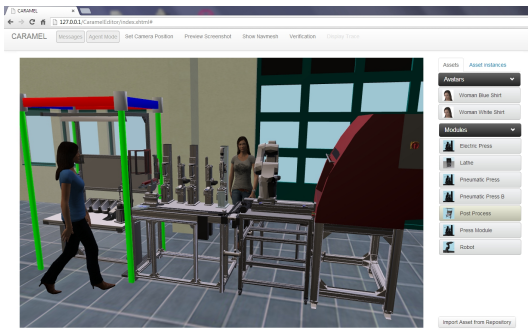
9.3 XML3D in Factory Planning

Increasingly, the continuing digitalization of the production process not only affects the design of products but also the design of the associated production facilities. In particular shorter development cycles and an increasing demand for customized products require flexible installations that can be (re-)configured quickly. In addition, suppliers are involved in this process to a growing extend. These changes prompt for new tools that allow planning and simulating a future factory in a collaborative way with remote partners.

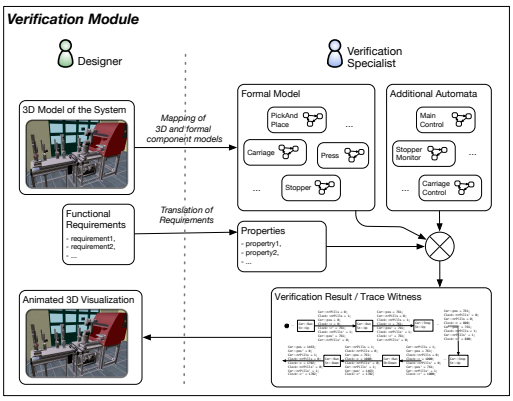
In the BMBF project *Collaborate3D*[3], concepts for such tools based on virtual prototyping have been developed. The proposed system [ZSC*13] consists of a collaborative 3D editor based on XML3D (Figure 9.4a). The editor allows configuring the factory model by adding and positioning factory assets and avatars. Two service-based modules help evaluating a configuration: The *verification module* evaluates functional properties of the configuration based on a formal model. This model is synchronized with the factory configuration, i.e. changing the factory in the browser automatically updates the formal verification model. If the verification module finds a constructive proof of the desired functional properties, it produces a trace witness, i.e. a description of the behavior that the system has to show in order to reach a certain desired state. Such a trace can be used to generate and trigger associated animations in the XML3D scene. These provide a comprehensible feedback about the results of the verification (Figure 9.4b).

The *avatar component* simulates interactions between workers and the production assets.

[3] `http://c3d.dfki.de/`, accessed 18 December 2015.



(a) Multiple designers can configure the factory collaboratively using the XML3D-based 3D editor. Functionality of the 3D editor includes adding factory assets and workers and placing these items using a 3D widget.



(b) The verification module interacts with the visualization. The configuration from the 3D editor updates the formal model of the verification. A valid result (trace witness) creates an Xflow animation that can be reviewed by the designers.

Figure 9.4: Collaborative design and evaluation of factory configurations in Collaborate3D. Images from [ZSC*13]

Using the 3D editor, designers can assign tasks to the avatars. The planning of tasks and resulting behavior of the avatars is controlled by an agent-based AI engine and the result is mapped to the visual avatars represented in XML3D. The agent-based avatars can be used to simulate workers including evaluation of reachability, time constraints, etc. Both simulation modules use semantic annotations to interpret the initial 3D scene and changes in the scene (cf. Section 6.3.2).

In the EU project *Interact*, researchers want to utilize workers' knowledge for "improving the precision and accuracy of digital human process simulation tools, thus achieving faster ramp-ups and first-time-right assembly processes"[4]. In this project, XML3D is again used for virtual prototyping, aiming to evaluate and maximize the ergonomics of future workplaces (see Figure 9.5). Therefore, a simulation engine synthesizes the required motions to achieve production tasks from small recorded motion sequences. XML3D is used to represent the results to work ergonomists, which should evaluate and improve the workplace with the user in the loop.

Both projects make extensive use of XML3D's dataflow concept in order to animate the virtual workers using animations and animation sequences that have been computed by external simulations and connected using service APIs. Additionally, both projects can leverage the connection to product data management systems (see previous section) in order to put the presentation into a realistic context.

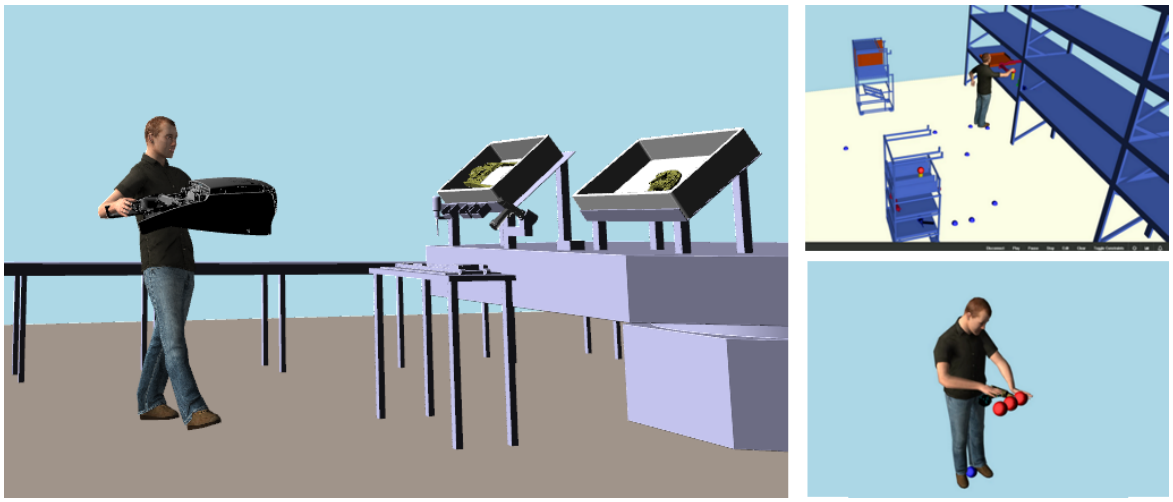


Figure 9.5: In Interact, researchers evaluate using virtual prototyping for planning future workplaces. The figure on the left shows a worker carrying a center console from the assembly line to the assembling place. The upper image a task picking up a tool form a rack system while following a planned path. On the image on the bottom, the worker performs a screw task. All required motions are simulated by a motion synthesis modules and applied to the XML3D/Xflow scene.

9.4 Conclusion

In this chapter we presented four non-trivial XML3D application from three application domains. Two of these applications run in a non-academic context.

[4] `http://www.interact-fp7.eu/`, accessed 18 December 2015.

Please note that – apart from some consulting – the researchers involved in the development of XML3D were not involved in implementing these applications. In general, no graphics experts were needed to realize the presented applications.

10 Evaluation and Discussion

The application examples discussed in the last chapter demonstrate the wide range of applications and requirements that are already supported by the current state of the XML3D architecture as presented in this thesis. However, a user-based evaluation is beyond the scope of this thesis. Instead we will review the design criteria developed in Chapter 3 to discuss them based on our results (Section 10.1). Subsequently, we will discuss the application model as a result of the novel concepts of the XML3D architecture and compare it with previous models (10.2). Finally, we will discuss some general limitations of our approach (10.3).

10.1 Review of the Design Criteria

In this section, we will review the four design criteria developed in Chapter 3: Web integration, renderer independence, usability, and expressiveness. We will argue how the concepts of the XML3D architecture help achieving the design criteria. In addition, we will provide indicators to demonstrate that we achieved our goals.

10.1.1 Integration into the Web Technology Stack

To meet this criteria, we designed XML3D as an extension to HTML5, leveraging W3C technologies such as the DOM and the DOM event model. However, merely integrating an XML-based language into the DOM would have been insufficient to meet the criteria (cf. X3DOM in Section 4.4.1). Instead, XML3D incorporates many of the HTML principles, e.g. restricts the data structure of visible elements to a tree, and has a deeper CSS integration.

We think the best indicator for our successful integration is XML3D's interoperability with jQuery. At present, jQuery is the most widely used JavaScript library.[1] While the DOM API provides a generic API to modify and monitor the web site, jQuery provides convenience functionality for the most common tasks. For instance, jQuery provides higher-level functionality to show and hide objects, to react on clicking on objects and for animations. All this functionality can also be applied to XML3D directly. In the following example, we use jQuery to animate the transparency of an XML3D object until the object is finally completely excluded from rendering:

```
jQuery({transparency: 0}).animate({transparency: 1}, {
  duration: 800,
  easing:'linear',
  step: function() {
    $(shaderId).text(this.transparency);
  },
```

[1] According to `http://w3techs.com`, accessed 17 November 2015.

```
  complete:  function() { $(".wall").hide(); }
})
```

With our approach, we achieved a new integration level of 3D graphics into web technologies exceeding previous approaches. As such, we can argue that we met our design criteria. Current limitations in CSS prevent an even deeper CSS integration, which would require new and more generic CSS properties being available in Web browsers.

10.1.2 Renderer Independence

One of the most challenging requirements was to not introduce any renderer-specific concepts and abstractions to the scene description, but nevertheless leverage the power and flexibility of recent GPUs.

We met this criteria by choosing a higher abstraction level for the scene description with self-contained concepts such as geometry, lights, and materials. Despite the high abstraction level, we are able to specialize the rendering of the scene for different target platforms and in particular for GPU pipeline shaders.

In order to exploit the functionality provided by graphics APIs we have developed *a*) data structures and types that map well to GPU buffers, *b*) a dataflow graph approach for data processing that can be mapped to the vertex processing stage of GPU pipelines[2], and *c*) a DSL for materials that can compile to different computations stages including vertex and fragments shaders of the GPU.

Despite being well-suited for GPU-based rendering, XML3D can still be rendered using other rendering algorithms. In this thesis we discussed using RTfact as back end for the native implementation of XML3D (Section 6.2.2). In Chapter 7.5, we have shown that shade.js materials can be rendered using forward and deferred-styled rasterization, ray tracing, and path-tracing (Figure 7.5).

Finally, Tamm and Slusallek propose an architecture that allows rendering of XML3D scenes on a server backend including distributed rendering with an arbitrary hierarchy of cluster nodes [TS16]. This architecture allows plugging in different renderers including a custom realtime ray tracer that exploits frame-to-frame coherence in a realtime context (see Figure 10.1).

10.1.3 Usability

Our design criteria included the usability of XML3D with a strong focus on learnability, scene organization, and performance. In terms of scene organization, the XML3D architecture provides a generalized addressing model that allows all referenced entities to be defined in external resources and asset instancing for encapsulation of complex structured scenes. These concepts allow for a fine-granular composition of scenes from smaller building blocks that can be organized in separate documents. As a result, only those objects are required to be exposed in the main HTML document that need to be modified as part of the application logic. Hence, the scene description remains concise, which improves the usability of XML3D tremendously.

[2]The actual mapping process has been briefly discussed in this thesis. For details, please refer to the related papers and the pending thesis of Felix Klein.

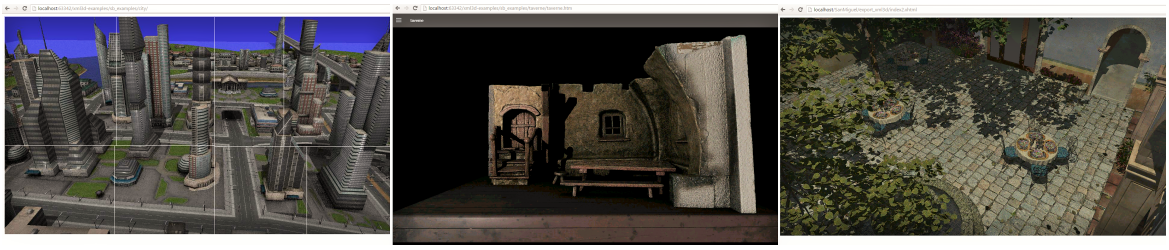


Figure 10.1: Three scenes described with XML3D in the browser but rendered on a server backend using a distributed realtime ray tracer based on Embree [WWB*14]. The white boxes in the city scene on the left show the contributions of each cluster node to the rendered frame. In the middle a laser-scanned movie set with a glossy table. The San Miguel hacendia scene on the right side has 7.7 Mio triangles, refraction for the glasses and the fountain, and a large amount of alpha mapped leafs (images from [TS16] by courtesy of the authors).

This novel level of encapsulation not only supports organizing 3D scenes, but also contributes to the learnability of XML3D. In particular the asset instancing allows separating 3D asset authoring (which still may involve expert knowledge) and development of the 3D application by the web developer. For instance, authoring of an animated 3D avatar may involve setting up complex data flow graphs that compute skinning, skeleton-based animation, and blending between multiple animations. A well authored 3D asset on the other hand can be simply configured with a simple set of parameters. For instance, an animated character can be included into a web page like this:

```
<model src="assets/avatar.xml#natalie">
  <assetdata name="animation-walk">
    <float name="key">3.12</float>
  </assetdata>
</model>
```

The animation can be controlled with a single parameter (*key*) exposed to the DOM. This parameter can in turn be animated using e.g. jQuery functionality.

The seamless integration into web technologies contributes to principles such as familiarity and predictability. For those concepts that we had to introduce, we focused on consistency. For instance, we use a single consistent data model for all entities of the abstract model. As we have shown above, using XML3D can be simple. However, we intended to not oversimplify the architecture at the costs of a monolithic design. Instead, XML3D offers incremental access, i.e. it offers a simple start for designing static scenes and using prefabricated assets. More complex dynamic scenes and advanced materials will require to learn the predefined operators and material models. Finally, at an expert level, XML3D offers custom operators, custom materials, and encapsulation of complex data processing in assets. However, this level is only necessary for special use cases. With this incremental approach we met our requirement to make simple things simple and complex things still possible.

In terms of performance, we have shown that we can exploit the data-parallelism offered by recent hardware without exposing renderer-specific concepts such as shaders. In particular Xflow and shade.js offer a novel degree of flexibility to scene designers without sacrificing performance or platform independence. This has been made possible by using data dependency analysis in Xflow dataflow graphs and in shade.js code using a compiler.

In addition to these concepts, we presented Blast. Blast not only allows for more efficient delivery of content. Since Blast allows streaming of content self-contained chunks, it is possible to show the rendering very early, which improves the user experience tremendously. But it is only by combining these concepts that we are able to create high performance applications such as the rendering of highly complex models with thousands of parts and millions of polygons (cf. the representation of CAD models described in Section 9.2).

On the other hand, and despite these measures and possibilities, users are still able to author scenes with poor performance. For instance, authors can create worst-case scenes with thousands of objects without exploiting instancing or scenes that require a lot of state changes. A smart implementation may detect and cushion some of these design flaws. However, XML3D does not prevent poor scene design. Hence, a similar scene can be designed in two different ways resulting in very different performance. This is a general issue in 3D scene libraries and strongly related to the design of graphics hardware.

10.1.4 Expressiveness

In terms of functionality, the XML3D architecture meets the requirements we developed earlier. This was shown in a series of application examples. Additionally, we want to compare XML3D's functionality to the feature set of common game engines. [RRA10] provides a comparison of eight popular open source 3D games engines and their most important features. The authors identified 55 graphical features supported by these engines. The number of supported graphical features per evaluated engine ranges between 18 (Panda3D) and 39 (Ogre3D), with an average of 29 features. XML3D supports 30 of the listed features, either as part of the model (e.g. mesh loading), Xflow (e.g. skeletal animation and skinning) or shade.js (e.g. anisotropic materials). Others come for free with HTML (2D-GUI). Some of the listed features are not related to XML3D but implementation-specific (e.g. shadow mapping techniques, BSPs). For these features we took the xml3d.js implementation as reference. We expect some of the implementation-specific features that are currently not available (e.g. occlusion culling) to show up in xml3d.js as soon as the required hardware features are exposed through the upcoming WebGL 2.0 API.

Most of the features not supported in XML3D fall under the category "Special Effects". Some of these effects (e.g. Fire, Explosions, Water, Weather) could probably be implemented using a combination of Xflow with animated shade.js materials. However, we considered only those features which have actually been used in at least one of the applications listed in the last chapter.

We identified four features, that cannot be easily implemented with XML3D because concepts are missing. These include LOD, post-processing (lens-flare, motion-blur), and atmospheric effects (fog). We discuss these missing concepts in more detail under Future Work (Section 11.2).

The paper also identifies non-graphical features, including networking, sound, artificial intelligence (AI), and physics. Networking (XHR, WebSocket, WebRTC) and sound (Audio API) are inherent features of the web platform. For AI and physics we presented integration strategies in Section 6.2.1.8 and Section 9.3.

We consider XML3D expressive enough to compare to the graphics and rendering functionality of game engines. Due to its flexible approach, XML3D may even exceed the domain-specific functionality provided by those engines. For instance, game engines re-

quire a specific structure and format for skinned characters. Thus, authors need to adapt their input data to fit to these data structures. In contrast, in XML3D the data processing required for skinning can be adapted to the available input data (see also the application model discussed in the next section).

However, we do not consider XML3D as a game engine replacement. Game engines offer a lot of game-specific high-level functionality that is not available in XML3D. Also, more recent game engines offer content creation tools and game-specific development environments not available for XML3D. Finally, game engines also offer better rendering quality than our current XML3D implementations. However, these restrictions are mainly due to limitations in engineering resources and restrictions of the browser environment rather than being conceptional issues.

10.1.5 Conclusion

In this section we discussed the extent to which the XML3D architecture attained the design criteria we developed in Chapter 3 against the background of the concepts and applications presented in this thesis. We can conclude that we meet these criteria to a large extend. All concepts that we presented in this thesis contribute to the design criteria. Hence, we cannot remove any of our concepts without missing our requirements.

On the other hand, there are obviously things to add and improve: Some of the current limitations have been already discussed during the review. Most of these limitations can be addressed in future work and are consequently discussed in the Future Work section. Some more general limitations of our approach are discussed in Section 10.3. However, before discussing those, we will discuss the impact of the novel concepts of the XML3D architecture on the design of 3D applications in the next section.

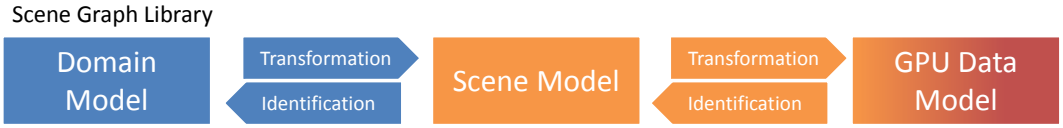


Figure 10.2: Typical application model using a scene graph library: Two model transformations are involved. Firstly, the application (blue) needs to map a subset of the domain model to the scene model. The scene graph library (yellow) transforms the scene model to the GPU data model (red), i.e. to buffers and (generated) shaders. Interactive applications require also the opposite path: The library identifies objects and per-pixel data from one or more framebuffers. The application needs to map the identified scene objects to objects from the domain model. Note, that if the domain model is modified, the whole transformation process needs to be reevaluated.

10.2 Application Model

In this section, we will discuss the classical application models that apply when developing a 3D application with *a*) a scene graph library, *b*) a dataflow-based scene library, and *c*) direct use of a graphics API without an intermediate library. Finally, we will show how the XML3D architecture contributes to resolving some issues with these classical models.

Scene Graph Library In general, implementing a 3D application using a scene graph library involves two model transformations (see Figure 10.2). The application holds its information in an application-specific domain model, which incorporates both, data and behavior. The domain-model is used to solve problems related to that domain. Often, the domain model is a representation of real-world objects. In *Building Information Modeling* (BIM), for instance, the domain-model could represent the structure and information about a building including a set of rules (behavior) that the building needs to comply to. In order to visualize the domain model, it needs to be mapped to the library-specific scene model.

In general, only the subset of the domain model relevant for the visualization is mapped to the scene model. Also, the domain model may not include all information required for the visualization. For instance, a domain model describing an airplane's cabin configuration, may include a type designation for each seat, but not its geometry. Also materials may be described only on a very high abstraction level (e.g. aluminum). Hence it is required to incorporated additional resources during the first mapping step. The model-to-model mapping requires mapping of data structures (e.g. for the representation of hierarchies) and data processing (e.g. mapping attributes to colors in the scene).

In a second transformation, the scene graph library maps the scene model to the data structures suitable for the graphics API. This includes buffers for vertex parameters and uniforms as well as generated shaders that implement the semantics of the scene model.

For *interactive* 3D applications, i.e. if a user wants to directly interact with the 3D model, the opposite mapping needs to be applied. If the user want to retrieve informations about the domain model using the 3D model as a *view* in a MVC architecture, the library also needs to identify 3D scene objects from the rendered framebuffer (picking). Similarly, the application needs to map the 3D scene objects back to the objects of the domain model. Then, additional information about the object can be retrieved and presented to the user. Other use cases not only require picking on an object level, but also need to pick data

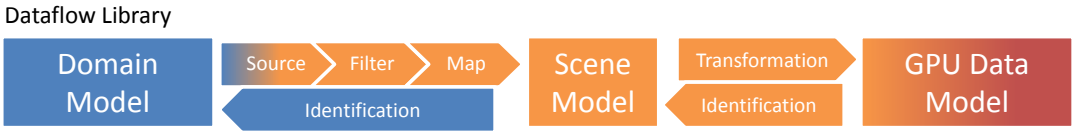


Figure 10.3: Dataflow-based libraries such as VTK provide mapping operations that are frequently used in scientific visualization applications. Hence, they support the first transformation step while the remaining model remains the same.

from the domain model based on the rendered representation. If domain data has been mapped, e.g. to a color, a simple back-mapping may be lossy and additional measures are necessary.

If the interaction with the domain model is not limited to information retrieval and a direct manipulation of the domain model is desired, the 3D scene acts not only as a view, but also as a controller. In such an architecture, or if the domain model may change over time, both transformation steps needs to be reapplied for all involved objects when the domain model changes. Therefore and since developing the domain to scene model transformation is a main effort in 3D application development, it should be a key strategy to keep the transformations as small and efficient as possible.

Dataflow-based Library Models used in scientific visualization have some common characteristics, e.g. less structure, and larger amounts of data compared to models from other domains. Consequently, libraries such as VTK provide library functionality to support the first transformation step (Figure 10.3). To be able to adapt the library functionality to the domain model, a dataflow approach is chosen: The operations can be arranged flexible in the sense of building blocks. Operators include functionality for reading the data (source), filtering out relevant subsets for visualization (filter), and for performing the required data processing (map). However, data still needs to be mapped to a very static scene model that is then converted to GPU structures in a second transformation.

No Scene Model In contrast, when implementing a 3D visualization for a specific domain model only, the intermediate scene model can be omitted: The application generates GPU data structures directly from the domain model and application-specific shaders (Figure 10.4). This approach results in best runtime performance because only a single transformation is involved. On the other hand, this model scales for very few applications

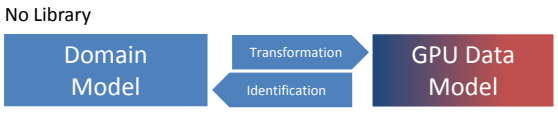


Figure 10.4: Omitting an intermediate library allows creation of GPU data structure that are specific to the application's domain model. Obviously, the absence of an additional abstraction layer yields increased performance but comes at the cost of increased development efforts and decreased platform scalability.

because it requires in-depth graphics knowledge and maintenance for the many heterogeneous hardware platforms. Additionally, it ties the application to GPU rendering.

Our Approach The XML3D architecture provides means for simplifying the application development process in many ways:

1. With external references, it is easy to link external visual representations (asset) of the domain object instances during the transformation. With the asset instancing concept, the interfaces of assets can be adapted to the objects in the domain model. Then, each asset can be configured based on the attributes of the related domain object instance. At best, there is a 1:1 mapping between domain model objects and corresponding asset instances.

2. Due to the GPU-friendly data structures of XML3D and since data processing is done explicitly in the Xflow graph, there is virtually no data processing necessary to transform the XML3D scene data structures to GPU data structures.

3. Similar to dataflow libraries, Xflow can be used to describe the transformation from the domain model to the scene model (optimally hidden inside an asset). In contrast to dataflow libraries, Xflow maps the data processing to GPU whenever possible.

4. Similar, shade.js can be used to map data from the domain model directly to visual material parameters. Again, this mapping gets compiled to specialized GPU shaders.

As a result, 3D application developers can take advantage of the higher abstraction level of the XML3D scene description, the data processing description, and the material description. At the same time, the compiler and the dataflow analysis create an execution model that is, at best, very close to an immediate use of the graphics API.

10.3 Limitations

Besides those limitations already discussed during review of the design goals, there are some more general limitations we want to discuss in this section.

Client vs. Server We have shown that our XML3D implementation can render, for instance, a car model from the automotive industry with more than 20 Million triangles (see Section 9.2). These kind of scenes run way above the synchronization rate of most systems (60Hz) even on a mid-range desktop PC with dedicated graphics hardware. On a laptop with integrated graphics hardware, the same scene runs with only 4Hz. The size of such a scene exceeds the hardware capabilities of current smartphones and tablet PCs. While the performance of integrated graphics hardware is steadily increasing, the demands on the hardware is increasing as well (larger models, more dynamics, more passes to increase the realism of lighting). The available hardware is an even more decisive factor when it comes to global illumination rendering algorithms. Thus, due to the heterogeneous devices used to access web pages, one can call into question if a client-side technology is the right approach at all. Server-sided rendering comes with many advantages including known hardware and easier protection of intellectual property for the content.

In our opinion, server-based rendering for 3D scenes integrated into web pages has the same requirements on the scene description in terms of functionality, renderer-independence, expressiveness, etc. As a result, the novel concepts for 3D scene descriptions presented in this thesis equally apply to server-based rendering as well. We do not see any reason why not to use the established and well-known DOM API to access and modify the scene on a server. At its best, it is transparent for the application developer as well as for the user where the scene is actually rendered. In such a system, the rendering location can be chosen freely based on parameters such as *a*) the user's device capabilities (CPU and GPU), *b*) the user's access to a power source[3], *c*) network quality, *d*) required responsiveness of the application, and *e*) required quality of the rendering.

As already discussed in Section 10.1, Georg Tamm is conducting research in using XML3D as a renderer-independent scene description in a adaptive and distributed server-based rendering environment.

Rendering Consistency A second conceptual limitation concerns the consistency of rendering results. The design of the XML3D architecture treats the lighting component as a black box. Light interaction of materials can be configured and provision is made to configure emissive objects (see also Future Work). However, it is left to the implementation in which way the rendering equation is approximated and how much effort is spent to achieve such an approximation. As a result, the rendering of a scene may differ between different XML3D implementations (see also Figure 7.5).

We presented the first general scene description that allows rendering non-trivial dynamic and interactive scenes with procedural materials using several rendering approaches. Hence, it is the first approach that allows for scaling the rendering quality to the requirements of an application (up to global illumination) without changing the scene description. This flexibility comes along with the effect that it is not possible to define the result of each pixel exactly. We expect, however, that some applications need to be able to predict the exact outcome of the rendering on every device. Again, server-based rendering with known implementations and hardware and therefore consistent and predictable results could be a solution for these kind of applications.

Convenience Functionality Despite its higher abstraction level compared to graphics APIs, XML3D can be considered low-level and generic by design when compared to scene descriptions such as X3D or A-Frame. However, we found that users asked for built-in convenience elements or domain-specific functionality (which we intentionally not provide), even if the requested functionality can easily be implemented using Xflow or JavaScript. A-Frame, for instance, focuses on simplicity: VR worlds can be created easily also by novices at the price that more complex scenarios tend to become impossible.

However, we think that Web Components can be used to add convenience and domain-specific functionality on top of XML3D without the need to extend its core functionality. We will discuss this option in more detail in future work (Section 11.2).

[3]For instance, we expect server-based rendering to have less impact on battery life (cf. [SAK15]).

11 Conclusions and Future Work

In this thesis, we proposed the design of a novel render-independent, web-integrated, and interactive 3D scene description. The proposed XML3D architecture (Chapter 5) goes beyond previous approaches (Chapter 4) as it is better integrated with the web technology stack, solves the dilemma of general material descriptions (Chapter 7.5), integrates data processing based on graphs of operations, and provides a general way for efficient delivery of content to the client (Chapter 8).

With the presented implementations we demonstrate that the concepts we presented are beyond a purely theoretical study. Moreover, we showed that our approach is extensible on various levels (both in Chapter 6). We presented various non-trivial application examples that demonstrate the applicability of our concepts (Chapter 9). Finally, we evaluated our design goals and discussed the impact of our work on common application models (Chapter 10).

11.1 Conclusion

The XML3D architecture consists of seven novel concepts. While some of these concepts may be considered as an incremental contribution, it requires all of them to answer the specific research questions from the introduction:

- **How does a web-integrated 3D scene description look like?**
  XML3D is the first 3D scene description to integrate seamlessly into existing web technologies. With our approach to extend HTML, we have met our requirement on a more thorough integration in particular with CSS and DOM events. We have shown that our approach yields a higher integration level compared to the existing X3DOM approach. Figure 11.1 depicts the integration of the XML3D architecture into the web technology stack.

- **How to describe highly dynamic effects in a declarative approach?**
  With Xflow, we provide a way to describe data processing (as required for dynamic effects) based on a graph of operations. Hence, we expose data processing to the user: Scene developers can compose the required functionality from basic building blocks in a very flexible way. Despite the offered flexibility, we have shown that suitable operations can still be mapped to available hardware.

- **How to design a lean yet flexible 3D scene description?**
  In previous approaches, data processing is a library-internal mechanism provided by the abstract model which, as a result, tends to become large with increasing functionality. With Xflow, we deliberately refrain from hiding data processing in the abstract model and expose it to the user. As a result, we have a very lean abstract model at significantly increased flexibility.

- **How to describe the material of a 3D object?**
  Our approach is the first to solve the material dilemma discussed in Section 4.1. The adaptivity of our material description language allows for materials which are general enough to apply to various 3D assets. With defined interfaces to space transformations and to the lighting, our materials remain portable. Our materials are self-contained and can be shared across the web. Despite the higher abstraction level, we are still able to fully exploit GPU capabilities. We have shown that we do not sacrifice performance despite the higher abstraction level.

- **How can we compose 3D scenes?**
  We presented a concept for encapsulation of reusable 3D assets. In contrast to previous approaches, our assets may include arbitrary dynamic effects and are still fully configurable during instantiation. This is achieved exploiting our consistent general data model with its overriding rules.

  Using our generalized addressing mechanism based on URIs, authors can compose scene from several documents and web services. In particular, our architecture is the first to enable composing scenes from various other 3D graphics formats. We have shown this for mesh data in STL and OpenCTM formats, scene formats (glTF), and compression techniques (Open3DGC, SIG). We expect that our concepts are general enough to allow integration of even more third-party formats.

- **How to deliver 3D content to the client?**
  We presented Blast, a container format for efficient delivery of highly structured binary data with flexible compression using a code on demand approach. Blast is schemaless and thus general enough to represent any kind of content, also beyond 3D graphics. We have shown that meshes, textures, and assets can be represented in compressed form in Blast containers.

With answering these six research question, we also answer the superordinate research question and conclude that it is feasible to integrate interactive 3D graphics as first class objects into the web technology stack.

Our research question focuses on the web context. However, our research also contributes to 3D graphics more generally: With XML3D we provide an abstraction level that is general enough to be rendered with virtually any rendering algorithm. We have shown this for forward and deferred-style GPU rendering, ray tracing, and path tracing. Despite the higher abstraction level, we do not impede the benefits of programmable graphics hardware. Quite the contrary, our architecture provide means for customizing per-vertex processing using Xflow and material-related aspects using shade.js. Given the increasing popularity of global illumination render systems and the increasing diversity of GPU-based rendering approaches, XML3D and its concepts are destined to serve as foundation for a general 3D scene description also beyond the web context.

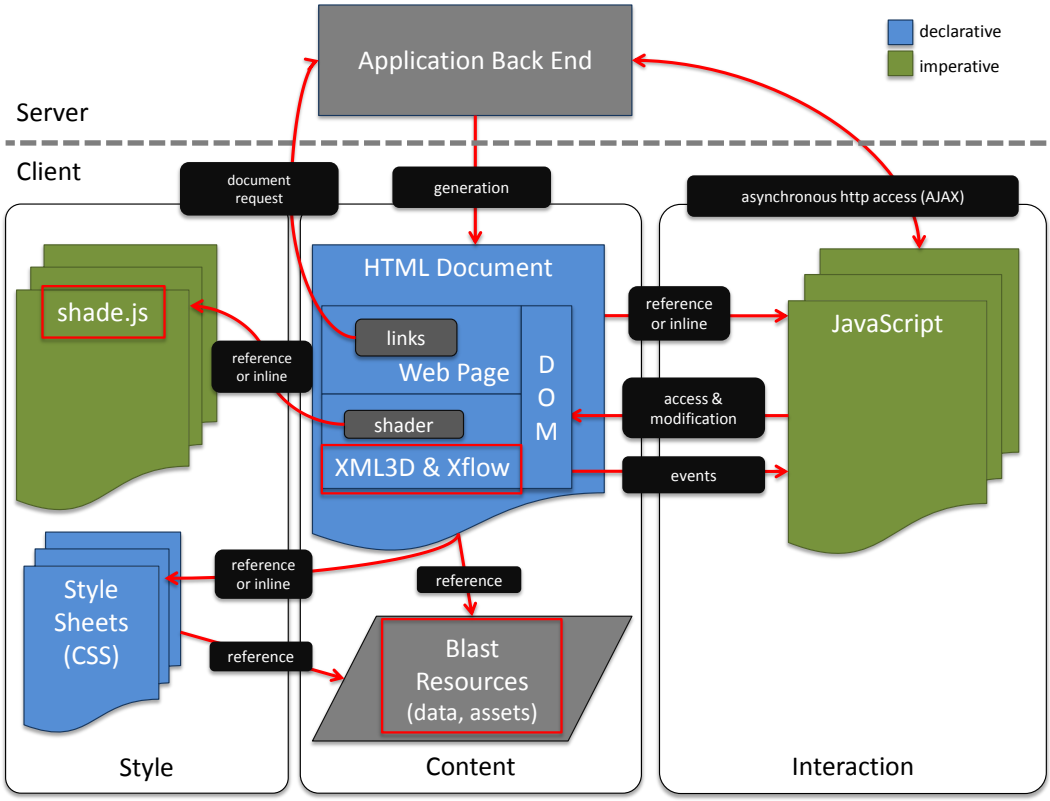


Figure 11.1: The technologies of the XML3D architecture integrated into the web technology stack: XML3D and Xflow extend HTML5 and are accessible from JavaScript through the DOM API. We use CSS to attach material instances to XML3D elements but also introduce programmable material models to the web technology stack with shade.js. Blast is necessary as a container format that supports the delivery of structured and compressed binary 3D data and assets.

11.2 Future Work

The XML3D architecture is an open and living architecture. The related libraries xml3d.js (including its Xflow implementation) and shade.js are under active development and freely available on GitHub.

Obviously, there is a list of topics which need further investigation, research, and development:

Geometry Shaders In this thesis we focused on the OpenGL ES/DirectX 9 graphics pipeline exposed by WebGL 1.0. However, we would like to add concepts that allows exploiting also the capabilities of more recent graphics pipelines, again without sacrificing renderer-independence and separation of concerns. For instance, the DirectX 11 pipeline (Figure 2.5) provides four additional programmable stages. In particular geometry shaders are useful concepts that help to avoid creation and modification of geometry on the CPU. A possible approach to allow the same functionality in XML3D could be to extend the <mesh>'s *type* attribute to not only support predefined primitives but also scripts based on shade.js that can e.g. emit primitives from input parameters.

Programmable Light Models In this thesis, we presented a concept to configure the lighting for a surface and an advanced general framework for lights that unifies the definition of geometry with the definition of light instances. Such light instances reference and configure a *predefined* light model (e.g. point lights) that describes a specific light distribution.

As a result, the lighting of an object is a grey box that can be configured using programmable material models and predefined light models. Similar to programmable material models, it would be possible to use shade.js concepts to support *programmable* light models. This would enable describing application-specific lights: A "projective light" could for instance project light based on a texture or on a procedural pattern. A "goniophotometric light" could emit intensity based on (measured) samples on the unit sphere.

Providing programmable light models is not a new concept: RenderMan provides light-shaders. The interface of these shaders is, however, very specific for ray tracing. Hence, it would be necessary to develop a more general interface for the sampling of lights suitable for the majority of rendering algorithms. We assume that the adaptivity of shade.js could be used to adapt the sampling to the renderer's needs.

Custom Xflow operators Currently, we provide a fixed set of useful Xflow operators generic enough to be sufficient for most use cases. The set of operators can be easily extended in the xml3d.js Polyfill implementation based on plug-ins. The operator author may provide multiple, platform-specific implementations of such an operator, but there is no way to provide an operation once that can then adapt to multiple platforms (similar to shade.js for materials). Additionally, our current implementation requires annotations and conventions so that the dataflow analysis is able to compute data dependencies, to be able to merge operations, to perform optimizations, and to create code for different platforms. Although there is work in progress to use shade.js to describe Xflow operators, use the shade.js compiler framework to analyze the code, and to translate the result to different platforms, the interfaces of the operators still need to be declared and annotated for the dataflow analysis.

Hence, a direction in future work is to evaluate interfaces that allow shifting Xflow operators from system space to user space and thus allows authors to write custom Xflow operators in a portable way.

Post-Processing Many multi-pass techniques (e.g. shadow mapping, subsurface scattering, and SSAO) are used to better approximate the lighting of the scene. Since lighting is a grey box in XML3D, this techniques can be applied without further notice to the user of the system. Another prominent use case for multi-pass rendering is post-processing. Most post-processing effects are image processing effects that combine multiple different versions of a scene rendering (i.e. render buffers) using a specific operation. We suggest to extend Xflow to be able to describe post-processing effects as a dataflow graph. Xflow is capable of describing operators and data dependencies – here between the render buffers to be generated – and can map these operations to the GPU.

Deeper CSS Integration Currently, we reuse existing CSS properties and attach materials to the scene hierarchy using CSS style rules. However, CSS lacks for defining generic properties. Thus we had to develop our own data model for defining generic parameters. However, a 3D scene description could benefit from the power of CSS and CSS selectors to define the parameters of material, light, and camera models. In [SSS15] we outline and implement a proof of concept for such a deeper CSS integration based on custom CSS properties.

Level of Detail In its current version XML3D has no concept for level of detail (LOD). Switching models naively based on the distance to the camera similar to the X3D <LOD> node is easily possible in JavaScript in a layer above XML3D and thus does not justify extending the abstract model. Obviously, it would be easy to extend the abstract model and rendering system by a more sophisticated progressive LOD mechanism similar to X3DOM's POP buffers [LJBA13]. However, we would like to investigate and develop a uniform LOD approach that integrates consistently into the XML3D architecture and in particular into our data processing and data delivery concepts. Next to geometry LOD, the shade.js compiler could provide automated LOD for materials similar to [Pel05].

Component Model We presented a powerful concept for encapsulation of 3D assets including geometry, materials, transformations, animations, and general data processing while maintaining full configurability. We intentionally did not include scripting into 3D assets to be able to exploit coherence between asset instances. However, for many use cases it is practical to be able to attach behavior to an asset and to expose interfaces on a higher abstraction level. For instance, if a car asset contains an animation opening the doors of the car, it could expose a *openDoor(side)* method instead of leaving it up to the developer to identify the parameter in the data flow graph that needs to be overridden to start the animation. A component model that allows bundling assets with scripting would allow authoring of reusable widgets and objects with a logic that is either driven by an internal logic or for instance by a simulation that runs as a web service.

In addition, we discussed the demand of users to have convenience functionality as first class citizens in the scene description. Therefore, we need means to extend XML3D with convenience and domain-specific elements which hide existing XML3D functionality and scripting behind a facade.

Lemme et al. adapt XML3D in order to harmonize with Web Components [LSSS16]. They show how Web Components can be used to implement convenience nodes on top of a small set of elements and concepts derived from XML3D that they call the *Basic Building Blocks of Declarative 3D*. The authors implement some basic convenience nodes and "envision community-maintained repositories of reuseable components for a wide range of Dec3D usecases".

Custom Picking Shaders For interactive scenes, XML3D supports object picking. For picking on data level, XML3D supports picking of attributes with a fixed semantic, e.g. for world space positions and for the color in the main framebuffer.

Providing normal picking is problematic because the meaning is ambiguous and normal could refer e.g. to the surface normal or to the shading normal (see Section 6.2.1.9). The same applies to other attributes. Hence, XML3D should provide user-defined picking that allows customizing the picking result using shade.js. This way, the user can pick not only arbitrary attributes, but also the result of custom expressions. It would also allow picking domain data directly from the rendered model, which would move the resulting execution model of XML3D even closer to the immediate use of a graphics API (cf. Section 10.2).

Relative Object Placement As already discussed in Section 5.2, relative placement of objects based on their dimensions is a common functionality in HTML, but has never been used in 3D scene descriptions. We think that such a novel positioning strategy would be beneficial for many 3D applications but leave the actual proof to future work.

Beside these technical investigations, an interesting topic is the quantitative evaluation of the XML3D architecture's usability. Now that we solved the majority of technical research questions, developed a portable and flexible 3D scene description, and present a consistent architecture for HTML-integrated 3D graphics, XML3D offers a good starting position to address this topic.

Bibliography

[10G13] 10Gen I.: BSON - Binary JSON Specification, 2013. URL: http://bsonspec.org/. 174

[AMHH08] Akenine-Möller T., Haines E., Hoffman N.: *Real-Time Rendering, Third Edition*. A K Peters/CRC Press, 2008. doi:10.1201/b10644-23. 15, 16

[Arn11] Arnaud R.: Rest3d: 3D for the REST of us, 2011. URL: https://rest3d.wordpress.com/. 164

[Aus05] Austin C. A.: *Renaissance: A Functional Shading Language*. Master's thesis, Iowa State University, 2005. URL: http://chadaustin.me/hci_portfolio/thesis.pdf. 147, 148

[AW90] Abram G. D., Whitted T.: Building Block Shaders. *SIGGRAPH Comput. Graph. 24*, 4 (1990), 283–288. doi:10.1145/97879.97910. 147

[BB07] Boeing A., Bräunl T.: Evaluation of real-time physics simulation systems. In *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia* (2007), GRAPHITE '07, ACM, pp. 281–288. doi:10.1145/1321261.1321312. 130

[BDR04] Behr J., Dähne P., Roth M.: Utilizing X3D for Immersive Environments. In *Proceedings of the Ninth International Conference on 3D Web Technology* (2004), Web3D '04, ACM, pp. 71–78. doi:10.1145/985040.985051. 44

[BEJZ09] Behr J., Eschler P., Jung Y., Zöllner M.: X3DOM: A DOM-based HTML5/X3D Integration Model. In *Proceedings of the 14th International Conference on 3D Web Technology* (2009), Web3D '09, ACM, pp. 127–135. doi:10.1145/1559764.1559784. 41, 42, 50, 52

[Ben86] Bentley J.: Programming Pearls: Little Languages. *Commun. ACM 29*, 8 (1986), 711–721. doi:10.1145/6424.315691. 145

[Ben06] Benthin C.: *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, Saarbrücken, Germany, 2006. 15

[BH15] Behr J., Hudson A.: Prototype, 2015. URL: http://sourceforge.net/p/x3dom/mailman/message/33435593/. 43

[BHK*02] Black A. P., Huang J., Koster R., Walpole J., Pu C.: Infopipes: An abstraction for multimedia streaming. *ACM Multimedia Systems Journal 8* (2002), 406–419. doi:10.1007/s005300200062. 35

[BJDA11] Behr J., Jung Y., Drevensek T., Aderhold A.: Dynamic and interactive aspects of X3DOM. In *Proceedings of the 16th International Conference on 3D Web Technology - Web3D '11* (2011), ACM Press, pp. 81–87. doi:10.1145/2010425.2010440. 52

[BJFS12] Behr J., Jung Y., Franke T., Sturm T.: Using Images and Explicit Binary Container for Efficient and Incremental Delivery of Declarative 3D scenes on the Web. In *Proceedings of the 17th International Conference on 3D Web Technology - Web3D '12* (2012), ACM Press, pp. 17–25. doi:10.1145/2338714.2338717. 53, 74, 89, 162, 175

[BK02] Bischoff S., Kobbelt L.: Streaming 3D geometry data over lossy communication channels. In *Proceedings of the 2002 IEEE International Conference on Multimedia and Expo* (2002), vol. 1, pp. 361–364. doi:10.1109/ICME.2002.1035793. 175

[BL89] Berners-Lee T.: Information Management: A Proposal. *Word Journal Of The International Linguistic Association February 2* (1989), 1–10. URL: http://www.w3.org/History/1989/proposal.html. 10

[BLC90] Berners-Lee T., Cailliau R.: WorldWideWeb: Proposal for a HyperText Project, 1990. URL: http://www.w3.org/Proposal. 10

[BLFM98] Berners-Lee T., Fielding R., Masinter L.: Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396 (Draft Standard), Aug. 1998. Obsoleted by RFC 3986, updated by RFC 2732. URL: http://www.ietf.org/rfc/rfc2396.txt. 10

[BLHL01] Berners-Lee T., Hendler J., Lassila O.: The Semantic Web. *Scientific American 284*, 5 (2001), 34–43. doi:10.1038/scientificamerican0501-34. 139

[Bli77] Blinn J. F.: Models of light reflection for computer synthesized pictures. *ACM SIGGRAPH Computer Graphics 11* (1977), 192–198. doi:10.1145/965141.563893. 26, 27, 48

[BLMM94] Berners-Lee T., Masinter L., McCahill M.: Uniform Resource Locators (URL). RFC 1738 (Proposed Standard), Dec. 1994. Obsoleted by RFCs 4248, 4266, updated by RFCs 1808, 2368, 2396, 3986, 6196, 6270. URL: http://www.ietf.org/rfc/rfc1738.txt. 40

[BLRS98] Blach R., Landauer J., Rösch A., Simon A.: A Highly Flexible Virtual Reality system. *Future Generation Computer Systems 14*, 3-4 (1998), 167–178. doi:10.1016/S0167-739X(98)00019-3. 179

[Bly06] Blythe D.: The Direct3D 10 System. In *ACM SIGGRAPH 2006 Papers* (2006), SIGGRAPH '06, ACM, pp. 724–734. doi:10.1145/1179352.1141947. 16

[BO04] Burns D., Osfield R.: Tutorial: Open scene graph A: introduction tutorial: Open scene graph B: examples and applications. In *IEEE Virtual Reality 2004* (2004), IEEE, pp. 265–265. doi:10.1109/VR.2004.1310100. 33, 49

[BOH11] Bostock M., Ogievetsky V., Heer J.: D3: Data-Driven Documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2011). doi:10.1109/TVCG.2011.185. 11

[BPP94] Bell G., Parisi A., Pesce M.: The Virtual Reality Modeling Language, 1994. URL: https://web.archive.org/web/20031128125521/http://www.web3d.org/VRML1.0/vrml10c.html. 40

[BZN13] Bryan P., Zyp K., Nottingham M.: JavaScript Object Notation (JSON) Pointer. RFC 6901 (Proposed Standard), Apr. 2013. URL: http://www.ietf.org/rfc/rfc6901.txt. 167

[CB97] Carey R., Bell G.: *The Annotated VRML 2.0 Reference Manual*. Addison-Wesley Longman Ltd., 1997. 20, 40

[CCC87] Cook R. L., Carpenter L., Catmull E.: The Reyes image rendering architecture. *ACM SIGGRAPH Computer Graphics 21*, 4 (1987), 95–102. doi:10.1145/37402.37414. 16

[CNS*02] Chan E., Ng R., Sen P., Proudfoot K., Hanrahan P.: Efficient Partitioning of Fragment Shaders for Multipass Rendering on Programmable Graphics Hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (2002), HWWS '02, Eurographics Association, pp. 69–78. 147

[CNW99] Chen Y. H., Ng C. T., Wang Y. Z.: Generation of an STL file from 3D measurement data with user-controlled data reduction. *The International Journal of Advanced Manufacturing Technology 15*, 2 (1999), 127–131. doi:10.1007/s001700050049. 26

[CO15a] Coumans E., Others: Bullet Constraints, 2015. URL: http://www.bulletphysics.org/mediawiki-1.5.8/index.php/Constraints. 133

[CO15b] Coumans E., Others: Bullet Physics Library, 2015. URL: http://bulletphysics.org/. 135

[COM98] Cohen J., Olano M., Manocha D.: Appearance-preserving Simplification. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques* (1998), SIGGRAPH '98, ACM, pp. 115–122. doi:10.1145/280814.280832. 155

[Coo84] Cook R. L.: Shade Trees. *SIGGRAPH Comput. Graph. 18* (1984), 223–231. doi:10.1145/280811.280984. 147

[Cro06] Crockford D.: The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006. Obsoleted by RFC 7159. URL: http://www.ietf.org/rfc/rfc4627.txt. 49, 162, 174

[CT82] Cook R. L., Torrance K. E.: A Reflectance Model for Computer Graphics. *ACM Trans. Graph. 1*, 1 (1982), 7–24. doi:10.1145/357290.357293. 152

[CV07] Coumans E., Victor K.: COLLADA Physics. In *Proceedings of the Twelfth International Conference on 3D Web Technology* (2007), Web3D '07, ACM, pp. 101–104. doi:10.1145/1229390.1229407. 129

[dC03] de Carvalho G. N. M.: High-level Procedural Shading VRML/X3D. In *ACM SIGGRAPH 2003 Web Graphics* (2003), SIGGRAPH '03, ACM, p. 1. doi:10.1145/965333.965337. 42

[dCGP04] de Carvalho G. N. M., Gill T., Parisi T.: X3D Programmable Shaders. In *Proceedings of the Ninth International Conference on 3D Web Technology* (2004), Web3D '04, ACM, pp. 99–108. doi:10.1145/985040.985055. 42

[Deu96a] Deutsch P.: DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), May 1996. URL: http://www.ietf.org/rfc/rfc1951.txt. 102

[Deu96b] Deutsch P.: GZIP file format specification version 4.3. RFC 1952 (Informational), May 1996. URL: http://www.ietf.org/rfc/rfc1952.txt. 102

[DH02] Döllner J., Hinrichs K.: A Generic Rendering System. *IEEE Transactions on Visualization and Computer Graphics 8*, 2 (2002), 99–118. doi:10.1109/2945.998664. 34

[DM75] Dennis J. B., Misunas D. P.: A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture* (1975), ISCA '75, ACM, pp. 126–132. doi:10.1145/642089.642111. 35

[DPGS10] Di Benedetto M., Ponchio F., Ganovelli F., Scopigno R.: SpiderGL: A JavaScript 3D graphics library for next-generation WWW. In *Proceedings of the 15th International Conference on Web 3D Technology - Web3D '10* (2010), ACM Press, pp. 165–174. doi:10.1145/1836049.1836075. 49

[DSR*13] Doboš J., Sons K., Rubinstein D., Slusallek P., Steed A.: XML3DRepo: A REST API for Version Controlled 3D Assets on the Web. In *Proceedings of the 18th International Conference on 3D Web Technology* (2013), Web3D '13, ACM, pp. 47–55. doi:10.1145/2466533.2466537. 6, 60, 164, 172

[DWWS04] Dietrich A., Wald I., Wagner M., Slusallek P.: VRML Scene Graphs on an Interactive Ray Tracing Engine. In *Virtual Reality, 2004. Proceedings. IEEE* (2004), pp. 109–282. doi:10.1109/VR.2004.1310063. 42

[ECM11] ECMA International: Standard ECMA-262 - ECMAScript Language Specification, 2011. URL: http://www.ecma-international.org/publications/standards/Ecma-262.htm. 1, 9, 11, 58

[Eic05] Eich B.: JavaScript at Ten Years. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming* (2005), vol. 40, p. 129. doi:10.1145/1090189.1086382. 11

[Ele85] Electronic Arts: Electronic Arts Interchange File Format, 1985. 168

[Ell04] Elliott C.: Programming Graphics Processors Functionally. In *Proceedings of the 2004 Haskell Workshop* (2004), ACM Press. doi:10.1145/1017472.1017482. 147

[FB96] Freed N., Borenstein N.: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045 (Draft Standard), Nov. 1996. Updated by RFCs 2184, 2231, 5335, 6532. URL: http://www.ietf.org/rfc/rfc2045.txt. 122

[FGM*99] Fielding R., Gettys J., Mogul J., Frystyk H., Masinter L., Leach P., Berners-Lee T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585. URL: http://www.ietf.org/rfc/rfc2616.txt. 4, 9, 170, 173

[FH11] Foley T., Hanrahan P.: Spark: Modular, Composable Shaders for Graphics Hardware. *ACM Trans. Graph. 30*, 4 (2011), 107:1—-107:12. doi:10.1145/2010324.1965002. 30, 147

[Fie00] Fielding R.: *Architectural Styles and the Design of Network-based Software Architectures*. Phd thesis, University of California, 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. 1, 164

[FM11] Fette I., Melnikov A.: The WebSocket Protocol. RFC 6455 (Proposed Standard), Dec. 2011. URL: http://www.ietf.org/rfc/rfc6455.txt. 4, 165

[FPV98] Fuggetta A., Picco G. P., Vigna G.: Understanding Code Mobility. *Software Engineering, IEEE Transactions on 24*, 5 (1998), 342–361. doi:10.1109/32.685258. 165

[FvDF13] Foley J. D., van Dam A., Feiner S. K.: *Computer Graphics: Principles and Practice (3rd Edition)*. Addison-Wesley Professional, 2013. 15

[Gar05] Garrett J.: Ajax : A New Approach to Web Applications How Ajax is Different. *experiencezencom* (2005), 1–5. URL: http://experiencezen.com/wp-content/uploads/2007/04/adaptive-path-ajax-a-new-approach-to-web-applications1.pdf. 12

[Gee09] Geelnard M.: Open Compressed Triangle Mesh file format, 2009. URL: http://openctm.sourceforge.net/. 82, 172, 175

[GHJV95] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995. 32, 43, 46, 97

[GMLP08] Gidén V., Moeller T., Ljung P., Paladini G.: Scene graph-based construction of CUDA kernel pipelines for XIP. In *Workshop on High-Performance Medical Image Computing and Computer Aided Intervention* (2008). 34

[Goe07] Goessner S.: JSONPath - XPath for JSON, 2007. URL: http://goessner.net/articles/JsonPath/. 167

[Goo02] Goodman D.: *Dynamic HTML: The Definitive Reference (2nd Edition)*. O'Reilly Media, 2002. 11

[Gra03] Grahn H.: DrawGroup & DrawOp, 2003. URL: http://www.bitmanagement.de/developer/contact/examples/multitexture/drawgroup.html. 42

[GS08] Georgiev I., Slusallek P.: RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. In *IEEE Symposium on Interactive Ray Tracing* (2008), pp. 115 – 122. doi:10.1109/RT.2008.4634631. 127

[GSKC10] Gritz L., Stein C., Kulla C., Conty A.: Open Shading Language. *ACM SIGGRAPH 2010 Talks* (2010), 33:1. doi:10.1145/1837026.1837070. 59, 100, 145, 148

[GTGB84] Goral C. M., Torrance K. E., Greenberg D. P., Battaile B.: Modeling the Interaction of Light Between Diffuse Surfaces. *SIGGRAPH Comput. Graph. 18*, 3 (1984), 213–222. doi:10.1145/964965.808601. 15

[HG12] Hackett B., Guo S.-y.: Fast and Precise Hybrid Type Inference for JavaScript. In *Proceedings of PLDI 2012* (2012), ACM, pp. 239–250. doi:10.1145/2254064.2254094. 153

[HHSS12] Herhut S., Hudson R. L., Shpeisman T., Sreeram J.: Parallel Programming for the Web. In *Proceedings of the 4th USENIX conference on Hot topics in parallelism* (2012), HotPar'12, USENIX Association. 6, 12, 13, 163

[HL90] Hanrahan P., Lawsont J.: A Language for Shading and Lighting Calculations. *SIGGRAPH Comput. Graph. 24*, 4 (1990), 289–298. doi:10.1145/97879.97911. 145, 146, 155, 158

[HMY12] Harada T., McKee J., Yang J. C.: Forward+: Bringing Deferred Lighting to the Next Level. In *Eurographics 2012 - Short Papers* (2012), Andujar C., Puppo E., (Eds.), The Eurographics Association. doi:10.2312/conf/EG2012/short/005-008. 17

[Hop96] Hoppe H.: Progressive Meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (1996), SIGGRAPH '96, pp. 99–108. doi:10.1145/237170.237216. 102, 175

[Hum06] Humanoid Animation Working Group: ISO/IEC 19774:2006, Humanoid Animation (H-Anim), 2006. URL: http://www.h-anim.org/. 47

[HW96] Hartman J., Wernecke J.: *The VRML 2.0 Handbook: Building Moving Worlds on the Web*. Addison-Wesley, 1996. 40

[ISO68] ISO: *ISO 8879:1986: Information processing — Text and office systems — Standard Generalized Markup Language (SGML)*. Tech. rep., International Organization for Standardization, 1968. 10

[ISO98] ISO: *ISO 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs) - Part II: Guidance on usability*. Tech. rep., 1998. 20

[ISO03] ISO: *ISO/IEC 14496-10:2003, Information technology – Coding of audio-visual objects – Part 10: Advanced Video Coding*. Tech. rep., International Organization for Standardization, 2003. 102

[ISO05] ISO: *ISO/IEC 14496-12:2005, Information technology – Coding of audio-visual objects – Part 12: ISO base media file format*. Tech. rep., International Organization for Standardization, 2005. 102

[ISO10a] ISO: *Ergonomics of human-system interaction - Part 210: Human-centred design for interactive systems (ISO 9241-210:2010(E))*. Tech. rep., 2010. 20

[ISO10b] ISO: *Information technology – Computer graphics and image processing – The Virtual Reality Modeling Language – Part 1: Functional specification and UTF-8 encoding*. Tech. rep., International Organization for Standardization, 2010. 40

[ISO12] ISO: ISO/IEC 14306:2012, Industrial automation systems and integration – JT file format specification for 3D visualization, 2012. 27

[Ist13] Istomina N.: *Volume Rendering in XML3D*. Master Thesis, Saarland University, 2013. 140, 143

[JB08] Jung Y., Behr J.: Extending H-anim and X3D for Advanced Animation Control. In *Proceedings of the 13th international symposium on 3D web technology - Web3D '08* (2008), ACM Press, pp. 57–65. doi:10.1145/1394209.1394224. 48

[Jen96] Jensen H. W.: Global Illumination Using Photon Maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96* (1996), Springer-Verlag, pp. 21–30. doi:10.1.1.36.8082. 15

[Jos06] Josefsson S.: The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), Oct. 2006. URL: http://www.ietf.org/rfc/rfc4648.txt. 174

[JRS13] Jankowski J., Ressler S., Sons K.: Declarative Integration of Interactive 3D Graphics into the World-Wide Web: Principles, Current Approaches, and Research Agenda. *Proceedings of the 18th International Conference on 3D Web Technology* (2013), 39–45. doi:10.1145/2466533.2466547. 14

[JS06] Junker G., Streeting S.: *Pro OGRE 3D programming*. The expert's voice in open source. Apress, 2006. 33, 78

[Kaj86] Kajiya J. T.: The Rendering Equation. In *ACM SIGGRAPH Computer Graphics* (1986), vol. 20, pp. 143–150. doi:10.1145/15886.15902. 15

[Kam10] Kamburelis M.: Shadow Maps and Projective Texturing in X3D. In *Proceedings of the 15th International Conference on Web 3D Technology* (2010), Web3D '10, ACM, pp. 17–26. doi:10.1145/1836049.1836052. 42

[KBR03] Kessenich J., Baldwin D., Rost R.: The OpenGL® Shading Language, 2003. URL: http://www.opengl.org/documentation/glsl/. 26, 59, 146

[KDK*01] Khailany B., Dally W. J., Kapasi U. J., Mattson P., Namkoong J., Owens J. D., Towles B., Chang A., Rixner S.: Imagine: Media Processing with Streams. *IEEE Micro 21*, 2 (2001), 35–46. doi:10.1109/40.918001. 35

[Khr07] Khronos Group: OpenGL ES Common Profile Specification - Version 2.0, 2007. URL: https://www.khronos.org/opengles/. 2, 17

[Khr08] Khronos Group: COLLADA - 3D Asset Exchange Schema, 2008. URL: https://www.khronos.org/collada/. 26

[Khr09] Khronos Group: WebGL, 2009. URL: https://www.khronos.org/webgl/. 2, 12, 49, 162

[Khr11] Khronos Group: Typed Array Specification, 2011. URL: https://www.khronos.org/registry/typedarray/specs/latest/. 76, 164

[Khr14a] Khronos Group: glTF - The GL Transmission Format, Draft, Version 1.0, 2014. URL: https://github.com/KhronosGroup/glTF/blob/master/specification/README.md. 27

[Khr14b] Khronos Group: WebCL Specification, 2014. URL: https://www.khronos.org/webcl/. 12, 163

[KLN*10] Kapahnke P., Liedtke P., Nesbigall S., Warwas S., Klusch M.: ISReal: An Open Platform for Semantic-Based 3D Simulations in the 3D Internet. In *The Semantic Web – ISWC 2010: 9th International Semantic Web Conference, Shanghai, China, Revised Selected Papers, Part II*. 2010, pp. 161–176. doi:10.1007/978-3-642-17749-1_11. 139

[KRS*13] Klein F., Rubinstein D., Sons K., Einabadi F., Herhut S., Slusallek P.: Declarative AR and Image Processing on the Web with Xflow. In *Proceedings of the 18th International Conference on 3D Web Technology* (2013), Web3D '13, ACM, pp. 157–165. doi:10.1145/2466533.2466544. 6, 91

[KS06] Kalkusch M., Schmalstieg D.: Extending the scene graph with a dataflow visualization system. In *Proceedings of the ACM symposium on Virtual reality software and technology* (2006), VRST '06, ACM, pp. 252–260. doi:10.1145/1180495.1180547. 35, 46

[KSJ*12] Klein F., Sons K., John S., Rubinstein D., Slusallek P., Byelozyorov S.: Xflow – Declarative Data Processing for the Web. In *Proceedings of the 17th International Conference on 3D Web Technology* (2012), ACM Press, pp. 37 – 45. doi:10.1145/2338714.2338719. 6, 164

[KSRS13] Klein F., Sons K., Rubinstein D., Slusallek P.: XML3D and Xflow: Combining Declarative 3D for the Web with Generic Data Flows. *IEEE Computer Graphics and Applications 33*, 5 (2013), 38–47. doi:10.1109/MCG.2013.67. 6, 89

[KSSS14] Klein F., Spieldenner T., Sons K., Slusallek P.: Configurable Instances of 3D Models for Declarative 3D in the Web. In *Proceedings of the 19th International Conference on 3D Web Technologies* (2014), Web3D '14, ACM, pp. 71–79. doi:10.1145/2628588.2628594. 6, 93, 94

[KW03] Krüger J., Westermann R.: Acceleration techniques for GPU-based volume rendering. *IEEE Visualization, 2003. VIS 2003.* (2003). doi:10.1109/VISUAL.2003.1250384. 140

[KW09] Kuck R., Wesche G.: A Framework for Object-Oriented Shader Design. In *Proceedings of ISVC 2009* (2009), Springer-Verlag, pp. 1019–1030. doi:10.1007/978-3-642-10331-5_95. 146

[LAC*92] Lucas B., Abram G. D., Collins N. S., Epstein D. A., Gresh D. L., McAuliffe K. P.: An Architecture for a Scientific Visualization System. In *Proceedings of the 3rd Conference on Visualization '92* (1992), VIS '92, IEEE Computer Society Press, pp. 107–114. 35

[Lam60] Lambert J. H.: *Photometria sive de mensure de gratibus luminis, colorum umbrae*. Eberhard Klett, 1760. 26

[LB99] Lie H. W., Bos B.: *Cascading Style Sheets: Designing for the Web (2nd Edition)*. Addison-Wesley Professional, 1999. 11

[LC87] Lorensen W. E., Cline H. E.: Marching cubes: A high resolution 3D surface construction algorithm, 1987. doi:10.1145/37402.37422. 140

[LCD13] Lavoué G., Chevalier L., Dupont F.: Streaming Compressed 3D Data on the Web Using JavaScript and WebGL. In *Proceedings of the 18th International Conference on 3D Web Technology* (2013), Web3D '13, ACM, pp. 19–27. doi:10.1145/2466533.2466539. 175

[Lev88] Levoy M.: Display of surfaces from volume data. *IEEE Computer Graphics and Applications 8*, 3 (1988), 29–37. doi:10.1109/38.511. 140

[LH91] Laur D., Hanrahan P.: Hierarchical splatting: a progressive refinement algorithm for volume rendering. *ACM SIGGRAPH Computer Graphics 25*, 4 (1991), 285–288. doi:10.1145/127719.122748. 140

[Lie05] Lie H. W.: *Cascading Style Sheets*. PhD thesis, University of Oslo, Faculty of Mathematics and Natural Sciences, 2005. 10

[LJBA13] Limper M., Jung Y., Behr J., Alexa M.: The POP Buffer: Rapid Progressive Clustering by Geometry Quantization. *Computer Graphics Forum 32*, 7 (2013), 197–206. doi:10.1111/cgf.12227. 53, 175, 198

[LL94] Lacroute P., Levoy M.: Fast Volume Rendering Using a Shear-warp Factorization of the Viewing Transformation. *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '94* (1994), 451–458. doi:10.1145/192161.192283. 140

[LP10] Lifton J., Paradiso J. A.: Dual reality: Merging the real and virtual. In *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering* (2010), vol. 33 LNICST, pp. 12–28. doi:10.1007/978-3-642-11743-5_2. 37

[LS02] Lalonde P., Schenk E.: Shader-driven Compilation of Rendering Assets. *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques 21*, 3 (2002), 713–720. doi:10.1145/566654.566641. 148

[LS11] Lawson B., Sharp R.: *Introducing HTML5 (2nd Edition) (Voices That Matter)*. New Riders, 2011. 12

[LSSS16] Lemme S., Sutter J., Schlinkmann C., Slusallek P.: The Basic Building Blocks of Declarative 3D on the Web. In *Proceedings of the 21st International Conference on 3D Web Technology* (2016), Web3D '16, ACM, pp. 17–25. doi:10.1145/2945292.2945303. 54, 98, 106, 198

[LTBF14] Limper M., Thöner M., Behr J., Fellner D. W.: SRC - A Streamable Format for Generalized Web-based 3D Data Transmission. In *Proceedings of the 19th International Conference on 3D Web Technologies* (2014), Web3D '14, ACM Press, pp. 35–43. doi:10.1145/2628588.2628589. 53

[LW93] Lafortune E. P., Willems Y. D.: Bi-Directional Path Tracing. In *Proceedings of Compugraphics '93* (1993), pp. 145–153. 15

[Mam13] Mamou K.: Open 3D Graphics Compression (Open3DGC), 2013. URL: https://github.com/amd/rest3d/tree/master/server/o3dgc. 82, 172, 176

[Mat05] Matroska: Matroška Media Container, 2005. URL: http://matroska.org/technical/specs/index.html. 168

[MCC11] McLaughlin T., Cutler L., Coleman D.: Character Rigging, Deformations, and Simulations in Film and Game Production. In *ACM SIGGRAPH 2011 Courses* (2011), SIGGRAPH '11, ACM, pp. 5:1—-5:18. doi:10.1145/2037636.2037641. 48

[McG05] McGuire M.: *The SuperShader*. 2005, ch. 8.1, pp. 485–498. 144

[Men10] Mental Images: Design Specification for MetaSL® 1.1, 2010. URL: http://www.nvidia-arc.com/fileadmin/user_upload/PDF/MetaSL_spec_1.1.6.pdf. 147

[MFM*14] McCutchan J., Feng H., Matsakis N., Anderson Z., Jensen P.: A SIMD Programming Model for Dart, Javascript, and Other Dynamically Typed Scripting Languages. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing* (2014), WPMVP '14, ACM, pp. 71–78. doi:10.1145/2568058.2568066. 13, 38

[MGAK03] Mark W. R., Glanville R. S., Akeley K., Kilgard M. J.: Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Trans. Graph. 22* (2003), 896–907. 146

[Mic02] Microsoft: DirectX HLSL Shader Model 1, 2002. URL: `http://msdn.microsoft.com`. 26, 146

[Mic08] Microsoft: DirectX HLSL Shader Model 5, 2008. URL: `http://msdn.microsoft.com`. 146

[Mic12] Microsoft: DirectShow, 2012. URL: `http://msdn.microsoft.com`. 35

[Mil09] Miller J.: Convention Over Configuration, 2009. URL: `http://msdn.microsoft.com/en-us/magazine/dd419655.aspx`. 106

[Moz13] Mozilla: *asm.js – Working Draft*. Tech. rep., 2013. URL: `http://asmjs.org/`. 38, 164

[Moz15] Mozilla: Gecko Plugin API Reference, 2015. URL: `https://developer.mozilla.org/en-US/Add-ons/Plugins/Reference`. 136

[MQP02] McCool M. D., Qin Z., Popa T. S.: Shader Metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (2002), HWWS '02, Eurographics Association, pp. 57–68. 144, 146

[MRR12] McCool M., Reinders J., Robison A.: *Structured parallel programming patterns for efficient computation*. 2012. `doi:10.1016/B978-008043924-2/50055-9`. 13

[MSPK06] McGuire M., Stathis G., Pfister H., Krishnamurthi S.: Abstract Shade Trees. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games* (2006), I3D '06, ACM, pp. 79–86. `doi:10.1145/1111411.1111425`. 147, 149

[MV94] Murray J. D., VanRyper W.: *Encyclopedia of graphics file formats*. O'Reilly Software Series. O'Reilly & Associates, Inc., 1994. 168

[Nie93] Nielsen J.: *Usability Engineering*, vol. 44. Morgan Kaufmann Publishers Inc., 1993. 20, 163

[NNH99] Nielson F., Nielson H. R., Hankin C.: *Principles of Program Analysis*. Springer, 1999. `doi:10.1007/978-3-662-03811-6`. 153

[Nvi10] Nvidia: NVIDIA OptiX Application Acceleration Engine, 2010. URL: `https://developer.nvidia.com/optix`. 19

[Nvi12] Nvidia: NVIDIA Material Definition Language, 2012. URL: `http://www.mdlhandbook.com/`. 147

[OA11] Olsson O., Assarsson U.: Tiled shading. *Journal of Graphics, GPU, and Game Tools 15*, 4 (2011), 235–251. `doi:10.1080/2151237X.2011.621761`. 17

[OBA12] Olsson O., Billeter M., Assarsson U.: Clustered deferred and forward shading. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics* (2012), EGGH-HPG'12, Eurographics Association, pp. 87–96. doi:10.2312/EGGH/HPG12/087-096. 17

[OKK09] Orthmann J., Keller M., Kolb A.: Integrating GPGPU Functionality into Scene Graphs. In *Vision, Modeling & Visualization* (2009), DNB, pp. 233–244. 34

[ON94] Oren M., Nayar S. K.: Generalization of Lambert's Reflectance Model. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques* (1994), SIGGRAPH '94, ACM, pp. 239–246. doi:10.1145/192161.192213. 152

[O'R05] O'Reilly T.: What is Web 2.0: Design patterns and business models for the next generation of software., 2005. URL: http://oreilly.com/web2/archive/what-is-web-20.html. 1, 12

[Pag08] Pagani M.: *Encyclopedia of Multimedia Technology and Networking*, 2nd ed. Information Science Reference - Imprint of: IGI Publishing, Hershey, PA, 2008. 164

[PBMH02] Purcell T. J., Buck I., Mark W. R., Hanrahan P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (2002). doi:10.1145/566654.566640. 16

[Pel05] Pellacini F.: User-configurable Automatic Shader Simplification. *ACM Transactions on Graphics 24* (2005), 445–452. doi:10.1145/1073204.1073212. 147, 198

[PH10] Pharr M., Humphreys G.: *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann, 2010. 15

[Pho75] Phong B. T.: Illumination for computer generated pictures. *Communications of the ACM 18* (1975), 311–317. doi:10.1145/360825.360839. 26, 99, 152

[PKP94] Pesce M., Kennard P., Parisi A.: Cyberspace. In *First International Conference on the World-Wide Web* (1994). URL: http://www94.web.cern.ch/WWW94/PrelimProcs.html. 40

[PMTH01] Proudfoot K., Mark W. R., Tzvetkov S., Hanrahan P.: A Real-time Procedural Shading System for Programmable Graphics Hardware. In *Proceedings of SIGGRAPH 2001* (2001), ACM, pp. 159–170. 147, 148

[Rag94] Raggett D.: Extending WWW to support platform independent virtual reality. In *First International Conference on the World-Wide Web* (1994). URL: http://www.w3.org/People/Raggett/vrml/vrml.html. 40

[Rei02] Reiners D.: *OpenSG*. PhD thesis, TU Darmstadt, Fachbereich Informatik, 2002. 33, 119

[Rei07] Reinders J.: *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*, vol. 23. O'Reilly Media, 2007. 127

[RGSS09] Rubinstein D., Georgiev I., Schug B., Slusallek P.: RTSG: Ray Tracing for X3D via a Flexible Rendering Framework. In *Proceedings of the 14th International Conference on 3D Web Technology* (2009), Web3D '09, ACM, pp. 43–50. doi:10.1145/1559764.1559771. 42, 119, 127

[RH94] Rohlf J., Helman J.: IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques - SIGGRAPH '94* (1994), ACM Press, pp. 381–394. doi:10.1145/192161.192262. 31, 33, 119

[RLV*04] Riffel A., Lefohn A. E., Vidimce K., Leone M., Owens J. D.: Mio: Fast Multipass Partitioning via Priority-based Instruction Scheduling. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (2004), HWWS '04, ACM, pp. 35–44. doi:10.1145/1058129.1058135. 147

[RLZ10] Ratanaworabhan P., Livshits B., Zorn B. G.: JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. *WebApps 10* (2010), 3–3. 102

[RRA10] Rocha R. V., Rocha R. V., Araújo R. B.: Selecting the Best Open Source 3D Games Engines. In *Proceedings of the Brazilian Symposium on Computer Games and Digital Entertainment* (2010), pp. 333–336. 187

[RRT95] Ramey D., Rose L., Tyerman L.: MTL material format (Lightwave, OBJ), 1995. URL: http://paulbourke.net/dataformats/mtl/. 26

[SA94] Segal M., Akeley K.: *The Design of the OpenGL Graphics Interface*. Tech. rep., Silicon Graphics Computer Systems, 1994. URL: http://www.graphics.stanford.edu/courses/cs448a-01-fall/design_opengl.pdf. 16

[SAK15] Schiewe A., Anstoots M., Krüger J.: State of the Art in Mobile Volume Rendering on iOS Devices. In *Eurographics Conference on Visualization (EuroVis) - Short Papers* (2015), The Eurographics Association, pp. 139–143. doi:10.2312/eurovisshort.20151139. 192

[SAMWL11] Sitthi-Amorn P., Modly N., Weimer W., Lawrence J.: Genetic Programming for Shader Simplification. *ACM Transactions on Graphics 30* (2011), 152:1—–152:12. doi:10.1145/2024156.2024186. 147

[SC92] Strauss P. S., Carey R.: An Object-Oriented 3D Graphics Toolkit. *ACM SIGGRAPH Computer Graphics 26*, 2 (1992), 341–349. doi:10.1145/142920.134089. 119

[Sch06] Schmittler J.: *SaarCOR — A Hardware Architecture for Ray Tracing*. PhD thesis, Saarland University, Saarbrücken, Germany, 2006. 16

[SDHS13] Sons K., Demme G., Herget W., Slusallek P.: Fortress City Saarlouis: Development of an interactive 3D City Model using Web Technologies. In *Proceedings of the CAA 2013 Conference Across Space and Time* (2013), Traviglia A., (Ed.). 179

[SJBF10] Schwenk K., Jung Y., Behr J., Fellner D. W.: A Modern Declarative Surface Shader for X3D. In *Proceedings of the 15th International Conference on Web 3D Technology* (2010), Web3D '10, ACM Press, pp. 7–16. doi:10.1145/1836049.1836051. 48, 53, 144

[SKR*10] Sons K., Klein F., Rubinstein D., Byelozyorov S., Slusallek P.: XML3D: Interactive 3D Graphics for the Web. In *Proceedings of the 15th International Conference on 3D Web Technologies* (2010), Web3D '10, ACM, pp. 175–184. doi:10.1145/1836049.1836076. 6

[SKSS14] Sons K., Klein F., Sutter J., Slusallek P.: shade.js: Adaptive Material Descriptions. *Computer Graphics Forum 33*, 7 (2014). doi:10.1111/cgf.12473. 6, 99, 145

[SKSS15] Sons K., Klein F., Sutter J., Slusallek P.: The XML3D Architecture. In *ACM SIGGRAPH 2015 Posters* (2015), SIGGRAPH '15, ACM, pp. 39:1–39:1. doi:10.1145/2787626.2792623. 109

[SML96] Schroeder W. J., Martin K. M., Lorensen W. E.: The Design and Implementation Of An Object-Oriented Toolkit For 3D Graphics And Visualization. In *Proceedings of the 7th conference on Visualization '96* (1996), VIS '96, IEEE, pp. 93–ff. 35

[SMZB14] Stamoulias A., Malamos A. G., Zampoglou M., Brutzman D.: Enhancing X3DOM Declarative 3D with Rigid Body Physics Support. In *Proceedings of the 19th International Conference on 3D Web Technologies* (2014), Web3D '14, ACM, pp. 99–107. doi:10.1145/2628588.2628602. 130

[SS11a] Sons K., Slusallek P.: XML3D Physics: Declarative Physics Simulation for the Web. In *Workshop on Virtual Reality Interaction and Physical Simulation* (2011), VRIPHYS, Eurographics Association, pp. 55–63. doi:10.2312/PE/vriphys/vriphys11/055-063. 6, 84, 129

[SS11b] Stocker H., Schickel P.: X3D Binary Encoding Results for Free Viewpoint Networked Distribution and Synchronization. In *Proceedings of the 16th International Conference on 3D Web Technology* (2011), Web3D '11, ACM, pp. 67–70. doi:10.1145/2010425.2010437. 175

[SS13] Sons K., Slusallek P.: *Towards a 3D Transmission Format for the Web*. Tech. rep., 9th International AR Standards Community Meeting, 2013. URL: http://www.perey.com/ARStandards/[Klein]3dtf-position-paper_Ninth_AR_Standards_Meeting.pdf. 6, 162

[SSBM03] Saeger M., Sons K., Böttger A., Mank A.: *Machbarkeitsstudie für eine Schnittstelle zwischen Open Inventor und OpenRT*. Tech. rep.,

2003. URL: https://graphics.cg.uni-saarland.de/fileadmin/cguds/papers/2003/projektbericht.pdf. 34

[SSK*13] Sons K., Schlinkmann C., Klein F., Rubinstein D., Slusallek P.: xml3d.js: Architecture of a Polyfill implementation of XML3D. In *Proceedings of the 6th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)* (2013), IEEE, pp. 17–24. doi:10.1109/SEARIS.2013.6798104. 6, 116

[SSK16] Sons K., Schlinkmann C., Klein F.: XML3D Specification, 2016. URL: http://www.xml3d.org/xml3d/specification/latest/. 65, 70, 71, 72, 76, 81, 86, 88, 99, 116

[SSS14] Sutter J., Sons K., Slusallek P.: Blast: A Binary Large Structured Transmission Format for the Web. In *Proceedings of the 19th International Conference on 3D Web Technologies* (2014), Web3D '14, ACM, pp. 45–52. doi:10.1145/2628588.2628599. 6, 162

[SSS15] Sutter J., Sons K., Slusallek P.: A CSS Integration Model for Declarative 3D. In *Proceedings of the 20th International Conference on 3D Web Technology* (2015), Web3D '15, ACM, pp. 209–217. doi:10.1145/2775292.2775295. 6, 198

[SSW06] Shirley P., Slusallek P., Wald I.: State of the Art in Interactive Ray Tracing, 2006. doi:10.1145/1185657.1185681. 19

[ST90] Saito T., Takahashi T.: Comprehensible Rendering of 3-D Shapes. *ACM SIGGRAPH Computer Graphics 24* (1990), 197–206. doi:10.1145/97880.97901. 17, 30, 156

[Sto06] Stoy C.: Game object component system. In *Game Programming Gems 6*. Charles River Media, 2006, pp. 393–403. 54

[Str93] Strauss P. S.: IRIS Inventor, a 3D graphics toolkit. *ACM SIGPLAN Notices 28*, 10 (1993), 192–200. doi:10.1145/167962.165889. 31, 33

[SVB*13] Schwenk K., Voss G., Behr J., Jung Y., Limper M., Herzig P., Kuijper A.: Extending a Distributed Virtual Reality System with Exchangeable Rendering Back-ends. *Vis. Comput. 29*, 10 (2013), 1039–1049. doi:10.1007/s00371-013-0836-y. 42

[Tel05] Telecommunication Standardization Sector of ITU: Information technology – Generic applications of ASN.1: Fast Infoset. 41, 175

[TS16] Tamm G., Slusallek P.: Web-enabled Server-based and Distributed Real-time Ray-Tracing. In *Eurographics Symposium on Parallel Graphics and Visualization* (2016), Eurographics Association. doi:10.2312/pgv.20161182. 185, 186

[UA06] Unicode Consortium, Allen J.: *The Unicode Standard, Version 5.0*, fifth ed. Addison-Wesley Professional, 2006. 162

[UFK*89] Upson C., Faulhaber Jr. T., Kamins D., Laidlaw D. H., Schlegel D., Vroom J., Gurwitz R., van Dam A.: The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Comput. Graph. Appl. 9*, 4 (1989), 30–42. doi:10.1109/38.31462. 35

[Ups89] Upstill S.: *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., 1989. 34, 145

[Vea98] Veach E.: *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, 1998. 15

[W3C00] W3C: Document Object Model Level 2 Events Specification, 2000. URL: http://www.w3.org/TR/DOM-Level-2-Events/. 11, 58, 64, 71

[W3C04] W3C: Resource Description Framework (RDF): Concepts and Abstract Syntax, 2004. URL: http://www.w3.org/TR/rdf-concepts/. 139

[W3C08] W3C: Extensible Markup Language (XML) 1.0 (Fifth Edition), 2008. URL: http://www.w3.org/TR/xml/. 10, 174

[W3C09a] W3C: Cascading Style Sheets, 2009. URL: http://www.w3.org/Style/CSS/. 1, 3, 9, 10, 58, 62

[W3C09b] W3C: CSS 3D Transforms Module Level 3, 2009. URL: http://www.w3.org/TR/css-transforms-1/. 66, 67

[W3C09c] W3C: Document Object Model, 2009. URL: http://www.w3.org/DOM/. 1, 9, 11, 63

[W3C09d] W3C: Scalable Vector Graphics, 2009. URL: http://www.w3.org/Graphics/SVG/. 11, 51

[W3C09e] W3C: The Web Sockets API, 2009. URL: http://www.w3.org/TR/websockets/. 102

[W3C09f] W3C: XMLHttpRequest, 2009. URL: http://www.w3.org/TR/XMLHttpRequest/. 1, 4, 12, 162, 165

[W3C11] W3C: Efficient XML Interchange (EXI) Format 1.0, 2011. URL: http://www.w3.org/TR/exi/. 175

[W3C12a] W3C: RDFa Core 1.1, 2012. URL: http://www.w3.org/TR/rdfa-core/. 53, 139

[W3C12b] W3C: Web Audio API – Working Draft, 2012. URL: http://www.w3.org/TR/webaudio/. 139, 162

[W3C13a] W3C: CSS Transitions, 2013. URL: https://www.w3.org/TR/css3-transitions/. 69

[W3C13b] W3C: Timing control for script-based animations - W3C Candidate Recommendation, 2013. URL: http://www.w3.org/TR/animation-timing/. 124

[W3C13c] W3C: Touch Events, 2013. URL: https://www.w3.org/TR/touch-events/. 71

[W3C14a] W3C: Custom Elements, W3C Working Draft (work in progress). http://www.w3.org/TR/2014/WD-custom-elements-20141216/, 2014. URL: http://www.w3.org/TR/2014/WD-custom-elements-20141216/. 14

[W3C14b] W3C: HTML Imports, W3C Working Draft (work in progress). http://www.w3.org/TR/2014/WD-html-imports-20140311/, 2014. URL: http://www.w3.org/TR/2014/WD-html-imports-20140311/. 14

[W3C14c] W3C: HTML5 - Candidate Recommendation, 2014. URL: http://www.w3.org/TR/html5/. 1, 9, 11, 14, 53, 61, 62, 64, 68

[W3C14d] W3C: Shadow DOM, W3C Working Draft (work in progress). http://www.w3.org/TR/2014/WD-shadow-dom-20140617/, 2014. URL: http://www.w3.org/TR/2014/WD-shadow-dom-20140617/. 14

[W3C14e] W3C: Streams API, 2014. URL: http://www.w3.org/TR/streams-api/. 12, 102, 165

[W3C15a] W3C: Shadow DOM, 2015. URL: http://www.w3.org/TR/shadow-dom/. 118

[W3C15b] W3C: Web Workers - W3C Working Draft, 2015. URL: http://www.w3.org/TR/workers/. 12, 102, 163, 166

[Wal04] Wald I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. 15

[War89] Ward, J G.: The RADIANCE Lighting Simulation and Rendering System. *21st Annual Conference on Computer graphics and Interactive Techniques* (1989), 459–472. doi:10.1145/192161.192286. 34

[War92] Ward G. J.: Measuring and Modeling Anisotropic Reflection. *SIGGRAPH Comput. Graph. 26*, 2 (1992), 265–272. doi:10.1145/142920.134078. 152

[Web04a] Web3D Consortium: ISO/IEC 19775:2004, Extensible 3D (X3D), 2004. URL: http://www.web3d.org/documents/specifications/19775-1/V3.0/index.html. 3, 41, 42, 46, 47, 65, 66, 91

[Web04b] Web3D Consortium: ISO/IEC 19776-1:2005, X3D encodings - XML, 2004. URL: http://www.web3d.org/documents/specifications/19776-1/V3.0/index.html. 51

[Web05] Web3D Consortium: ISO/IEC 19775-2, Extensible 3D (X3D) – Part 2: Scene Access Interface (SAI), 2005. URL: http://www.web3d.org/documents/specifications/19775-2/V3.0/index.html. 41

[Web06] Web3D Consortium: ISO/IEC 19775:2004/Am1:2006, X3D Architecture and Base Components - Amendment 1: Additional Functionality, 2006. URL: http://www.web3d.org/documents/specifications/19775-1/V3.1/index.html. 42

[Web07] Web3D Consortium: ISO/IEC 19776-3:2007, X3D encodings - Compressed Binary Encoding, 2007. URL: http://www.web3d.org/documents/specifications/19776-3/V3.1/index.html. 175

[Web08] Web3D Consortium: ISO/IEC 19775-1.2:2008, 37 Rigid body physics component, 2008. URL: http://www.web3d.org/documents/specifications/19775-1/V3.2/Part01/components/rigid_physics.html. 130

[Web16] WebAssembly Community Group: WebAssembly, 2016. URL: https://webassembly.github.io/. 38

[Whi79] Whitted T.: An Improved Illumination Model for Shaded Display, 1979. doi:10.1145/965103.807419. 15, 26

[WWB*14] Wald I., Woop S., Benthin C., Johnson G. S., Ernst M.: Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Trans. Graph. 33*, 4 (2014), 143:1—-143:8. doi:10.1145/2601097.2601199. 180, 186

[WYY*14] Wang R., Yang X., Yuan Y., Chen W., Bala K., Bao H.: Automatic Shader Simplification Using Surface Signal Approximation. *ACM Transactions on Graphics 33*, 6 (2014), 1–11. doi:10.1145/2661229.2661276. 147

[Zak11] Zakai A.: Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (2011), OOPSLA '11, ACM, pp. 301–312. doi:10.1145/2048147.2048224. 38, 138

[ZSC*13] Zinnikus I., Spieldenner T., Cao X., Klusch M., Krauss C., Nonnengart A., Slusallek P.: A Collaborative Virtual Workspace for Factory Configuration and Evaluation. In *Proceedings of the 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom-13)* (2013), o.A. 36, 181