A CSS Integration Model for Declarative 3D

Jan Sutter*
DFKI
Saarland University

Kristian Sons†
DFKI
Saarland University

Philipp Slusallek‡
DFKI
Saarland University
Intel VCI

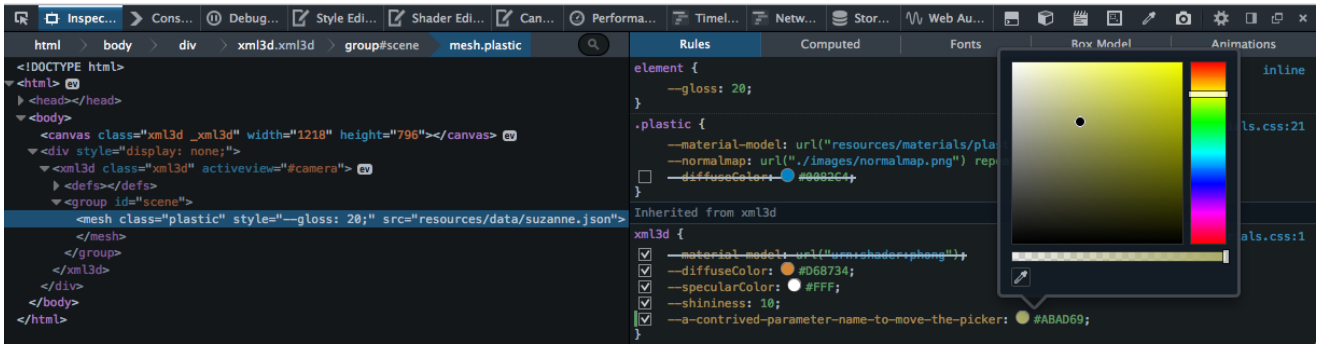

Figure 1: *Debugging an XML3D scene that uses our proposed CSS integration model. It is now possible to change, add or remove material parameters right within the well-known debugging facilities of today's browsers (here: Mozilla Firefox).*

Abstract

Declarative 3D (Dec3D) implementations, most notably XML3D and X3DOM, have enabled a seamless integration of 3D and 2D content on the same web page. Yet one of the major web technologies, Cascading Style Sheets (CSS), has not been integrated. The usage of CSS for 3D content has always been envisaged but never fully approached, because only polyfills for declarative 3D implementations exist and only recent developments have made custom CSS properties available.

In this paper we will present a deep integration and adaption of CSS for Dec3D content and, hence, provide the final component necessary to fully integrate 3D content into the web technology stack. Our integration model allows for appearance definitions, such as visibility and materials, at a novel level of expressiveness. CSS-Selectors, inheritance, as well as media types provide unique means to change a scene's final appearance in a flexible and powerful way. Using CSS, it is possible to define the appearance of a 3D object dependent on the DOM hierarchy position or the screen resolution and orientation without a single line of JavaScript. The integration of CSS further enables the use of browser debugging facilities that have not been usable before. Because the requirements of 3D content are different compared to those of 2D content, we will point out existing limitations and necessary future additions to improve the interoperability of CSS with 3D content.

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

Keywords: DOM, HTML5, CSS, XML3D, Dec3D, materials

1 Introduction

Web pages today are built on top of three core technologies: HTML, CSS and JavaScript. While HTML is used to define the structure and content, and to some degree its semantics, CSS is used for layout and styling purposes and JavaScript for interaction and application logic. The Declarative 3D For The Web Architecture Community Group[1] aims at developing and establishing a standard for describing interactive 3D content that seamlessly integrates with these technologies. Approaches such as X3DOM [Behr et al. 2009] and XML3D [Sons et al. 2010] work towards a standard set of custom 3D related HTML elements to mix 3D scene definitions and standard HTML markup and, as a result, to make 3D content part of the DOM. As part of the DOM, all 3D scene content can be managed using existing and well-known JavaScript libraries such as jQuery[2]. With the integration into standard web technologies the Dec3D community further tries to make 3D content accessible to the broad audience of web developers with little to no knowledge of 3D graphics in general and rasterization in particular (see [Behr et al. 2011; Sons et al. 2010]).

As of today, however, no single standard for declarative 3D on the web exists and none of the existing approaches is natively supported by any browser. XML3D and X3DOM, therefore, rely on polyfill implementations based on WebGL [Khronos 2011]. Jankowski et al. [2013] describe various levels of integration of 3D graphics on the web (see Figure 2). The first level is defined to be on par with

*e-mail:jan.sutter@dfki.de
†e-mail:kristian.sons@dfki.de
‡e-mail:philipp.slusallek@dfki.de

[1]https://www.w3.org/community/declarative3d/
[2]http://jquery.com

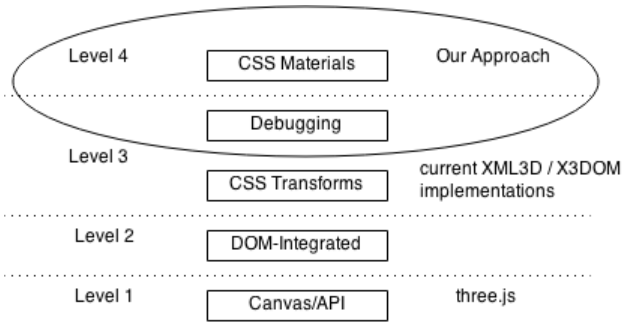

Figure 2: *Integration levels for 3D content on the Web according to Jankowski et al. [2013]. Our contribution for the first time provides an integration at the final level.*

```
<p>outer</p>
<div id="content">
    <p>first level</p>
    <div class="description">
        <p>description</p>
    </div>
    <div class="details">
        <p>details</p>
        <span class="disclosure"
            data-elem="details">
            ...</span>
    </div>
<div>
```

Listing 1: *A basic HTML structure to realize progressive disclosure in dependence of the available screen width.*

WebGL, an imperative API to define 3D content. This is the level of integration that libraries such as three.js[3] provide. The second level is defined as the incorporation of 3D specific elements into the DOM, e.g. mesh and material elements. The third and fourth level of integration require CSS transformations, debugging functionality and CSS based material descriptions. XML3D and X3DOM have achieved a full DOM integration, including events and specific 3D related HTML elements. However, both support inline style definitions for CSS based 3D transformations only [Sons et al. 2010; Behr et al. 2011]. XML3D and X3DOM can thus be considered to reside in-between the second and third integration level. However, neither of the two approaches has a full CSS integration and, thus, intertwines styling and structure in the DOM. This lack of a deeper CSS integration renders many existing debugging facilities for 3D content useless and prevents the use of existing CSS libraries and frameworks.

CSS is a well-known and integral technology for every web developer and a powerful and expressive tool to define the appearance of an element and its content on a web page. Two important features of CSS are selectors [W3C 2011b] and media queries [W3C 2012]. Selectors provide a way for developers to match a set of elements based on class names, ids, DOM hierarchy position, interaction state and attributes. Every object matched by such a selector is subject to the specified CSS rule. Media queries on the other hand allow for the conditional styling [W3C 2013a] of elements depending on client specifics such as the screen resolution.

[3] http://threejs.org

```
.disclosure:not([data-elem=""]) {
    display: none;
}
@media only screen
and (max-width: 320px) {
    #content {
        color: gray;
    }
    div.details > p {
        display: none;
    }
    .disclosure:not([data-elem=""]) {
        display: block;
        color: blue;
    }
    .disclosure:hover:not([data-elem=""]) {
        color: red;
    }
}
```

Listing 2: *CSS rules for the progressive disclosure example.*

Example Consider the example in Listing 1 with two nested divs, each containing a paragraph. To show the expressiveness of CSS we will now change the presentation of this content depending on the width of the available screen space, a technique called *progressive disclosure*. On less than 320px wide screens we want to hide all paragraphs underneath a div with the `details` class. Instead we want to show the child `span` with the text content colored in blue, but only if it has a `data-elem` attribute set and the class `disclosure` attached. When the users hovers over the `span`, it should change its color to red. Finally all text inside the `content` div will be colored gray on small screens. To achieve these effects a web developer would use a CSS based solution similar to the one shown in Listing 2.

If we now imagine an almost equivalent 3D scene, exchanging each div with an XML3D `<group>` or X3DOM `<transform>` element and each paragraph with an XML3D `<mesh>` or X3DOM `<shape>`. Because progressive disclosure is not a common technique in computer graphics, considering that we typically zoom fit the screen, we will replace the mesh under the `details` group with a different level-of-detail. The mesh should still be highlighted on hover and the color of all remaining visible meshes should change to gray on small screens. Since declarative 3D lacks a proper CSS support we can only achieve this using JavaScript. We first have to detect the screen width. In case the screen is less than 320px wide we have to collect all meshes underneath the `details` group and all distinct meshes underneath the `content` group. Then the new level-of-detail has to be made visible. In order to change the color, `mouseover` and `mouseout` listeners have to be registered that change the corresponding parameters appropriately. Afterwards, all other meshes underneath the `details` group have to be hid. Finally, the color of all the materials of the remaining meshes in the `content` group has to be changed to gray. However, during this change we have to ensure that we do not accidentally affect other meshes that share the same material description but are not part of this group. To avoid such unintentional effects, XML3D provides material override semantics [Sons et al. 2010], but this would require new DOM elements to be inserted underneath each mesh.

The simple example above already shows how much application-logic is required to achieve the same result as five elementary CSS rules. In this paper we will describe an integration of CSS for

declarative 3D content that will enable the exact same powerful use of CSS for 3D content such that all the application-logic from the given example is not needed anymore and can be moved into a declarative CSS based description. We will show how CSS can be used for material, lights, cameras, and visibility definitions. The integration of CSS will not only increase the level of integration and "debuggability", but provides an expressiveness for material definitions that has not been available yet. However, CSS is not meant to obsolete other existing concepts needed for external references to existing 3D content [Klein et al. 2014; Sutter et al. 2014]. The integration is meant to complement these approaches and to configure and style assets in these formats. Finally, we will describe an implementation of the proposed CSS integration for the *xml3d.js* polyfill [Sons et al. 2013] based on the *CSS Custom Properties for Cascading Variables Module Level 1* working draft [W3C 2014a], making XML3D the first implementation to reach the fourth level of integration.

2 Related Work

Alongside HTML, SVG [W3C 2011a] – a declarative language for describing two-dimensional vector graphics – has adopted CSS properties to define how graphics elements are to be rendered. In contrast to HTML, where style properties replaced attributes that define styling, SVG allows for both: Each property can be defined using either CSS or attributes. SVG has a predefined set of CSS properties to configure its fixed function shading model.

Using CSS for 3D content has been envisaged from the very beginning of the declarative 3D initiative [Behr et al. 2009; Sons et al. 2010]. Both existing implementations, XML3D and X3DOM, use CSS to define the style of the canvas element that is placed in the DOM, e.g. width, height or border. CSS Transforms [W3C 2013b] can additionally be used to define an object's 3D transformation [Sons et al. 2010]. However, neither XML3D nor X3DOM provides a full CSS integration for all elements of a scene. Sons et al. [2010] discuss the importance of CSS for a thorough integration of 3D content into the web but only in regard to 3D transformations and an object's material. Jankowski et al. [2013] propose to use existing properties, such as color and opacity, and to define new properties for shading similar to the approach taken for SVG.

In contrast to HTML, where unknown tags are ignored by the browser and left in the DOM untouched, unknown CSS properties are stripped from an element's style and are unaccessible from JavaScript. The current working draft for CSS custom properties [W3C 2014a] defines a standard way for the specification of user-defined CSS properties that are accessible from JavaScript. Moreover, these new properties can be used as variables, using the `var` statement, and can be evaluated in `calc` expressions. The working draft requires that all custom properties start with a double hyphen (`--`) to not collide with possible future CSS properties. The type of the value of a custom property is a literal string that is passed *as is* to JavaScript. No interpretation, except for the substitution inside `var` statements, is done by the browser and it is the application's sole responsibility to parse and interpret the provided value. This value is inherited following the standard rules of CSS and the resolved style of a node can be queried in JavaScript using the `getComputedStyle` functionality.

The use of CSS to specify shader uniforms has already been discussed in the *Filter Effects Module Level 1* working draft from 2013 [W3C 2013c], but has been removed since then. It describes *Custom Filter Functions*, a way to define custom GLSL-based shaders to specify the vertex and fragment shading of an HTML element's rendering pipeline. The definition of uniform variables is discussed and how the necessary data types can be expressed in CSS. Most of the definitions of our data types are either taken literally from this draft or are designed to closely match their way of specifying data types. This should increase the probability that a possible future reappearance of this feature is as compatible as possible with our proposed handling of CSS based material descriptions.

The expressiveness of CSS and its cascading nature, notwithstanding its power, often results in the unintentional propagation of styling into unrelated parts of the DOM, if selectors are not specific enough. Recent advances such as the Shadow DOM [W3C 2014d] provide a way for web developers to hide parts of the DOM. This hidden DOM is then unaffected by any CSS that is not specifically targeting these hidden parts. In conjunction with Custom Elements [W3C 2014b] the Shadow DOM is the new approach to encapsulate, reuse, and instantiate existing sub-trees of a DOM. The XML3D asset format [Klein et al. 2014] is a novel way to achieve the same for 3D asset instances. It provides a way to hide an already configured asset to prevent unintentional changes, while still providing the ability to fully configure parts of an instance from the outside by specifically targeting named elements. As the 3D counterpart to the Shadow DOM we will show how we can exploit this to avoid the accidental leaking of CSS styles into an already configured 3D model, but still enabling a full configuration of the asset using CSS rules that explicitly target named parts of the model.

3 CSS-based Styling of 3D Scenes

One of the basic principles of CSS-based styling is the inheritance of property values from parent elements to child elements inside the DOM tree. The browser provides an API to access the final style properties of an element using the `window.getComputedStyle` method.

X3D and X3DOM use a directed acyclic graph (DAG) as a data structure, which allows for instancing whole sub-scenes. Applying CSS on DAGs is possible, but requires additional bookkeeping in order to track the computed styles for all existing paths to a node. This is similar to SVG, where the standard states that "... the conceptual deep cloning of the referenced element into a non-exposed DOM tree also copies any property values resulting from the CSS cascade on the referenced element and its contents." [W3C 2011a]. Because the conceptually cloned DOM tree is not exposed, it is not possible to derive the style of the referenced element using mechanisms such as `window.getComputedStyle`. Without this functionality to access the evaluated style, it is hard – if not impossible – to integrate CSS into X3DOM without re-implementing the whole CSS stack.

In contrast, XML3D organizes all graphical objects in a tree structure. As a result, we can use the available functionality to evaluate the style of these elements. Hence, we have chosen XML3D to discuss the integration of CSS into Dec3D.

Jankowski et al. [2013] define the final level of integration solely on the availability of CSS based *material* descriptions. However, other scene components can benefit from the expressiveness of CSS as well. Because CSS is a mechanism to separate style from content and structure, it is important to identify the parts of a 3D scene description that are style related and to separate these from the parts that provide the structure and content. We identify object visibility, lights, materials and camera intrinsic as "style-able" properties of a 3D scene. For these aspects we will discuss how existing CSS properties can be exploited for 3D content, how they have to be used to increase the interoperability with existing libraries and frameworks, and where more than a fixed set of CSS properties is necessary in order to fully support 3D content styling.

```
xml3d > * > * > * > * mesh {
    display: none;
}
```

Listing 3: *Hiding all meshes deeper than the fifths hierarchy level using a CSS child selector and the display property.*

3.1 Visibility

Hiding or showing an object is a fundamental operation for 2D as well as 3D content. Visibility in HTML and CSS is a tri-state attribute definable through two different CSS properties: *display* and *visibility*. The visibility property can hide or show an element on the web page while it still affects the overall layout and spacing. To avoid any remaining effect on the layout the display property can be set to *none*, in which case the element is neither visible nor does it have any visible effect. The latter is generally what the user intends and the default property that is set by libraries, for instance jQuery, to hide or show an element.

In the context of 3D computer graphics visibility is usually defined as a binary state, where hidden objects are simply not rendered. However, the concept of visibility can be extended to incorporate a third state. Mouse interaction in 3D graphics is done using picking, which requires every pick-able part of an object to be an individual mesh. Scanned objects, which are important for domains such as cultural heritage, are typically comprised of a single large mesh. To allow for more fine granular interactions multiple transparent proxy-meshes are overlaid. For this reason we define both properties, display and visibility, for 3D content. The `visibility` property is used to hide an object, i.e. not visible in the final image, that still responds to mouse events. The `display` property is used to hide an object from the scene ignoring any interaction. This use of the display property is consistent and compatible with the behavior of toolkits such as jQuery.

Usage Example Visibility definitions using CSS is not only mandatory for our introductory example, but especially helpful for the visualization of large construction models. In these models smaller, often uninteresting details, are at very deep levels of the hierarchy. With CSS it is possible to use single selectors to hide all objects starting at a certain hierarchy level to reduce visual clutter and the number of rendered objects. To hide all meshes starting at the fifth level in the hierarchy, the CSS rule shown in Listing 3 can be used.

3.2 Materials

Material definitions in declarative 3D implementations come in two forms. First, using predefined "uber-shaders" such as the *CommonSurfaceShader* proposal [Schwenk et al. 2010] for X3D. These material models have a well-known set of input parameters that can be changed to configure the material. This approach allows for the definition of a fixed set of CSS properties with predefined semantics to cover all configuration parameters of the material model. Custom material definitions, on the other hand, render such an integration method impossible because the set of potential parameters and their types are user-defined. Especially material models based on *shade.js* [Sons et al. 2014], which are able to adapt itself to the defined and available input parameters require more than a finite set of predefined CSS properties and semantics. Integrating CSS into a declarative 3D implementation that uses custom material definitions requires custom CSS properties; without, a complete integration would be impossible. CSS, however, requires that unknown

```
1  #example-material {
2      material-model: url("./shade.js");
3      --numIterations: 10;
4      --reflectivity: 0.2;
5      --diffuseColor: rgb(255, 128, 0);
6      --specularColor: crimson;
7      --upVector: vec3(1.0, 2.0, 3.0);
8      --coeff: array(1, 2, 3.9, 4);
9      --texModulation: mat2(2, 0, 0, 2);
10     --heightmap: url("./height_map.png")
11         clamp repeat linear nearest;
12     --normalmap-src: url("#normal_map");
13     --normalmap-clamp-s: clamp;
14     --normalmap-clamp-t: repeat;
15     --normalmap-filter-min: linear;
16     --normalmap-filter-mag: nearest;
17 }
```

Listing 4: *An example material definition using all the different 3D related data types.*

properties are removed from any definition. The *CSS Custom Properties for Cascading Variables Module Level 1* working draft [W3C 2014a] specifies an approach for custom properties to be defined using a special discerning prefix, the double hyphen. Properties starting with this prefix are not removed and left intact inside a definition. Moreover, the value of these properties is passed literally to JavaScript to enable the definition of *new* data types. Because more than a fixed set of CSS properties is necessary to define *all* possible user-defined material parameters we base our CSS material integration on this specification. By building upon this specification we further ensure that the integration can take advantage of all future improvements to custom CSS properties, for instance CSS animations for custom properties.

In addition to custom properties we could use existing CSS properties such as "color". The different CSS related standards currently define over 200 properties and their usefulness for material definitions depends on the underlying material model. We deliberately confine the CSS material definitions to custom CSS properties. This is mainly for consistency reasons but also to avoid the necessity of overriding rules for cases when a custom property with the same name as an already existing property is defined and used by the material model. CSS overriding rules are already very powerful and complex, and introducing another rule only in order to use existing properties is not reasonable.

In the context of 3D graphics we need the vector, matrix and array data types and we have to differentiate between floating point and integer literals. For the definition of the necessary new data types we follow the proposition of the *Filter Effects Module Level 1* draft specification [W3C 2013c], which uses the typical "constructor-like" syntax to determine the type. For example, `rgba(255, 255, 255, 1)` defines a four component opaque white color in CSS.

In Listing 4 an overview of the possible data types for an example material definition is shown. The `material-model` property defines the underlying material model, e.g. a *shade.js* material description. Because this is a fixed property the prefix is not necessary. Floating point literals (see Line 4) are required to be a nonempty sequence of decimal digits containing a decimal point character, similar to the definition in the C programming language. Matrices (see Line 9) are defined in column major order following the WebGL [Khronos 2011] convention. For color definitions all existing ways to specify a color in CSS are sup-

```
mesh {
    material-model: url("./phong.js");
    --color: red;
    --shininess: 10;
}
#special-mesh{
    --color: green;
}
mesh:hover {
    material-model: url("./highlight.js");
}
```

Listing 5: *An example material definition that uses the hover pseudo-class to highlight meshes underneath the cursor.*

ported: named colors, hexadecimals, rgb and rgba constructors (see Line 5ff.). Textures can be defined using a *shorthand*. Shorthand properties are a unique feature of CSS that allow the definition of two or more related properties using a single one. For instance the shorthand *padding* can be used to define `padding-left`, `padding-right`, `adding-top`, and `padding-bottom` as a single property instead of defining all four individually. The `heightmap` property defines a texture input using such a shorthand syntax (see Line 10ff.). In contrast, the `normalmap` defines them using the individual properties. Using the `url` syntax it is not only possible to reference external resources but to also reference elements in the DOM using id references. This is especially important in the case of XML3D that uses Xflow [Klein et al. 2012] to declaratively define dataflows, whose result may be used as a parameter for the material.

Usage Examples Highlighting objects depending on the user interaction, e.g. mouseover, is very common in 3D applications. Current Dec3D approaches require event listeners to implement highlighting. Inside an event listener's implementation the target object's material is exchanged. Thus, application-logic as well as a second material definition with an id is required. Styling a DOM element depending on special states, such as mouseover, are also typical in 2D web applications. For this CSS defines a mechanism to apply a context dependent style through so called *pseudo-classes*. These classes can be used to limit the scope of a rule to elements that have a specific position in the DOM or a special state. Special states include the pseudo-class *hover*. Even though other classes may prove useful in the 3D context, this class has the most imminent meaning for 3D. Using our proposed CSS based approach for material definitions we can now use this pseudo-class for 3D (see Listing 5) in the same way as it was used for the 2D case in the introductory example. In the example shown in Listing 5 all meshes will have a default Phong material with a red color. The mesh with the id `special-mesh` will have a green color. If the user moves the cursor over a mesh its material will change to a special highlighting material. All CSS properties defined on the mesh through other rules are still available in this rule. For instance, the highlight material can use the defined `--color` property of the mesh. In the special case of the mesh with the id `special-mesh` this color will evaluate to green, for all other meshes it will be red.

Sharing and reusing existing material definitions is an important topic in computer graphics. Being able to take a well configured material and apply it to a different 3D object is, however, not trivial. Meshes can have different vertex attributes available, the rendering system may be unable to provide certain features, or the overall rendering algorithm can be different. Recent advances such as *shade.js* [Sons et al. 2014] have made it possible to write material

```
.red-material {
    material-model: url("shader.js");
    --diffuseColor: #FF0000;
    --specularColor: #FF0000;
    --reflectionColor: #FF0000;
    --shininess: 10;
}
```

Listing 6: *An example red colored material definition.*

```
.red-material {
    material-model: url("shader.js");
    --diffuseColor: var(--color, #FF0000);
    --specularColor: var(--color, #FF0000);
    --reflectionColor: var(--color, #FF0000);
    --shininess: 10;
}
```

Listing 7: *An example red material definition that provides a new configuration variable "–color".*

models that can adapt its own control-flow depending on the availability of such features and attributes. In combination with CSS based material definitions we can reach another step towards easily sharable and reusable materials in the context of declarative 3D. Materials can now be shared by referencing an external CSS file and adding a class name to a mesh, e.g. "red-plastic". Further specialization and configuration can then be done using an additional CSS rule that overrides predefined values. Adapting the material model to the available features and mesh attributes is done inside the shade.js material description.

Material models can be complex and their physical plausible configuration heavily depends on the actual implementation and can require an in-depth understanding of advanced topics in computer graphics. A user who only wants to change the overall color of a material does not care about all configuration options. Because custom CSS properties can also be used as variables it is possible for a complex material definition to provide explicit configuration points to users. Consider the example material in Listing 6. For a computer graphics expert the names of the properties are self-explanatory and how to change the final color from red to, for instance, green is obvious. Non experts, however, may not know about the difference between specular or diffuse reflections and their final contribution to the overall color. Using custom properties as variables makes it possible to provide a new simplified property `--color`, as a basic way to configure the material, while still allowing experts use all the complex configuration possibilities. Listing 7 shows the same rule but with a specific configuration variable. The user can now define the property "–color". If it is not set, a default red color will be used for all properties. Experts, however, can still use the individual properties to fully configure the different material aspects.

3.3 Camera

The extrinsic parameters of a camera are usually defined by the camera's position in the transformation hierarchy. The camera's intrinsic parameters are subject to the selected camera model. Both, X3DOM and XML3D provide camera models based on the perspective projection of a pinhole camera or on an orthographic projection. However, these idealized models do not take into account effects of real world cameras such as depth of field and motion blur or parameters of cameras for specific use-cases (e.g. stereoscopic

```
#camera {
    camera-model: url("urn:camera:perspective");
    --vfov: 61deg;
    --clip-planes: 0.1 auto;
}
```

Listing 8: *Defining a camera using CSS.*

cameras, fisheye cameras, environment cameras). There might be a wide range of possibly useful camera models that we do not want to obstruct by the definition of a fixed set of camera-related CSS properties. Thus, we follow an approach similar to our proposed material descriptions.

We propose the new fixed CSS property `camera-model`, that defines the camera model to be used for an element. A number of predefined camera models should be available via URN (e.g. `urn:camera:perspective`, `urn:camera:orthographic`). The intrinsic parameters, that depend on the selected camera model, are defined using custom CSS properties.

Listing 8 shows the definition of a camera model using the proposed approach. It also demonstrates how we can exploit already existing CSS facilities such as units and the special *auto* keyword for the definition of a camera's intrinsic properties. Numerical literals in CSS can additionally carry dimensions, two of which are degree (deg) and radians (rad), that we can reuse for the definition of the field of view. The `auto` keyword has the special meaning that the value of this property should be determined by the system. In this case the far plane will be determined automatically, using the scene's bounding volume for example, while the near plane is set explicitly. The `clip-planes` property is another shorthand property. Instead of defining two properties for the near and far plane we can use the property `clip-planes` and define the `clip-plane-near` first and the `clip-plane-far` last.

Note that with our approach, every element in the scene graph can act as a camera definition: The extrinsic parameters are derived from the transformation hierarchy, the intrinsic parameters by cascaded style properties.

Example Usage To be visible, the final image of a virtual 3D scene has to be rendered from the perspective of a specific camera. In XML3D, for example, this specific camera is defined using the `activeView` attribute on the XML3D root tag. Cameras are typically handled as special and distinct entities in a scene, still scenes are often rendered from perspectives such as the viewing direction of a light. Being able to use any object as the active camera, regardless of its actual type, is an important debugging facility and allows for special point of view shots. Without CSS we would need to define a field of view and the clipping planes on each object that may potentially be used as a camera. Because of the cascading nature of CSS the definition of these intrinsics on specific objects or on all objects in the scene can be handled by a single rule. For example, if all objects in the scene should be usable as a camera with the same field of view and the same clipping planes these properties can be defined on the root element of the scene and they will propagate to each individual child. Overrides to these "default" properties can then be defined using a more specific rule as seen in Listing 9.

```
xml3d {
    camera-model: url("urn:camera:perspective");
    --vfov: 61deg;
    --clip-planes: auto;
}

#specific {
    --vfov: 40deg;
    --near-plane: 1.0;
    --far-plane: 1000;
}
```

Listing 9: *Cascading definition of camera properties using CSS.*

```
.spotlight {
    light-model: url("urn:light:spot");
    --intensity: 100;
    --attenuation: array(1, 0, 0);
    --color: rgb(255, 255, 255);
    --cone-angle: 40deg;
}
```

Listing 10: *Spotlight definition using CSS.*

3.4 Lights

In general, light sources are objects in the scene emitting light. Yet specific light models such as point lights or spot lights are omnipresent in computer graphics. While emitting geometry is what we already have in advanced path-tracers, we have become accustomed to use fixed light models to illuminate scenes, and real-time applications require them for performance reasons. Although the set of common light models (e.g. point, spot, and directional lights) could be configured with a predefined set of properties, we can observe work towards user-definable light models, such as light shaders in XML3D and shade.js [Sons et al. 2010; Sons et al. 2014]. We therefore do not define a fixed set of CSS properties for lights but rather follow the approach taken for material definitions.

A light definition has a property called *light-model* that is used to specify the light type as a URI. In the future we expect this to be used to point to user-defined light models. The definition of a typical spotlight using CSS is shown in Listing 10.

Example Usage The effective and atmospheric placement of lights in a scene is a non trivial task, especially the lighting and shadowing with spotlights. Looking through the "eyes" of the spotlight is invaluable in such situations. Cameras and lights, however, are in general treated special and have different configuration properties. To see the scene from the spotlight's point of view a camera has to be setup and made active, the field of view has to be calculated from spotlight's cone angle, and the light's transformation has to be applied to the camera. Configuring the camera and lights using CSS allows for a very generic handling of scene objects where the mere presence of a property can turn an object into one or multiple different scene component types. To turn the spotlight defined in Listing 10 into a perspective camera we can use the CSS rule in Listing 11. Because custom properties can be used as variables we can further use the definition of the spotlight's cone angle to initialize the field of view of the camera using a simple calculation.

3.5 Summary

We propose moving four essential functionalities of declarative 3D scene descriptions (c.f. [Jankowski et al. 2013]) from DOM-based definitions to CSS-based properties: The existing *visible* and *dis-*

```
.spotlight-camera {
    camera-model: url("urn:camera:perspective");
    --vfov: calc(var(--cone-angle) * 2.0);
}
```

Listing 11: *Turning a spotlight into a perspective camera.*

```
<model src="#asset">
    <assetmesh name="part">
        <float3 name="diffuseColor">
            1.0 0.0 0.0
        </float3>
    </assetmesh>
</model>
```

Listing 12: *Overriding the color of a sub mesh inside an asset.*

play properties for the control of visibility and picking, and new properties for the definition of material, camera and light models for subgraphs of the scene. The parameters of these models depend of the model itself, hence we took the approach to use custom CSS properties in order to configure the models. We consider this approach not only to be consistent and future-proof, but it is also mandatory for programmable models such as shade.js for materials.

4 CSS and Reusable Asset Instances

The Shadow DOM [W3C 2014d] is a 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 rendered. For CSS this means that no rule can interfere with content that resides in the Shadow DOM, if it is not explicitly targeting this content with special selectors. In conjunction with the Custom Element working draft [W3C 2014b] reusable and encapsulated HTML elements can be created.

Reusing existing assets has always been an important topic in computer graphics. A new approach to configurable instances for declarative 3D has been proposed by Klein et al. [2014]. This approach is used in XML3D and provides a way to reuse existing assets, hiding all its complexity, while making explicit configuration and external changes possible. The encapsulation and isolation of this approach is very similar to that of the Shadow DOM. Listing 12 shows an example of how an asset is instanced and configured. In this example the sub-mesh with the name `part` is configured to have a different `diffuseColor` than specified in the asset's definition.

The boundary introduced by the Shadow DOM avoids CSS rules to interfere with its content. In the same way this should hold for the definition of an instanced asset. It would be confusing for the user if an unrelated CSS rule could accidentally match and modify an instanced asset in an unpredicted way. Nevertheless, should it be possible to configure, respectively style, parts of an asset using CSS, but limited to the parts explicitly targeted by the user. The specification of the part that should be affected by CSS has to be named by the user in the same explicitness as in the markup in order to configure a part of an asset. If we reconsider the instancing example from Listing 12, to change the color of the sub-mesh `part` we can use the first CSS rule in Listing 13. This rule is specifically targeting this asset mesh. More coarse-grained rules such as the second and third rule would have no effect, because assets are a boundary that prohibit CSS to propagate into hidden parts of it.

```
model[src=#asset] assetmesh[name="part"] {
    --diffuseColor: rgb(255, 0, 0);
}
model[src="#asset"] assetmesh {
    --diffuseColor: rgb(255, 0, 0);
}
model {
    --diffuseColor: rgb(255, 0, 0);
}
```

Listing 13: *Using CSS to configure an asset. The first rule specifically targets the named mesh `part` of the asset and will change its color to red. The following two rules will have no effect due to the boundary defined by the asset.*

5 Results

We evaluated the proposed integration of CSS for declarative 3D using an implementation based on the *CSS Custom Properties for Cascading Variables Module Level 1* [W3C 2014a] working draft and XML3D. Based on the discussed CSS integration and our current implementation we can revisit the initial example from Listing 1 and provide a solution for XML3D (see Listing 14). The class names are left unchanged and the paragraph texts are used as names for the external mesh references. Because our implementation relies on the custom properties specification, all fixed CSS properties discussed so far are prefixed with a double hyphen. The CSS rules for the XML3D scene shows how well CSS applies to 3D content. Beside the 3D specific properties and changes to the tag names all rules are identical to the original 2D case from Listing 2.

The ability to use CSS for the definition of materials, visibility, lights, and camera properties is not only important to reduce the necessary application logic for styling but to use all the available CSS debugging facilities of today's browsers. The DOM integration let us use the DOM inspector, CSS integration now enables the usage of the CSS rules viewers and style editors. In Figure 1 the debugger of the Firefox browser is shown. Since CSS is used for the material definition in this scene we can see on the right hand side of the debugger, in the "rules view", all material related properties of the selected mesh. This view shows which properties are defined, which are inherited, and which are overridden. It is now possible to change the visibility, color, or any other property from within this view and and see all changes immediately reflected in the 3D scene. With the "style editor" we can introduce new rules or edit existing ones to affect more than a single scene object.

Two currently existing limitations require an additional plug-in for the Mozilla Firefox Browser: activating pseudo-classes using JavaScript and observing all possible kinds of style changes. Because the browser has no information about the content of the canvas element it cannot get information about the DOM node that corresponds to the currently picked object. Therefore, pseudo-classes such as hover are not set by the browser. Unfortunately no standard JavaScript API exists to trigger these states manually. The plug-in is necessary to provide such an API.

Various modifications to the DOM can change the style of an object. An object's class list can be mutated, its id changed, or it can be moved in the DOM hierarchy. Any of this can change the final style of an element and, through inheritance, all of its descendants. While these changes can be handled by tracking those mutations using *MutationObservers* in JavaScript, external changes that affect the style of an object cannot be handled through such a mechanism. External changes that can alter the style of an object are media-queries and changes to the style through the browser's debugging facilities.

```
<style>
.disclosure:not([data-elem=""]) {
    display: none;
}
@media only screen
and (max-width: 320px) {
    #content {
        --diffuseColor: grey;
    }
    group.details > mesh {
        display: none;
    }
    .disclosure:not([data-elem=""]) {
        --diffuseColor: blue;
        display: block;
    }
    .disclosure:hover:not([data-elem=""]) {
        --material-model: url("highlight.js");
        --highlight-color: red;
    }
}
</style>
<xml3d>
<mesh src="./outer.json"></mesh>
<group id="content">
    <mesh src="first_level.json"></mesh>
    <group class="description">
        <mesh src="description.json">
        </mesh>
    </group>
    <group class="details">
        <mesh src="details.json">
        </mesh>
        <mesh class="disclosure"
            data-elem="details"
            src="ellipsis.json">
        </mesh>
    </group>
</group>
</xml3d>
```

Listing 14: *Introductory example in XML3D using CSS based styling.*

Media-queries, as explained in the initial example from Listing 1, can be used to conditionally define the style of an element. These queries are, however, not static but reevaluated if, for instance, the browser window is resized. Because JavaScript does not provide a style observer API[4] similar to the `MutationObserver`, these changes go unnoticed unless we poll each node's computed style every single frame. For efficiency reasons, the plug-in handles these cases and sends an event to the implementation in case any style change happened due to media-query evaluation or the debugging facilities of the browser.

6 Conclusion and Future Work

In this paper we propose a full CSS integration model for declarative 3D. This model allows for the definition of material, visibility, light and camera properties using CSS. Our approach exceeds all existing integration models. For the first time, we achieve an integration even beyond the initial discussed fourth level proposed by Jankowski et al. [2013].

Additionally, we have shown, how declarative 3D can benefit from the separation of content and style and in particular from the expressiveness of CSS, including techniques such as CSS inheritance, selectors and media-queries. This separation of concerns has not been done in computer graphics before and it enables the reduction of application logic and extended use of of the browsers' debugging facilities.

A common assumption of the Dec3D community is that the integration into the HTML and web technology stack makes 3D content and computer graphics more accessible to web developers. With the integration level we achieve with our approach and its accompanying implementation, we believe that we are now approaching the possibility of an empirically evaluation of this claim.

Recent development towards a new specification for media-queries specifically targets custom queries [W3C 2014c]. Based on these custom queries we can then define 3D specific queries based on the distance to the camera, the bounding volume size, or pixel coverage. With these media-queries we can explore the possibilities of a declarative level-of-detail approach in order to combine techniques such as POP Buffer [Limper et al. 2013] with CSS.

Post-processing is a vital part of every rendering pipeline and of great importance for a realistic image. The Filter Effects working draft [W3C 2013c] specifies how post-processing on HTML elements is specified in CSS. We have shown how CSS can be used for the definition of camera intrinsic properties. Following the Filter Effects approach this can be a way towards declarative post-processing definitions.

Native implementations of a declarative 3D standard are the ultimate goal of the Dec3D community group. Recent advances such as Web Components[5] and our achieved level of integration for the *xml3d.js* polyfill raises the question if native browser support is actually necessary or if we should try to improve existing technologies to remove the remaining limitations in order to provide a pure JavaScript based solution for declarative 3D on the web.

Acknowledgments

The research leading to these results has received funding from the European Union's Seventh Framework Programme under grant agreement no. 603662 (FI-CONTENT2), 632893 (FI-Core),

[4] http://xml3d.org/xml3d/specification/cssobserver/

[5] http://webcomponents.org

604674 (FITMAN), 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 - Web3D '09*, ACM Press, New York, New York, USA, 127–135.

BEHR, J., JUNG, Y., DREVENSEK, T., AND ADERHOLD, A. 2011. Dynamic and Interactive Aspects of X3DOM. In *Proceedings of the 16th International Conference on 3D Web Technology - Web3D '11*, ACM Press, New York, New York, USA, 81–87.

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 - Web3D '13*, ACM Press, New York, New York, USA, 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*, 37–46.

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 - Web3D '14*, ACM Press, New York, New York, USA, 71–79.

LIMPER, M., JUNG, Y., BEHR, J., AND ALEXA, M. 2013. The POP Buffer: Rapid Progressive Clustering by Geometry Quantization. *Computer Graphics Forum 32*, 7 (Oct.), 197–206.

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 - Web3D '10*, ACM Press, New York, New York, USA, 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, 175–184.

SONS, K., SCHLINKMANN, C., KLEIN, F., RUBINSTEIN, D., AND SLUSALLEK, P. 2013. xml3d.js: Architecture of a Polyfill implementation of XML3D. In *2013 6th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, IEEE, 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. 2014. Blast: A Binary Large Structured Transmission Format for the Web. In *Proceedings of the Nineteenth International ACM Conference on 3D Web Technologies - Web3D '14*, ACM Press, New York, New York, USA, 45–52.

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, 2011. Selectors Level 3, W3C Recommendation (work in progress). http://www.w3.org/TR/2011/REC-css3-selectors-20110929/, Sept.

W3C, 2012. Media Queries, W3C Recommendation (work in progress). http://www.w3.org/TR/2012/REC-css3-mediaqueries-20120619/, June.

W3C, 2013. CSS Conditional Rules Module Level 3, W3C Candidate Recommendation (work in progress). http://www.w3.org/TR/2013/CR-css3-conditional-20130404/, Apr.

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, 2013. Filter Effects Module Level 1, W3C Working Draft (work in progress). http://www.w3.org/TR/2013/WD-filter-effects-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. Media Queries Level 4, W3C Working Draft (work in progress). http://www.w3.org/TR/2014/WD-mediaqueries-4-20140605/, June.

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