The Basic Building Blocks of Declarative 3D on the Web

Stefan Lemme*
DFKI
Saarland University

Jan Sutter†
DFKI
Saarland University
Intel VCI

Christian Schlinkmann‡
DFKI

Philipp Slusallek§
DFKI
Saarland University
Intel VCI

Abstract

WebGL enabled real-time 3D graphics on the Web. With the objective to integrate 3D graphics into the rest of the Web technology stack, and to make it easier for Web developers to develop interactive 3D graphics, Declarative 3D approaches were developed: X3DOM and XML3D. While the former focuses on backward-compatibility to X3D and a large set of convenience elements, the latter attempts to define a minimal set of flexible elements as an extension to HTML5.

It has now been more than 6 years since Declarative 3D was first proposed for the Web. However, despite their different philosophies neither X3DOM nor XML3D has yet been able to achieve the same momentum and adoption rate as imperative frameworks like three.js. In the meantime, the underlying Web technology stack has made significant advances.

In this paper we revisit both approaches in light of new Web technologies, such as Web Components, to define a small set of core elements that can provide the convenience of X3DOM while remaining as flexible and customizable as XML3D. Further, we present a strategy for building upon these core elements to enable user-defined elements, with the ability to cover domain-specific needs in Declarative 3D. Lastly, we show how these concepts can be used to simplify existing approaches (i.e. X3DOM and XML3D) and provide the basic building blocks of Declarative 3D on the Web.

Keywords: HTML, Dec3D, Web Components, X3DOM, XML3D

Concepts: •**Computing methodologies** → **Graphics systems and interfaces;** *Graphics file formats; Rendering;*

1 Introduction

The HTML `<canvas>` element has introduced an *imperative* interface to produce 3D renderings on the Web via WebGL. However, the core content technologies, HTML and CSS, are declarative. *Declarative 3D for the Web (Dec3D)* [Jankowski et al. 2013] attempts to extend HTML with a declarative option for 3D graphics that integrates with the existing Web technology stack to be compatible with the DOM, events, and CSS. X3DOM [Behr et al. 2009] and XML3D [Sons et al. 2010], the two most notable approaches, provide Web developers with new HTML elements to embed interactive 3D content into their Web pages. Both also integrate with JavaScript to create dynamic 3D content using the DOM API.

Figure 1 shows the integration levels for 3D graphics on the Web defined by Jankowski et al. [2013]. The first level is defined to be on par with WebGL, an imperative API to define 3D content. This is the level of integration provided by libraries such as three.js[1]. Approaches at the second integration level have the 3D scene description integrated into the DOM. This level of integration is achieved by both X3DOM and XML3D. The third and fourth level of integration require CSS transformations, debugging functionality, and CSS based material descriptions. A CSS integration for declarative 3D on the Web has recently been proposed by Sutter et al. [2015], which provides the necessary means to achieve the third and fourth level of integration.

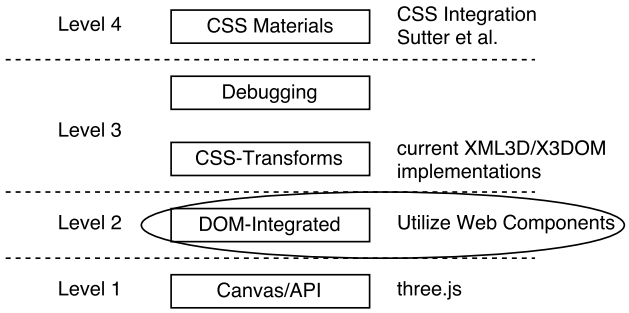

Figure 1: *Integration levels for 3D content on the Web according to Jankowski et al. [2013]. Our contribution will focus on the DOM integration taking into account recent Web technologies like Web Components.*

The DOM integration (Level 2 in Figure 1) provided by XML3D and X3DOM includes specific 3D-related HTML elements and events. Both approaches rely on polyfill implementations [Sons et al. 2010; Behr et al. 2010] based on JavaScript and WebGL. As a result, the integration of XML3D and X3DOM, including the chosen elements and the level of CSS integration, were heavily shaped by the available Web technologies at that time.

While X3DOM and XML3D both provide a DOM-integrated scene description, both approaches differ considerably: X3DOM is focused on backward compatibility to X3D and, thus, integrates a subset of X3D's abstract model as well as many of X3D's concepts. X3DOM, due to its compatibility with X3D, provides highly specialized nodes. They make it easier for beginners to create a 3D scene from scratch. However, those highly specialized nodes also make it hard for Web developers to extend X3DOM with application-specific functionality as X3D prototypes are not supported by X3DOM. XML3D, on the other hand, attempts to keep the number of 3D-specific DOM elements minimal, providing only a generic set of elements while leaving application-related functionality to JavaScript and data processing to Xflow [Klein et al. 2012]. However, this generic approach is more difficult for Web developers to start with due to the lack of convenience elements and domain-specific functionality.

Both approaches have not managed to gain the momentum for a wide adoption in the Web developer community. On the other hand, libraries such as three.js that do not provide any integration with the

*e-mail:stefan.lemme@dfki.de
†e-mail:jan.sutter@dfki.de
‡e-mail:christian.schlinkmann@dfki.de
§e-mail:philipp.slusallek@dfki.de

Web3D '16,, July 22-24, 2016, Anaheim , CA, USA
ISBN: 978-1-4503-4428-9/16/07
DOI: http://dx.doi.org/10.1145/2945292.2945303

[1] http://threejs.org

existing Web technology stack beyond the first Level (see Figure 1) have a large community and strong momentum. We maintain that Dec3D is an important part of the future of 3D graphics on the Web and its adoption should be the primary goal of the Declarative 3D community, before considering standardization efforts or native implementations in the browsers. To increase the adoption, we identified that two major aspects must be considered: Usability and extensibility.

Usability in the Dec3D context should be focused on Web developers without an extensive background in 3D graphics. The highly specialized nodes of X3DOM are useful to provide the convenience that such developers need to quickly start to incorporate 3D renderings into a Web page with minimal effort.

Extensibility is the other important factor for adoption. A prime example of this is the prototype system of X3D. Developers demand the ability to extend existing elements as well as create new functionality on top the already provided elements. This is something that XML3D provides with its generic as well as flexible approach.

Neither X3DOM nor XML3D covers both aspects to the degree necessary for a wider adoption of Dec3D on the Web. In order to fulfill the requirements of usability as well as extensibility we need an approach that provides the same convenience that X3DOM offers while remaining as extensible and configurable as XML3D. Consequently, we have decided to reevaluate the second level of integration of Dec3D for the Web, the DOM integration level, taking into account recent Web technologies that were not available during the initial design of XML3D and X3DOM.

In the following, we identify the minimum set of necessary new HTML elements that are required to provide a declarative way for integrating 3D graphics into a Web page. These *core elements* will follow the philosophy of XML3D and offer the minimum set of required functionality to embed 3D graphics. To provide the necessary convenience we show how this set of core elements, together with the concepts of Xflow and shade.js, can be used as building blocks for user-defined elements by leveraging the power of Web Components. Such elements may cover domain-specific functionality or provide additional convenience through an easy-to-use interface. In addition, we will demonstrate how existing nodes in X3DOM and also some of the nodes of XML3D can be implemented using our unifying approach.

2 Related Work

2.1 X3D and X3DOM

X3D is an ISO Standard [ISO 2008] file format to represent 3D scenes. Its optional XML encoding is backward compatible to VRML97. In addition to the 3D scene description it defines a full runtime environment including an event system and scripting that are not compatible with the Web.

X3DOM [Behr et al. 2009] is an approach to embed a subset of X3D scenes into the DOM of a Web page. Similar to SVG [W3C 2011], the 3D rendering should be displayed without the need to install a plug-in. To integrate X3D into the DOM, its functionality was stripped down to visualization components while dynamics, distribution, security, and scripting are managed through existing Web technologies provided by the browser.

X3DOM attempts to add an existing 3D graphics format to the Web, rather than extending the current Web technology where necessary. This way, existing X3D content can be reused in the browser - as long as it does not exceed the proposed DOM profile, which is only a subset of X3D. Although X3DOM is an obvious approach to integrate X3D into the DOM, it is sometimes counter-intuitive for Web developers as it keeps many of the VRML/X3D concepts, for instance DEF/USE and routing.

2.2 XML3D and Xflow

XML3D [Sons et al. 2010] is a declarative approach for 3D graphics designed as a minimal extension to HTML5. It defines very lightweight scene graph elements, such as meshes, groups, materials, light sources, and viewpoints. The general design of XML3D is focused around the data structures to provide a very generic scene graph that is independent of the underlying rendering system. In contrast to the highly specialized X3D/X3DOM nodes that each have custom data structures, XML3D elements are built around generic data blocks with typed buffers using the `<data>` element [Klein et al. 2012]. This nicely matches the design of today's graphics hardware and APIs that support programmable shaders and work on generic input buffers. With this approach XML3D can easily take advantage of programmable GPUs [Klein et al. 2013].

Similar to HTML for 2D, XML3D provides only elements to describe a static 3D scene. Dynamic changes to the scene are handled by application logic handled in JavaScript. Additionally, XML3D takes advantage of Xflow for declarative dataflow processing that also integrates into the DOM. Instead of covering functionality in a use-case-oriented and specialized way, Xflow provides data processing graphs in a generic way, just as XML3D provides a generic approach to scene descriptions.

2.3 shade.js

A key objective of Dec3D has always been about the independence from the concrete rendering approach. For X3D material definitions typically come in the form of predefined "uber-shaders", such as the *CommonSurfaceShader* proposal [Schwenk et al. 2010]. These material models have a well-known set of input parameters that can be changed to configure the appearance of the material. This, however, makes it impossible to use material appearances that go beyond the built-in models. Custom material models cannot be defined this way.

shade.js [Sons et al. 2014] addresses this issue and provides a way to describe material models in a renderer-agnostic way. It uses a domain-specific language (i.e. a dialect of JavaScript) and a compiler framework that translates shade.js material definitions to the targeted rendering approach and API (e.g. WebGL, ray and path-tracer, etc.).

2.4 Web Components

Web Components is an umbrella term for a set of four new Web technologies: Custom Elements [W3C 2014b] for defining new HTML elements, HTML Templates (which are part of HTML5 [W3C 2014d]) for defining reusable mark-up as DOM fragments, Shadow DOM [W3C 2014e] to encapsulate, scope, and hide DOM content, and HTML Imports [W3C 2014c] to reuse existing HTML documents. Together, these Web technologies provide means for encapsulation of functionality similar to OOP or the façade pattern in software engineering. It provides reusability of a once defined behavior in multiple contexts, and composability for constructing complex (application) components from smaller general-purpose components. This nicely addresses the requirement of extensibility for Dec3D, and furthermore, allows for streamlining the set of core elements to its essential building blocks.

```
<xml3d view="#camera">
  <mesh material="#env" src="cube.json"></mesh>

  <material id="env"
            model="urn:xml3d:material:diffuse">
    <data compute="genCubeMap()">
      <texture name="front">
        <img src="front.jpg"/>
      </texture>
      <!-- same for back, left, right, top -->
      <texture name="bottom">
        <img src="bottom.jpg"/>
      </texture>
    </data>
    <float3 name="emissiveColor">1 1 1</float3>
    <float3 name="diffuseColor">0 0 0</float3>
  </material>

  <view id="camera"></view>
</xml3d>
```

Listing 1: *Scene background with XML3D using six static images.*

3 Limitations of X3DOM and XML3D

As a running example of the usability and extensibility requirements in the context of Dec3D we examine the declarative approach to define a background for a 3D scene. This is a common task in 3D graphics, often achieved by wrapping the scene into a cube with specially prepared textures for each face. Typically, this is known as a skybox, or environment map, and may use cube mapping or polar mapping to sample the textures without visible seams at the edges of the cube. X3D provides the *Background* node, which offers cube mapping by defining up to six static images using the attributes *frontUrl*, *backUrl*, *leftUrl*, *rightUrl*, *topUrl*, *bottomUrl*. Thus, the utilization of a single `<Background>` element in X3DOM leads to the desired result and consequently scores high in terms of usability.

XML3D requires several more elements to achieve similar results (see Listing 1). While powerful in terms of flexibility, the required boiler plate code may discourage Web developers from adding a background to their scene. Worse still, this background will fail as soon as the camera leaves the origin of the scene because the skybox must be transformed with the camera. Hence, we immediately hit a further limitation of XML3D in that we are unable to change how the model-view-matrix is applied to an object during rendering.

In addition to the Background node X3D also provides a *TextureBackground* node, which supports the use of animated scenery (e.g. MovieTextures) as backdrop. However, the TextureBackground node is not supported by X3DOM, and thus it is only possible to use static images as a background in X3DOM.

XML3D generally allows texture images to be replaced with the `<video>` element, which allows animated backgrounds. Alternatively, XML3D provides the flexibility that the six textures be the output of a dataflow operator which can generate the image data programmatically. Something that is impossible in X3D.

However there are many other things to consider when trying to specify a background. For example, we may wish to use other kinds of environmental projections (e.g. polarmaps) instead of cubemaps. We may wish to provide our 6 textures as a single image, or use High Dynamic Range textures that require a different sampling algorithm. With this many permutations specialized elements can quickly become over-complicated, while a completely generic approach becomes difficult for Web developers to work with.

In the following sections we will focus on this example to show how our proposed approach to Dec3D can be used to provide both extensibility and an ease of usability similar to X3D's TextureBackground node.

Hence, we enable the community of 3D developers and domain experts to create and share these components with non-experts.

4 Core Elements of Declarative 3D

Following the generic fashion of XML3D, we propose a new approach for Dec3D that allows for the specification of very few *core elements*. Moreover, we leverage recent Web technologies to enable the creation, sharing, and reuse of new elements, built from these core elements. This can introduce additional semantics or convenience as provided by X3D/X3DOM and XML3D.

The effort required for the implementation of Dec3D on the Web is significantly reduced by the small number of core elements. Furthermore, the approach allows users and communities to build new domain-specific elements and share them, for instance in the fashion of libraries, which ensures extensibility.

Overall, we have identified 8 interrelated core elements: For rendering (`<drawable>`, `<material>`, `<light>`), data description (`<data>`, `<value>`), scene graph organization (`<camera>`, `<group>`), and a root element (`<web3d>`) to embed the 3D content in HTML (see Figure 2).

4.1 Rendering Elements

Rendering elements act as data sinks to describe and send data to the renderer. Thus, each rendering element represents a dedicated interface for data that is fed into the respective part of the rendering process. In a web-based environment, we specifically target rasterization by utilizing WebGL. However, the Dec3D layer is designed to be renderer independent and may target other rendering approaches (e.g. ray tracing) as well.

4.1.1 Rendering Primitives

The `<drawable>` element represents geometry in the scene similar to the XML3D concept. A *Drawable* supports common rendering primitives (i.e. triangles, lines, points, triangle strips, line strips, triangle fans) via its attribute *type*. Hence, the vertex position of the geometry is mandatory input data. In terms of WebGL, a Drawable requires the value that is written to *gl_Position* in the vertex shader stage. Moreover, an index to fetch vertices of the geometry for each rendering primitive can be utilized as input data. This will result in an indexed draw call in WebGL. Other input data will be made available to the renderer, which may take them into account as additional vertex attributes and uniforms.

With regard to our scene background example, the input for the Drawable will consist of *position* and *texcoord* for 12 triangles to render the six sides of a cube.

4.1.2 Appearance and Lighting

Each Drawable refers via document id to a `<material>` element that defines the geometry's surface appearance. A *Material* uses one of the built-in material models (e.g. matte, diffuse, Phong) and configures its properties. In addition, a Drawable may override specific properties of its assigned Material in order to customize its appearance.

Moreover, we utilize a `<light>` element that represents a light source in the scene. A *Light* uses one of the built-in light models

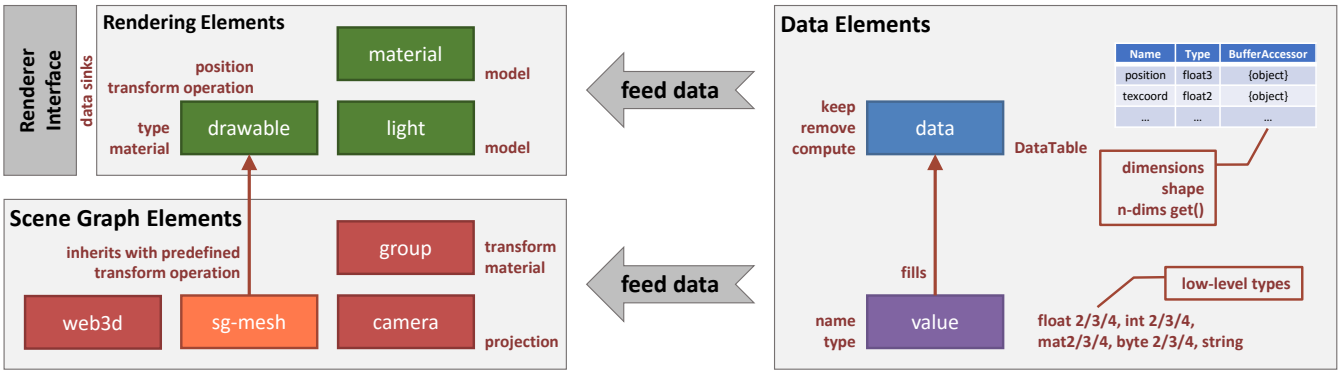

Figure 2: *The relation of core elements for Dec3D on the Web: Rendering elements (drawable, material, light), data elements (data, value), scene graph elements (camera, group), and root element (web3d). The sg-mesh element is a built-in Web Component based on drawable.*

(i.e. distant, point, spot), which are commonly used and well-known from APIs like OpenGL. These Lights may be taken into account by the Material of a Drawable during the rendering process.

For our background cube we utilize a material model that renders with constant intensity over our backdrop.

4.1.3 Custom Material Models

To go beyond ubershaders, we make use of shade.js [Sons et al. 2014] as a concept of portable material descriptions. Consequently, a Material may refer to a user-defined material model that is specified via shade.js rather than using one of the built-ins.

Despite the fact that it would be possible to define the appearance of our backdrop by a built-in material model, defining a custom one has several advantages. First, we can get rid of unnecessary computational overhead. Moreover, we can utilize an actual cubemap sampler using respective texture lookups rather than folding the 2D-texture around the cube. It would also allow us to support polar mapping directly, rather than resampling it into a cubemap as would be needed in X3D/X3DOM.

4.2 Data Elements

Our proposal for data elements heavily builds upon the concept of generic dataflow processing as described by Klein et al. [2012]. The most relevant `<data>` element represents a data container exposing an interface to retrieve a *DataTable* with entries consisting of a name, the type of available data, and a *BufferAccessor* (see Figure 2). The name of a data entry appears only once per DataTable and is arbitrary, but it may have a semantic for the data sink that it feeds into. The type identifier of the available data has predefined values representing low-level types, such as float[234], int[234], mat[234], byte[234] as well as string. The BufferAccessor acts as an interface to the actual data represented as an n-dimensional array of the defined type, providing the dimension and the shape of the available data. Moreover, the BufferAccessor offers an n-dimensional `get()` to fetch data directly (i.e. specify all n dimensions) or retrieve another BufferAccessor to data containing less dimensions than before (i.e. slice of n-dimensional data).

4.2.1 Data Input

To initially feed a data container with entries, a `<value>` element is utilized that represents a single data entry. This element offers two attributes to define the *name* of the entry in the DataTable as well as the *type*. Thereby, the type identifier indicates how to parse the text content of the value element as well as the exposed BufferAccessor.

Using this approach, we can interface commonly used data structures such as vector types, arrays of simple and vector types, images (i.e. 2-dimensional arrays of vector types), and arrays of images, in the same manner. Listing 2 (Lines 3-5) demonstrates a `<value>` element containing a list of float2 that are interpreted by the renderer of the Drawable as texture coordinates.

4.2.2 Data Compositing

The DataTables of several data containers can be merged either by nesting them or by using references via document id in the attribute *src* (see Listing 2). Moreover, external data can be incorporated by referring to external documents (e.g. json, xml) while taking advantage of built-in format handlers that fill a respective DataTable. This data composition approach allows a DataTable to be extended with additional data. It is efficient due to the reuse of data buffers (e.g. on the GPU) while remaining flexible through the ability to work with both individual entries or entire sets of entries.

Furthermore, unnecessary data entries can be stripped-off from the table by using the attributes *keep* and *remove* of the `<data>` element. When both attributes are present the attribute keep has priority. Listing 2 shows the attribute keep being used to proceed only with the data from the external file that we're interested in.

4.2.3 Data Processing

Additionally, a data container may perform a computation on its DataTable to derive new entries or replace existing ones. This is done by declaring an operator invocation on a `<data>` element using the attribute *compute*. Operators are designed to have named inputs rather than positional arguments. Thus, generic operators may need to map specific entries of the DataTable to their named inputs. Likewise, the results returned by an operator computation may need to be redirected to the right data entries by name. For that purpose, we utilize a destructuring assignment and passing parameters as object properties (see Listing 2 Line 24) as defined in ECMAScript 2015[2].

4.2.4 Custom Operators and Data Types

Additional operators may also be registered with the dataflow engine and used at any point. This enables extensibility of the

[2]http://www.ecma-international.org/ecma-262/6.0/index.html

```
<data id="a" keep="position, texcoord, index">
  <data src="cube.json"></data>
  <value name="texcoord" type="float2">
    0 0.33 0.25 0.33 0 0.66 0.25 0.66 ...
  </value>
</data>

<data id="b" compute="rgbeToFloat()">
  <!-- stores rgbe values in rgba channels -->
  <img name="rgbeImg" src="sky-cube.png"/>
</data>

<script type="application/compute-operator">
  function rgbeToFloat(inputs) {
    var img = inputs.rgbeImg.map(pixel => {
      return rgbe2float(pixel);
    }
    return {rgbaImg: img};
  }
</script>

<data id="background-cube">
  <data src="#a"></data>
  <data compute="assign({envmap:rgbaImg})">
    <data src="#b"></data>
  </data>
</data>
```

Listing 2: *An example data graph that filters, renames and manipulates a set of mesh data.*

data processing using custom operators and allows to perform application-specific computations such as data decoding. With regard to our backdrop example, we utilize a custom operator to decode an RGBE into a floating point image compatible with our existing background (see Line 8 of Listing 2).

Similarly, one would like to amend the built-in types. Hence, the type identifier in the DataTable can be any arbitrary name to represent custom types. A custom BufferAccessor would be required to interface with this new data type. Custom types can be introduced either as output of a custom operator or by a custom `<value>` element. In any case, they must resolve to built-in low-level types during the dataflow processing before being fed into a data sink. This is normally done using an operator that converts the custom type into a type that the data sink expects.

4.2.5 Sampling Data Buffers

The BufferAccessor itself solely interfaces with the raw data. When operating on n-dimensional data, such as 2D/3D images, the interpolation method and the boundary handling come into play. These sampling concepts can easily be implemented on top of the BufferAccessor and thereby left to the relevant places (i.e. shading).

4.3 Scene Graph Elements

Rendering elements of the scene (i.e. `<drawable>`, `<light>`) can be organized by being placed into `<group>` elements [Sons et al. 2010]. Each *Group* defines a coordinate system relative to its parent element that can be manipulated via CSS 3D Transforms [W3C 2013]. This way, a tree of `<group>` elements represents the transformation hierarchy of the scene within a `<web3d>` element as root. In addition to rendering elements, this hierarchy may include others, such as `<camera>` elements for viewpoints. This type of scene description implicitly defines several spaces (i.e. coordinate systems) that are well-known in computer graphics. Furthermore,

```
<script type="application/transform-operation">
  // usual scene graph mesh (sg-mesh)
  ClipSpace := projectionMatrix * ViewSpace
  ViewSpace := viewMatrix * WorldSpace
  WorldSpace := modelMatrix * ObjectSpace
</script>
<script type="application/transform-operation">
  // environmental background
  // mat3() represents upper 3x3 matrix
  ClipSpace := projectionMatrix * ViewSpace
  ViewSpace := mat3(viewMatrix) * WorldSpace
  WorldSpace := mat3(modelMatrix) * ObjectSpace
</script>
<script type="application/transform-operation">
  // screen-aligned quad
  ClipSpace := ObjectSpace
</script>
```

Listing 3: *Example transform operations for usual meshes, environmental background, and screen-aligned quads. These expressions are used to generate code for the respective platform (e.g. a vertex shader).*

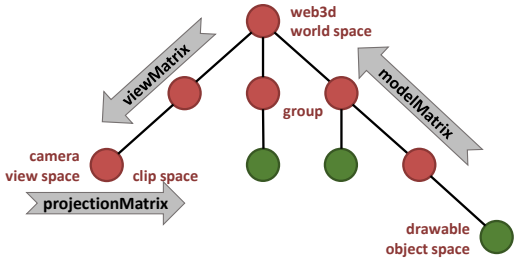

Figure 3: *Spaces and transform operations between them within a scene graph hierarchy.*

a `<camera>` element may include intrinsic camera properties to be passed to the renderer.

4.3.1 Scene Graph Mesh

The geometry of a scene is usually defined in a local coordinate system referred to as *object space*. Furthermore, the `<web3d>` element defines a coordinate system referred to as *world space*, whereas the active `<camera>` element defines the *view space* and *clip space* based on the (perspective) projection. The transformation required to switch from one space into another is illustrated in Figure 3.

Geometry that is fed into a Drawable must be transformed into clip space during the rendering process. Thus, a Drawable specifies a *transform operation* in the form of expressions (see Listing 3) that is used by the renderer to generate implementation-specific code (i.e. vertex shader). The matrices are provided by the renderer, utilizing the scene hierarchy of `<group>` elements (with their respective transforms) as well as the active `<camera>` element. Since the vast majority of usual scene content shares the same transform operation we introduce the `<sg-mesh>` element as built-in Web Component, which maps to a Drawable with a transform operation as defined in Listing 3 (Lines 3-5).

4.3.2 Custom Meshes

In some cases we may want to define a different transform operation for a piece of geometry. For example our backdrop cube should only be affected by camera rotations and not translations.

Normally, such geometry would require special handling, whereas the proposed <drawable> element would simply define a different transform operation (see Listing 3 Lines 10-12). Likewise, the inverse direction of space transforms can be easily derived, which is essential for other rendering approaches such as ray tracing.

4.3.3 Configuring the Scene Graph

In addition to the <sg-mesh> element, other scene graph elements may also act as sinks for data containers. The attribute *transform* of a <group> element may refer to a <data> element with an entry of type mat4 named "transform" in order to manipulate its local coordinate system. For interactive changes this can be much faster than updating the CSS 3D Transform of the respective <group> element. It also enables dataflow processing to affect transforms.

Moreover, the attribute *projection* of a <camera> element may refer to a <data> element with an entry of type mat4 named "projection", allowing arbitrary projections. Built-in dataflow operators provide support for commonly used configurable projections (e.g. perspective, ortho).

5 Building upon Core Elements of Dec3D

On top of these core elements, we show in the following how to develop user-defined elements by utilizing Web Components. Web Components cover a set of APIs to introduce *Custom Elements*, reuse DOM-fragments by utilizing *HTML Templates*, and encapsulate subtrees of the document in the scope of a *Shadow DOM*. To illustrate this process, we will develop a <x-background> element as described in Section 3 as well as a convenience element for rendering text in a 3D scene (see Figure 4).

5.1 Custom Elements

It has always been possible to use arbitrary elements in HTML. Elements unknown to the parser are not rendered and are represented as generic DOM elements without specific functionality. The Custom Elements specification provides an API to register new types of elements using JavaScript. To avoid naming collisions with existing and future HTML elements, custom elements require a dash in their name. This is the reason for naming our user-defined elements <x-background> and <x-label> respectively.

Instances of registered elements receive life cycle callbacks on creation, DOM attachment and detachment, and on attribute changes. This allows for a per-element API to build fully-featured DOM elements.

5.2 HTML Templates

HTML5 introduced the <template> element to create reuseable DOM fragments, which are declared as inert DOM subtrees. Subsequently, they can be instantiated and inserted into the DOM via JavaScript and thereby significantly reduce the effort to programmatically create and append DOM elements.

Moreover, HTML Templates ease the declaration of complex or compound elements that bring higher semantics into 3D scenes. As an example, we introduce an <x-label> element that creates a piece of text based on quadrilateral strips and bitmap fonts. The element exposes, via an attribute, the property *string* containing the text to be displayed. Listing 4 shows the element's HTML template. A generic method of a one-way binding of the attributes of a custom element to a named placeholder (i.e. {{string}}) is used in order to feed the content of an element's attribute into the instance of a template.

```
<template id="x-label" string="default text">
  <sg-mesh material="text.xml#shader-alphatest">
    <data compute="xflow.quads2triangles()">
      <data compute="xflow.text()">
        <data src="text.xml#font-config"></data>
        <value type="string" name="text">
          {{string}}
        </value>
      </data>
    </data>
  </sg-mesh>
</template>
```

Listing 4: *HTML Template for a Custom Element x-label to place text in 3D scenes. The value of the attribute "string" is being fed into the named placeholder where it is used to generate 3D text geometry.*

5.3 Shadow DOM

Taking advantage of reusable components may introduce a similar issue that a simple copy and pasting of existing HTML layouts raises: Applying CSS to define the style of HTML elements may lead to unexpected results due to the multiple ways CSS selectors may unintentionally match parts of the copied HTML.

The Shadow DOM is a W3C working draft to enable encapsulation and isolation of DOM content. Content within the Shadow DOM is not visible in the website's actual DOM, but still considered for rendering. For CSS this means no rule can interfere with content that resides in the Shadow DOM as long as it is not explicitly targeting this content with special selectors. The same applies the other way around, which means that CSS bundled with the reusable component does not interfere with content outside the Shadow DOM (i.e. the hosting element) except where explicitly targeted.

In conjunction with Custom Elements this means reusable but encapsulated HTML elements can now be created. With regard to our example of the <x-label> element, the boxed part in Figure 4 containing the <sg-mesh> and <data> is encapsulated in the Shadow DOM.

5.4 Generic Templates and Attribute Binding

The combined capabilities of Web Components and dataflows in Dec3D reduces the implementation of <x-label> to a simple instantiation of the template as shown in Listing 4 in the Shadow DOM and a primitive one-way attribute binding. Hence, a reuseable Dec3D component requires no runtime logic in JavaScript since its behavior can be completely encoded in a declarative dataflow graph. As a consequence, even user-defined elements can be fully declarative. Ultimately, the complexity of rendering text in a 3D scene (see Figure 4) is defined in a declarative fashion and hidden behind a single element. As a result a user can add 3D text to a scene with only a single line of HTML.

6 Representation of Pre-Existing Elements

Creating full-fledged custom elements is not the only motivation for the introduction of Custom Elements. According to its specification, it is also designed to rationalize the Web platform, i.e. "these new features could be used to explain the functionality of existing Web platform features, such as HTML elements" and the spec "is designed to shorten the distance to a much more ambitious goal of rationalizing all HTML, SVG, and MathML elements into one coherent system" [W3C 2014b]. Hence, a Dec3D scene description

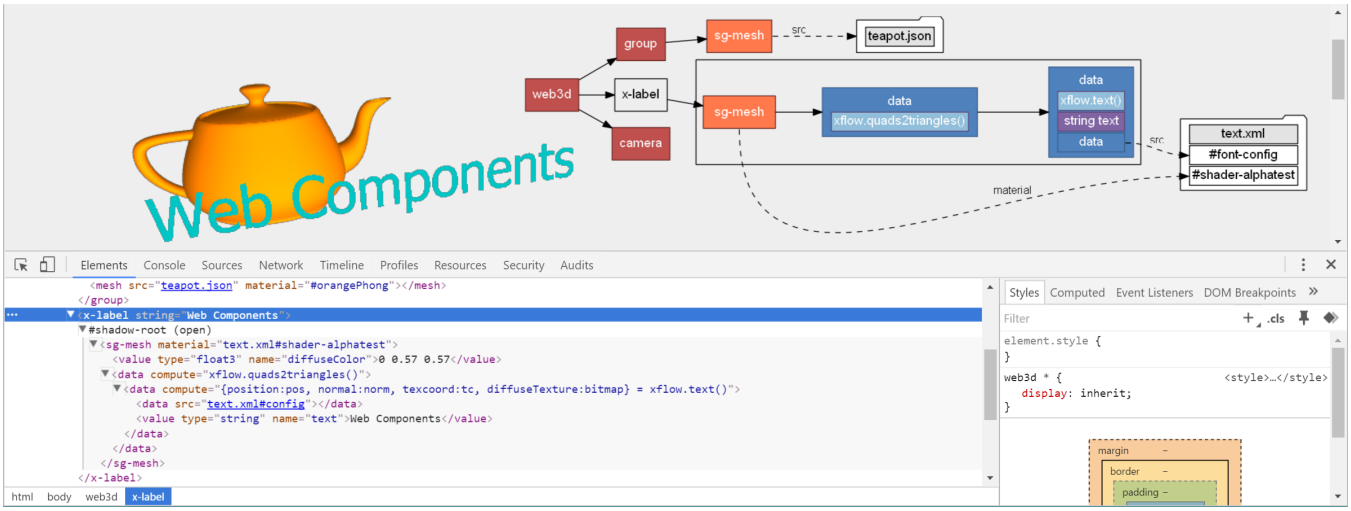

Figure 4: *3D scene showing Custom Element x-label, that hides the complexity of text rendering using Web Components. Shown in Google Chrome with debugging and expanded DOM tree including the Shadow DOM as well as a visualization of the dataflow.*

should set itself the same ambitions goals.

With this in mind it is worth revisiting the abstract models of X3DOM and XML3D. It would be possible to streamline both, discarding elements which can be implemented using Web Components. XML3D was designed to provide a minimal set of elements necessary to describe 3D scenes [Sons et al. 2010] and provides very few candidates for streamlining. In contrast, X3DOM already implements large parts of the related X3D specification [ISO 2008], which consists of more than 250 nodes. Many of these nodes can now be implemented using Web Components.

This approach dramatically reduces the need for special handling of Dec3D content by the browser or any polyfill implementation, opting instead to encapsulate it inside user-defined components that can be further combined and shared.

These components can be created and maintained by the Web community, greatly easing the implementation burden currently put on developers of Dec3D approaches. They can also be easy to work with, as most of the complex implementation details are hidden behind the encapsulated Shadow DOM. Web developers may then use the component as they would any other HTML element, through a well defined interface of attributes and JavaScript functions.

6.1 X3DOM Nodes

The current X3DOM specification contains about 187 individual leaf nodes[3] (i.e. HTML elements). Some elements, for instance those related to audio, are outside the scope of this paper as we focus solely on the core functionality needed for 3D graphics. Listing 5 shows a portion of the example "Hello X3DOM" that we utilize to demonstrate Web Components and our core elements of Dec3D. This scene contains some of the most common scene content such as materials, transforms and various generated shapes.

We prefix every element with "x3d-" to meet the naming scheme of Custom Elements and subsequently embed the scene into a `<web3d>` element. This process is done programmatically with a script replacing the polyfill of X3DOM. The script also gathers the list of required components and loads them dynamically from a central repository. That allows for providing a compatibility layer for

[3] http://doc.x3dom.org/author/nodes.html

```
<web3d>
  <x3d-scene>
    <!-- x3d-group with x3d-box -->
    <x3d-transform translation="-3 0 0">
      <x3d-shape>
        <x3d-appearance>
          <x3d-material diffuseColor="0 1 0">
          </x3d-material>
        </x3d-appearance>
        <x3d-cone></x3d-cone>
      </x3d-shape>
    </x3d-transform>
    <!-- x3d-transform with x3d-box -->
  </x3d-scene>
</web3d>
```

Listing 5: *A portion of the example "Hello X3DOM" implemented with Web Components on top of the core elements.*

displaying existing X3DOM scenes as well as seamless sharing of other custom components.

Scene The `<x3d-scene>` element is a wrapper around the 3D scene content. It is not part of the X3D spec, but acts as a simple Group while also handling the configuration of some global X3DOM-specific parameters such as picking.

Transform The `<x3d-transform>` element also behaves like a Group and always defines a 3D transformation to be applied to its children. This can be mapped using CSS 3D Transforms or, as we did, with a `<data>` element that calculates an entry "transform" of type mat4 as described in Section 4.3.3.

Shape A `<x3d-shape>` element combines a surface material with geometry to define renderable objects. Hence, it maps to one `<material>` element, referenced by a `<group>` element that wraps all attached geometry, such as elements resolved to Drawables.

Cone The `<x3d-cone>` element represents the geometry needed to render a cone. We generate the geometry using a dataflow and feed the resulting mesh data into a `<sg-mesh>` element. The va-

```
<template id="xml3d-transform"
          translation="0 0 0"
          rotation="0 0 1 0" center="0 0 0"
          scale="1 1 1" scaleOrientation="1 1 1">
  <data compute="createTransform()">
    <value type="float3" name="translation">
      {{translation}}
    </value>
    <value type="float4" name="rotation">
      {{rotation}}
    </value>
    <!-- center, scale, scaleOrientation -->
  </data>
</template>
```

Listing 6: *Custom Element xml3d-transform using HTML Templates and dataflow operators for the computation.*

rious attributes (e.g. `height` and `bottomRadius`) are used as input to the dataflow. The other geometry nodes of X3DOM (e.g. box and sphere) are implemented similarly.

Appearance The `<x3d-appearance>` element aggregates several properties affecting the surface appearance of the associated geometry. The combined set of these properties is fed into the `<material>` element of a shape and thus applied to the attached geometry.

Material The `<x3d-material>` element covers properties relevant to surface shading, such as diffuse color and transparency. Values can be set through its attributes and are subsequently passed into the dataflow of the surrounding `<x3d-appearance>` element, which ultimately feeds the `<material>` element of a shape.

6.2 XML3D Elements

Although the XML3D specification has traditionally contained a relatively small number of elements even some of them can now be replaced by Web Components. Similar to X3D/X3DOM we prefix the elements with "xml3d-" to meet the naming scheme of Custom Elements as well as make clear that they relate to XML3D.

Value Elements XML3D currently contains many elements which act as input to Xflow's data processing. All those can be represented using a single `<value>` element while setting the respective low-level type through the *type* attribute as described in Section 4.2.1. This approach would also be more flexible as it can easily be extended with new data types. For example, the current XML3D specification lacks an `<xml3d-int3>` element, which can be created similar to the existing `<xml3d-int4>`.

Transform XML3D provides multiple ways to define a 3D transformation on scene elements. The transformation of a scene object is usually specified using CSS 3D Transforms [W3C 2013], or alternatively by referencing a `<data>` element that holds the transformation. For convenience XML3D also offers the `<xml3d-transform>` element, which composes a transformation matrix from attributes such as translation and rotation. The result of this element is just a transformation matrix, so it can also be defined as the output of a dataflow. Hence, the built-in functionality of the `<xml3d-transform>` element can be substituted with a component as shown in Listing 6.

7 Evaluation

We have presented a set of core elements for embedding 3D graphics into the DOM, achieving a flexible and extensible approach. Furthermore, we have shown that we can use Web Components to build convenience and domain-specific elements on top of this existing core. In doing so, we address the crucial requirements of usability as well as extensibility for Dec3D.

We implemented 16 of the most common X3DOM nodes using our set of core elements in the context of several example scenes. To achieve full compliance with X3DOM the subset of the remaining 175 nodes that is relevant to 3D graphics must be implemented in the same fashion. Ultimately, concepts such as routes must be added as well to achieve full backward compatibility.

Since our system takes advantage of Web Components it makes heavy use of DOM elements. Thus, we performed first measurements of the memory usage and found that it increases almost linearly according to the number of scene objects (i.e. number of DOM elements). The results are shown in Table 1.

Table 1: *The memory usage (in MB) increases almost linearly with regard to the number of scene objects (i.e. 5k, 10k, 15k). Compared to XML3D the base memory usage of our system is higher, whereas the average increase per 5k scene objects is slightly lower.*

	Base	**5k**	**10k**	**15k**	**avg per 5k**
XML3D	7.8	49.8	91.3	133.0	
		+42.0	+83.5	+125.2	**41.78**
Our System	12.9	49.4	86.3	123.0	
		+36.5	+73.4	+110.1	**36.67**
Our System	13.0	32.9	53.2	73.9	
w/o rendering		+19.9	+40.2	+60.9	**20.17**

8 Conclusion and Future work

Given the presented set of core elements combined with Web Components, we have demonstrated how to rationalize elements of existing Dec3D approaches, such as X3DOM and XML3D. This allows the Declarative 3D specification for the Web to be streamlined down to its essential components.

As a result, we can amend the integration model defined by Jankowski et al. [2013] with a fifth level of integration that resides at the top of the stack and consists of re-usable and sharable elements, in the form of component libraries (see Figure 5). Thereby, we keep the *basic building blocks of Dec3D* separated from convenience and domain-specific functionality, including compatibility layers for the two existing approaches. Instead, we allow Web developers to take advantage of selected components that exactly fit their needs. This also enables communities to become much more engaged with Dec3D by contributing to this fifth level. Hereby, we envision community-maintained repositories of reuseable components[4] for a wide range of Dec3D usecases. Ultimately, we feel this will help achieve the primary goal of the Dec3D community to increase the adoption of Dec3D on the Web.

Following our re-evaluation of the DOM integration, we deem it worthy to revisit further features of Dec3D in light of the advancing Web technology stack. For example, we believe that the instantiation of assets (i.e. prefabs) and interfacing configurable assets [Klein et al. 2014] would be good candidates for future work.

[4] similar to https://customelements.io/ and https://elements.polymer-project.org/

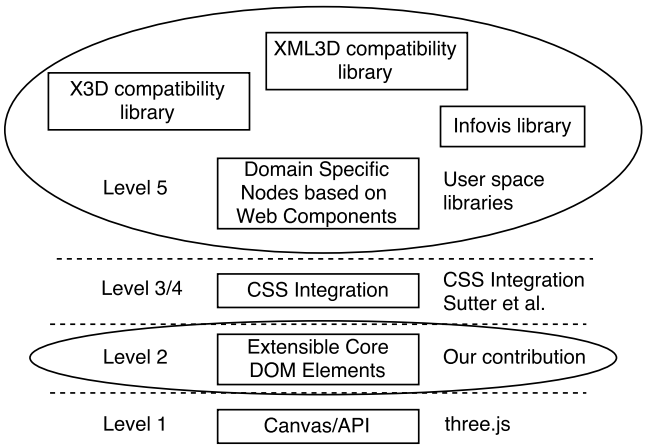

Figure 5: *Our proposed new integration model for 3D content on the Web incorporating the new Web Components technology as well as the CSS integration approach.*

Acknowledgements

The research leading to these results has been generously supported by the Intel Visual Computing Institute and has received funding from the European Union's Seventh Framework Programme under grant agreement no. 632893 (FI-Core), and under grant agreement no. 641191 (CIMPLEX) in the European Union's H2020 Framework Programme.

References

BEHR, J., ESCHLER, P., JUNG, Y., AND ZÖLLNER, M. 2009. X3DOM: A DOM-based HTML5/X3D Integration Model. In *Proceedings of the 14th International Conference on 3D Web Technology*, ACM, New York, NY, USA, Web3D '09, 127–135.

BEHR, J., JUNG, Y., KEIL, J., DREVENSEK, T., ZÖLLNER, M., ESCHLER, P., AND FELLNER, D. W. 2010. A Scalable Architecture for the HTML5/X3D Integration Model X3DOM. In *Proceedings of the 15th International Conference on Web 3D Technology*, ACM, New York, NY, USA, Web3D '10, 185–194.

ISO. 2008. ISO/IEC 19775-1:2008 Extensible 3D (X3D) – Part 1: Architecture and base components. Tech. rep., International Organization for Standardization.

JANKOWSKI, J., RESSLER, S., SONS, K., JUNG, Y., BEHR, J., AND SLUSALLEK, P. 2013. Declarative Integration of Interactive 3D Graphics into the World-Wide Web: Principles, Current Approaches, and Research Agenda. In *Proceedings of the 18th International Conference on 3D Web Technology*, ACM, New York, NY, USA, Web3D '13, 39–45.

KHRONOS, 2011. WebGL Specification Version 1.0, Khronos Specification. https://www.khronos.org/registry/webgl/specs/1.0.0/, Feb.

KLEIN, F., SONS, K., RUBINSTEIN, D., BYELOZYOROV, S., JOHN, S., AND SLUSALLEK, P. 2012. Xflow - Declarative Data Processing for the Web. In *Proceedings of the 17th International Conference on 3D Web Technology*, ACM, New York, NY, USA, Web3D '12, 37–46.

KLEIN, F., SONS, K., RUBINSTEIN, D., AND SLUSALLEK, P. 2013. XML3D and Xflow: Combining Declarative 3D for the Web with Generic Data Flows. *IEEE Computer Graphics and Applications 33*, 5 (Sept.), 38–47.

KLEIN, F., SPIELDENNER, T., SONS, K., AND SLUSALLEK, P. 2014. Configurable instances of 3d models for declarative 3d in the web. In *Proceedings of the Nineteenth International ACM Conference on 3D Web Technologies*, ACM, New York, NY, USA, Web3D '14, 71–79.

SCHWENK, K., JUNG, Y., BEHR, J., AND FELLNER, D. W. 2010. A Modern Declarative Surface Shader for X3D. In *Proceedings of the 15th International Conference on Web 3D Technology*, ACM, New York, NY, USA, Web3D '10, 7–16.

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

SONS, K., SCHLINKMANN, C., KLEIN, F., RUBINSTEIN, D., AND SLUSALLEK, P. 2013. xml3d.js: Architecture of a Polyfill Implementation of XML3D. In *Proceedings of the 6th Workshop on Software Engineering and Architectures for Realtime Interactive Systems*, IEEE, New York, NY, USA, SEARIS '13, 17–24.

SONS, K., KLEIN, F., SUTTER, J., AND SLUSALLEK, P. 2014. shade.js: Adaptive Material Descriptions. *Computer Graphics Forum 33*, 7 (Oct.), 51–60.

SUTTER, J., SONS, K., AND SLUSALLEK, P. 2015. A CSS Integration Model for Declarative 3D. In *Proceedings of the 20th International Conference on 3D Web Technology*, ACM, New York, NY, USA, Web3D '15, 209–217.

W3C, 2011. Scalable Vector Graphics (SVG) 1.1 (Second Edition), W3C Recommendation (work in progress). http://www.w3.org/TR/2011/REC-SVG11-20110816/, Aug.

W3C, 2013. CSS Transforms Module Level 1, W3C Working Draft (work in progress). http://www.w3.org/TR/2013/WD-css-transforms-1-20131126/, Nov.

W3C, 2014. CSS Custom Properties for Cascading Variables Module Level 1, W3C Last Call Working Draft (work in progress). http://www.w3.org/TR/2014/WD-css-variables-1-20140506/, May.

W3C, 2014. Custom Elements, W3C Working Draft (work in progress). http://www.w3.org/TR/2014/WD-custom-elements-20141216/, Dec.

W3C, 2014. HTML Imports, W3C Working Draft (work in progress). http://www.w3.org/TR/2014/WD-html-imports-20140311/, Mar.

W3C, 2014. HTML5 - A vocabulary and associated APIs for HTML and XHTML, W3C Recommendation (work in progress). http://www.w3.org/TR/2014/REC-html5-20141028/, Oct.

W3C, 2014. Shadow DOM, W3C Working Draft (work in progress). http://www.w3.org/TR/2014/WD-shadow-dom-20140617/, June.