Configurable Instances of 3D Models for Declarative 3D in the Web

Felix Klein*1,2, Torsten Spieldenner†1,3, Kristian Sons‡1,2,3, and Philipp Slusallek§1,2,3

1Saarland University 2Intel VCI 3DFKI

Abstract

The Declarative 3D for the Web initiative by the W3C [W3C 2011] connects 3D content to the Web document, intertwining it with other Web technologies known to millions of Web developers. The goal is to make 3D on the Web more accessible compared to low-level APIs such as WebGL. However, all proposals for Declarative 3D for the Web are missing an essential feature: configurable instances of structured 3D models. While instance mechanisms do exist, they all have limited capabilities to configure instances individually.

In this paper we present a new approach for configurable instances of 3D models that is integrated into XML3D. Our approach comes with a compact interface, a powerful extension mechanism to handle configurations, and an efficient data structures for efficient instancing. We demonstrate how our instance mechanism simplifies the handling of 3D models in several different application areas, including Virtual Worlds, and provide several performance results for the instancing process.


Keywords: XML3D, instancing, assets, Web3D, WebGL

1 Introduction

With the introduction of WebGL [Khronos Group 2011] the Web received a powerful technology for the creation of complex and plug-in free 3D applications. WebGL is a low-level graphics API that provides much control over hardware details and therefore covers the needs of many potential 3D applications. However, the low-level design of WebGL also differs significantly from other Web Technologies. 3D content created with WebGL is disconnected from the Document Object Model (DOM) of the Web page and therefore detached from the DOM event model, Cascading Style Sheets (CSS), and other common Web technologies. As a result, WebGL is not a technology that is easy to learn and use for Web developers.

In contrast to WebGL, the “Declarative 3D for the Web” initiative aims for a better integration of 3D graphics into Web technologies, by making 3D content part of the Web document. Since such a declaration of 3D content is comparable to HTML and compatible with many other Web technologies, it is potentially easier to pick up for Web developers and non-experts in computer graphics in general.

Most 3D scenes are composed out of a number of 3D models, whereas individual models may be reused several times. In most cases, those 3D models are structured, containing several transformed meshes with attached materials. Further, structured 3D models may embed a diverse set of additional features such as dynamic meshes, skeleton animation, or several levels of detail (LOD).

An essential feature every 3D technology has to provide is an instance mechanism for structured 3D models. With instancing, the same 3D model can be placed multiple times within the scene without duplicating its data. This does not only enhance performance by avoiding the duplication of hardware buffers, it also provides encapsulation. When instancing a 3D model, the model itself can be referenced as a black box: Authors do not need knowledge about the model’s internal structure. However, the abstraction of a black box alone is not sufficient, because often it is required to have an interface to configure each instance individually, e.g., to change surface shading properties or the current pose of the animation.

Many types of 3D applications make heavy use of configurable instances of 3D models. Examples range from tools for collaborative
3D content creation over visualization tools to Virtual World [Dionisio et al. 2013] and game scenarios. Workflow in content creation often consists of selecting a predefined 3D model, placing it into the scene, and then modifying the configuration of each instance. Applications for 3D traffic or crowd behavior visualization may employ a large number of potentially animated objects to represent individuals in these simulations. Virtual Worlds as known from popular applications such as Second Life or massively multiplayer online games such as World of Warcraft or Eve Online1 are usually populated by objects that are rendered from the same 3D model, but need to be treated individually, e.g., for individual animation states, colors, or equipment.

Despite its high relevance for 3D applications, configurable instances of 3D models remain a poorly supported feature among the current proposals of Declarative 3D for the Web. X3DOM [Behr et al. 2009] does support the instancing of 3D models with its DEF/USE mechanism, effectively blackboxing their content, but lacks the option to configure instances individually. XML3D [Sons et al. 2010] only supports the reuse of generic data and shaders and therefore requires the replication of the 3D model’s structure for each instance.

In this paper we propose a new mechanism for configurable instances of 3D models for Declarative 3D in the Web. This mechanism is integrated into XML3D and leverages the data composition and processing capabilities of X3DOM [Klein et al. 2012a]. Our approach supports the instancing of structured 3D models with a compact interface, provides a powerful extension mechanism to configure the content of each instance, and makes use of a data structure that allows for fast and efficient instancing.

The paper is structured as follows: First, we explain different kinds of instancing approaches in Section 2, followed by an explanation on how the current XML3D version handles 3D models in Section 3. Afterwards we explain the design and interface of our instance mechanism in more detail in Section 4, compare it to other approaches in Section 5, and give a few insights about the implementation and optimizations in Section 6. Following that, we present several use-cases that make use of our new instance mechanism in Section 7. Finally, we will present several performance results in Section 8, followed by Conclusion and Future Work.

2 Related Work

In computer graphics the reuse of resources can be achieved through many different techniques. Reasons for the reuse of graphics resources include memory and runtime performance, reduction of overhead in the rendering pipeline, scene management, and rapid creation of complex scenes using encapsulation of functionality. In the following, we shortly discuss several techniques that allow for reuse of graphics resources and are relevant for declarative 3D in the browser.

Geometry Instancing Geometry instancing is a technique that enables efficient rendering of multiple mesh instances with differentiating shader parameters. This technique is a batch processing optimization that reduces the overhead that comes with activating the differentiating shader parameters one by one and doing the draw calls separately. Geometry instancing is a concept orthogonal to other instancing techniques and can be used by every rendering system capable of identifying copies of meshes. Geometry instancing is available in WebGL through the \texttt{ANGLE\_instanced\_arrays} extension. It was first introduced in Direct3D 9 [Microsoft 2003]. Nowadays several similar but more flexible approaches with even less driver overhead exist, e.g., \texttt{glMultiDrawArraysIndirect} in OpenGL 4.3 [Khranos Group 2012b].

Data Level Instancing The next level of reuse is based on data blocks. This technique reduces memory consumption. In the best case, graphics data is stored in buffers in the graphics card’s local memory and only referred by the application, possibly by multiple objects.

X3D [Web3D Consortium 2011] supports the reuse of data with its DEF/USE mechanism. Compatible are all nodes that represent data entries such as \texttt{<Coordinate>}, \texttt{<Normal>}, or \texttt{<Color>}, as well as arbitrary vertex attributes via \texttt{<X3DVertexAttributeNode>}. Some X3D geometry nodes such as the commonly used \texttt{<IndexedFaceSet>} allows multiple indices (position, normal and color indices) and per face colors and normals, which are all features not supported by recent graphics APIs. As a result, the data entries need to be pre-processed before being stored on the GPU, which makes it difficult to share GPU buffers. Only one specific configuration of \texttt{IndexedTriangleSet} can be directly mapped to GPU buffers.

XML3D on the other hand provides fine-granular data composing based on generic \texttt{<data>} elements. If used as vertex attributes of a mesh, these entries directly correspond to OpenGL Vertex Buffer Objects (VBOs). The set of vertex attributes can then be packed into Vertex Array Objects (VAOs) and activated in a single API call to reduce overhead.

In contrast to X3D’s DEF/USE mechanism, a \texttt{<data>} element can also refer to resources contained in external documents. In that case, there is no interface to change the content of the external resource. This knowledge can be exploited to optimize the data structures internally. With the data composition model it is possible to compose mesh data from internal and external resources, and to override shader attributes from mesh instances.

Sub Graph Instancing X3D supports the reuse of arbitrary sub-scene graphs using the DEF/USE mechanism. This extends the data level instancing, because it includes structure, transformations, materials, and even nodes with a higher abstraction level, e.g., humanoid animation nodes. Although this is an established mechanism in scene graphs, it comes with a number of issues in particular in the DOM context. At first, the scene structure represented in the DOM tree becomes a Directed Acyclic Graph (DAG). Apart from the fact that the complexity of CSS inheritance rules increases a lot, the main issue of a DAG is that picked objects do not map to a single DOM element and can only be differentiated by examining the path of picked objects. This does not match with the DOM event system, and is therefore not compatible with libraries build on top of DOM events. Moreover, it is a concept web developers are required to learn.

In addition to the issues that come with sub graph instancing, the usefulness of this concept is very limited, because the instances can only be modified collectively. This makes sub graph instancing useless for all the many cases the instances shall be configured (e.g., animated) individually.

Higher-Level Instancing A logically separated unit can also include behavior, i.e., intrinsic scripting and events. The idea of these higher-level instancing is to be able to define modular objects (components) in the sense of reusable building blocks. As long as the interface does not change, the component and the instancing scene can be modified independently.

Several approaches exist that provide instancing capabilities based on components. In HTML, a well-known way to inline existing

1\url{www.secondlife.com}, \url{www.battle.net/wow}, \url{www.eveonline.com}
XML3D is a minimal extension to HTML5 for 3D Models in XML3D. It defines a small set of new elements to describe a scene graph with 3D geometry, surface shading, and lighting. In addition, it features the declaration of generic data structures as well as the declarative language Xflow [Klein et al. 2012a] used to perform processing on this data. Thanks to its generic design and its integration with existing Web standards, XML3D supports a wide range of 3D related features, such as skinned mesh animations and Augmented Reality [Klein et al. 2012b], without adding a large amount of new DOM elements.

However, one major downside of XML3D is its lack of an instance mechanism for structured 3D models. While XML3D does provide the means to reuse generic data and shaders, it requires users to duplicate the entire structure of instance 3D models.

Our goal is to find a new approach to 3D model instancing that integrates well with the existing concepts of XML3D. Thus, we first have a look on how 3D models are currently handled in XML3D. We will highlight what elements are required to describe 3D models, how they are handled when instancing the model, and how they need to be modified to configure each instance.

### 3.1 Instancing of 3D models

Structured 3D models in XML3D are declared using a combination of `<mesh>`, `<group>`, `<transform>`, and `<shader>` elements (see Figure 2). It is possible to assign `<shader>` and `<transform>` element to several `<group>` elements via a reference to their (local) URL and their document id. In addition, generic buffer data can be declared in separate `<data>` elements and shared among several meshes via external URL based references or internal references based on document ids.

The first limitation becomes apparent when the user plans to instance the whole content of a `<group>`. To do so, it is required to duplicate each and every `<group>` and `<mesh>` node in the sub tree. On top of that, shared data is re-exported two possible ways: Either they are shared among all instances and must not be duplicated, or they are duplicated, which makes it mandatory to adapt their document id and related references as such ids must be unique inside the document.

It is possible to structure XML3D content in a way to simplify the instancing process, which involves extracting all shared content into external files such that the remaining content only includes external references and can be copied without further modifications. However, all XML3D content without this clean separation will be hard to instance.

Overall, the explicit duplication of the 3D models structure is inconvenient for simple instancing.

### 3D Models in XML3D

XML3D [Sons et al. 2010] is a minimal extension to HTML5 for interactive 3D content. It defines a small set of new elements to de-
4.1 Basic assets definition and instancing

We introduce the concept of assets, a structure that describes a complete 3D model with transformed geometry, shading, and dataflow processing. A basic asset with a number of meshes is defined using the <asset> and <assetmesh> elements:

```
<asset id="exampleAsset">
  <assetmesh shader="#woodShader"/>
  <!-- Mesh buffers declared here -->
</asset>
```

The <assetmesh> element includes all attributes of a regular <mesh> element. In addition, shaders and transformations can be attached to each <assetmesh> directly.

An asset can be instanced with the <model> element, which is placed inside the scene graph and refers to an internal or external asset via the src attribute:

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

Consequently, all meshes declared in the asset will be visible in the scene graph, correctly transformed, and with shaders attached.

4.2 Assets with shared data

Commonly, 3D models are composed in a way to maximise the reuse of data. A common structure is one set of vertex buffers shared by several meshes with individual index buffers. For our asset declaration we can share buffers by using already existing data composition techniques of XML3D and Xflow:

```
<data id="sharedAttribs*">
  <!-- Shared vertex buffers -->
</data>
```

```
<asset id="sharedExampleAsset">
  <assetmesh shader="#wood" >
    <data src="#sharedAttribs"/>
    <int name="#index">0 1 2 3 ...</int>
  </assetmesh>
  <assetmesh shader="#iron" >
    <data src="#sharedAttribs"/>
    <int name="#index">1045 1046 1047 ...</int>
  </assetmesh>
</asset>
```

However, in addition to these techniques we introduce a new data sharing concept for assets with the <assetdata> element and the name and includes attribute:
Finally, the chain.

First recursively resolve its own includes-connections and extension that, before we connect an asset entry to another via refers to the fully resolved version of the asset entry. This means includes be assigned to all meshes. Note, that this behavior is due to the referred asset as well with exactly the same syntax. Thus, we can extend the shared generic data content of asset entries. For an named entries of the source assets. In general, this enables us to declared 

In the following example, we extend the shared \texttt{<asset>}

4.3 Extension of assets

The concepts presented so far already allow for a compact definition and instancing of static 3D models. However, in order to support configurable 3D models as discussed in Section 3.2, we need to be able to modify our 3D model instances in certain ways, e.g., to set the key value of the animation.

To support configuration, we add a powerful extension mechanism to our \texttt{<asset>}

In the following example, we extend the \texttt{#sharedExampleAsset2} asset of the previous subsection by adding a new color attribute to the shared vertex buffers:

The original source asset had all \texttt{<assetmesh>} elements referring to the shared \texttt{<assetdata>} element via the \texttt{includes} attribute. Since we now extend the shared \texttt{<assetdata>}, the new color buffer will be assigned to all meshes. Note, that this behavior is due to the semantic of the connection via the \texttt{includes} attribute, which always refers to the fully resolved version of the asset entry. This means that, before we connect an asset entry to another via \texttt{includes}, we first recursively resolve its own includes-connections and extension chain.

Finally, the \texttt{src} attribute of \texttt{<model>}

Overall, the combination of the dataflow processing capabilities of Xflow and the new asset extension mechanism allows for an instancing of complex animated 3D models with a minimal yet flexible interface for configurations.

5 Comparison to X3D Prototypes

After presenting the design of our new instance mechanism, we compare it now to another high-level instancing approach of X3D: Prototypes. X3D Prototypes support the instancing of arbitrary
scene graph content with plenty of options for configuration. Our instance mechanism supports a similar degree of configuration, but is limited to structured 3D models.

On the other hand, we support the attachment of dataflows to arbitrary parts of the 3D model, such as mesh data, shader parameters and transformations. This feature can be used in many ways to simulate functionality of Prototypes. For instance, we can simulate modifications in a transformation hierarchy by processing transformations in a dataflow and connecting the result to mesh entries. In general, we can compose our dataflow to provide the most compact interface for the desired functionality. For example, if we want to be able to rotate a wheel we accept a float value specifying the angle around a fixed axis, computing the matching transformation within the dataflow.

Another limitation compared to Prototypes is the lack of embedded Sensors and Routing nodes. This is because XML3D doesn’t define these kind of elements, leaving all interactive functionality to JavaScript. Consequently, our instance mechanism is a perfect fit, providing a compact, yet flexible interface for JavaScript programs to control any aspect of the instance.

In contrast to Prototypes, we do not support the instancing of view points and light sources. The instancing of view points is usually not relevant and in fact may result in strange behavior with respect to the selection of the active view point. The instancing of light sources may be more relevant and our instance mechanism might be extended to support them in a later version. Up to this point, however, light sources are usually carefully placed independent of 3D models to achieve a good lighting without putting too much stress on the performance.

Finally, we have the advantage that our `<asset>` elements can be referred easily from external files. In contrast, for external Prototypes the user has to replicate the interface of the Prototype in the main document which is error prone.

6 Implementation

We extended the xml3d.js [Sons et al. 2013] library to support our new instance mechanism, focusing on an efficient instancing process with little memory overhead.

Our past experiences have shown that a great number of DOM elements will result in a large memory footprint and prolonged load times, even for structured content without embedded vertex buffer data. Fortunately, the design of our `<asset>` structures allows for an instancing process that doesn’t require the replication of DOM elements. Instead, we map each instance to light-weight rendering structures, which were introduced in recent versions of xml3d.js and are separated from the DOM. For instance, just as each `<mesh>` element is mapped to a `RenderObject` and each `<group>` to a `RenderGroup`, a single `<model>` element is mapped to a whole sub graph of these two structures.

In addition to the optimized creation of new instances, we also need an efficient process to update the configuration of each instance. This is critical e.g., for animated characters where the pose is modified each frame. Consequently, modifying the generic data content of an `<assetmesh>` or `<assetdata>` entry should quickly result in an update of the rendering structures of all connected `<model>` instances. However, with the options of extending asset entries and connecting them implicitly with the `includes` attribute, a `<model>` instance may contain a complex connection of generic data that needs to be considered during updates. We efficiently handle such updates by mapping these data connections to an internal Xflow graph. We structure this graph such that it correctly combines all the generic data of the asset entries according to their extension and connection properties. See Figure 4 for an example. With this approach, consequent data updates inside asset entries are automatically propagated by Xflow, which is designed to efficiently handle dataflow modifications.

7 Use-Cases

In the following, we will discuss some use-cases in more detail and show how they profit from the simple and intuitive scene description introduced by our instancing implementation.

7.1 Collaborative Content Creation

An type of application that became more and more popular in the scope of 3D Web-applications during the last years is collaborative 3D content creation. For example, DiFac [Sacco et al. 2007] showed how the design of industrial production sites benefits from virtual design and evaluation. Smparounis et al. [Smparounis et al. 2009] presented a browser-based solution of a collaborative prototype designer. Zinnikus et. al implemented a browser-based solution for factory design and evaluation, using declarative 3D for design and simulation [Zinnikus et al. 2013].

In these applications, a common step in the workflow is to choose a 3D model from a set of provided models as building block of the final 3D scene, and create an instance of the respective asset in the scene, comparable to recent game level editors. Obviously, this kind of use-case profits significantly from our improvement in XML3D. Placing an asset in the application then corresponds to nothing but creating a `<model>` node in the DOM. This in turn...
means that the DOM structure reflects the actual scene content very well. Each `<model>` tag corresponds to one building block in the scene, other than before, when each building block introduced a full copy of a possibly complex declarative geometry representation.

Our implementation of instancing in addition allows for custom changes to individual instances. Instances can for example be scaled individually or receive a distinct color by overriding the respective entry in the exposed `<assetdata>`.

Figure 5 (left) shows an example of a collaborative construction site planning tool. Each of the containers, trucks, or cranes are instances of a shared asset definition. Although the underlying geometry may be rather complex, each object can be instanced by just a single `<model>` node.

### 7.2 Customizable User Avatars for Virtual Worlds

There are also investigations of how to transfer Virtual World applications like Second Life to the Web browser: Byelozyorov et al. [Byelozyorov et al. 2012] presented an application that creates virtual versions of real world cities using a declarative 3D representation for the graphics. Dahl et al. [Dahl et al. 2013] presented a browser-based Virtual World Web-client that uses three.js for rendering.

In such a Virtual World scenario, users who explore the world are usually represented as 3D avatars. As the number of connected users in a virtual environment can range from a few dozen to several hundreds [Dionisio et al. 2013], it is obvious that instancing geometry for user avatars is highly desirable. In addition, the instanced geometry needs to be customizable, not only concerning details that may vary between different users, but also considering individual poses of objects.

With our approach, it is easy to re-use geometry for a number of avatars and still allow for various visual variations. We can for example override `<assetdata>` nodes to provide an interface for changing the color that is used by a certain shader (see Figure 5, bottom). Apart from configuring the color of skin or hair, users may also change the appearance of the clothing of their avatar by switching to different textures. Moreover, by exposing `<assetmesh>` nodes that contain references to mesh `<data>`, we can also change actual geometry that is part of the avatar mesh, as for example carried equipment in a gaming scenario.

By exposing individual animation keys per avatar model, the different instances can be animated individually. This is obviously a strict requirement for a Virtual World scenario, as users also perform individual actions and interactions with the world. For this, we need means to display the interactions by selecting the respective avatar animations and display them independent of other entities. Instancing a customized avatar model could for example look like this:

```xml
<model src="avatar.xml#asset">
  <assetdata name="walkingKey">0.3</assetdata>
  <assetmesh shader="sword.xml#shd" includes="handXfm">
    <data src="sword.xml#mesh" />
  </assetmesh>
</model>
```

Using this code, the model would be capable of animation (by exposing the respective animation key) and have the option to change the object it is carrying in its hand by replacing the respective `<assetmesh>` entry. Including the `handXfm` entry provides the transformation to align the geometry with the hand’s position and orientation of the current pose.

### 8 Performance Results

In the previous section, we have already shown how different use-cases may profit from the simpler scene description. However, the resulting smaller size of the DOM tree and the fewer number of DOM nodes that have to be inserted compared to copying the geometry representation should also influence both memory consumption and page load times.

To verify this assumption and estimate the benefit of instancing compared to the previous method of copying the complete DOM representation of the instanced objects to the Web page, we load various instances of the same model, first with the previous approach, second with our new instancing method. We use 3 different models for this (see also Figure 6):

- **Fire Truck**, which is a static model that consists of 41 nodes for geometry definition. This rather complex representation results from various different shaders that all need to be assigned to distinct group nodes. However, as the geometry is completely static, and thus no animation keys need to be exposed, we can instance a Fire Truck asset with just one `<model>` node.

- **Natalie**, which consists of only 11 nodes for geometry definition and one `<float>` node as animation key. Thus we can instance Natalie with one `<model>` node that contains one `<assetdata>` node containing the animation key.

- **Sniper**, which is animated and consists of 29 nodes for geometry definition and one `<float>` node as animation key. With our instancing approach, we also need only one `<model>` and one `<assetdata>`, as for Natalie as well.

The generated scene that consists only of the instanced assets is created on a server in an abstract representation. We connect to the server with an XML3D-based client. Upon receiving information about which assets are present in the remote scene, the client retrieves the XML3D representation of the models and adds them dynamically to the DOM, first by the old approach of copying the complete representation for each of them, second by using the presented instancing approach. This client also adds an enclosing group node to each instance to allow for individual positioning and orientation. This behavior corresponds to what one would expect in the use-cases described in Section 7.

Measurements are done in Google Chrome Web Browser. Assets and test application are hosted on the same machine. This eliminates delays in the measurement from retrieving geometry definition from a remote location and allows for a better comparison of
Table 1: Page size and load times for different scene complexity. The new approach keeps the DOM significantly smaller, which results in a speedup of page load times of at least 85% in most cases. For the complex models of Fire Truck and Sniper, creating 1,000 copies was not possible by cloning geometry due to process memory limitations of the Chrome browser.

<table>
<thead>
<tr>
<th>Model</th>
<th>Copy</th>
<th>Instance</th>
<th>Copy</th>
<th>Instance</th>
<th>Copy</th>
<th>Instance</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>FireTruck</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50 Inst.</td>
<td>2.100</td>
<td>100</td>
<td>105 MB</td>
<td>50 MB</td>
<td>2.36 s</td>
<td>0.285 s</td>
<td>87.9 %</td>
</tr>
<tr>
<td>500 Inst.</td>
<td>21.000</td>
<td>1,000</td>
<td>777 MB</td>
<td>111.0 MB</td>
<td>79.3 s</td>
<td>5.15 s</td>
<td>93.5 %</td>
</tr>
<tr>
<td>1000 Inst.</td>
<td>42.000</td>
<td>2,000</td>
<td>N/A</td>
<td>178 MB</td>
<td>N/A</td>
<td>19.0 s</td>
<td>N/A</td>
</tr>
<tr>
<td>Natalie</td>
<td>Copy</td>
<td>Instance</td>
<td>Copy</td>
<td>Instance</td>
<td>Copy</td>
<td>Instance</td>
<td>Speedup</td>
</tr>
<tr>
<td>50 Inst.</td>
<td>650</td>
<td>200</td>
<td>85 MB</td>
<td>77 MB</td>
<td>1.34 s</td>
<td>0.44 s</td>
<td>67.1 %</td>
</tr>
<tr>
<td>500 Inst.</td>
<td>6.500</td>
<td>2,000</td>
<td>337 MB</td>
<td>235 MB</td>
<td>59.1 s</td>
<td>7.2 s</td>
<td>87.8 %</td>
</tr>
<tr>
<td>1000 Inst.</td>
<td>13.000</td>
<td>4,000</td>
<td>640 MB</td>
<td>384 MB</td>
<td>224.1 s</td>
<td>31.8 s</td>
<td>85.8 %</td>
</tr>
<tr>
<td>Sniper</td>
<td>Copy</td>
<td>Instance</td>
<td>Copy</td>
<td>Instance</td>
<td>Copy</td>
<td>Instance</td>
<td>Speedup</td>
</tr>
<tr>
<td>50 Inst.</td>
<td>1.500</td>
<td>200</td>
<td>87 MB</td>
<td>74 MB</td>
<td>3.2 s</td>
<td>0.43 s</td>
<td>86.5 %</td>
</tr>
<tr>
<td>500 Inst.</td>
<td>15.000</td>
<td>2,000</td>
<td>341 MB</td>
<td>241 MB</td>
<td>164 s</td>
<td>7.7 s</td>
<td>95.3 %</td>
</tr>
<tr>
<td>1000 Inst.</td>
<td>30.000</td>
<td>4,000</td>
<td>N/A</td>
<td>381 MB</td>
<td>N/A</td>
<td>31.4 s</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 2: Render frame rate for Xflow animations for copied geometry compared to instanced geometry. The new approach does not introduce a bottleneck in Xflow processing.

<table>
<thead>
<tr>
<th>Model</th>
<th>Copy</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natalie</td>
<td>5 Inst.</td>
<td>60 fps</td>
</tr>
<tr>
<td></td>
<td>10 Inst.</td>
<td>39 fps</td>
</tr>
<tr>
<td></td>
<td>20 Inst.</td>
<td>21 fps</td>
</tr>
<tr>
<td>Sniper</td>
<td>5 Inst.</td>
<td>60 fps</td>
</tr>
<tr>
<td></td>
<td>10 Inst.</td>
<td>36 fps</td>
</tr>
<tr>
<td></td>
<td>20 Inst.</td>
<td>19 fps</td>
</tr>
</tbody>
</table>

load times, as they are independent of the file size of the different asset definition files. Thus, the page load times we compare result only from DOM construction when building the scene dynamically.

The respective resulting DOM sizes, peak memory usage reached by the browser process, and page load times are listed in Table 1. We can see that page load times decrease drastically when switching from copying to instancing geometry. The impact is largest for the complex, but static Fire Truck Model. All of the complex geometry can be hidden in the external asset definition from which we can create an instance with just one `<model>` node (plus one `<group>` node for individual positioning). Also complex animated models, like Team Fortress Sniper, profit tremendously. As we cannot reduce the number of nodes for Natalie much, the effect is lowest here.

For Fire Truck and Sniper, we hit the browser tab process memory limit when trying to create 1,000 copies of the model with the old approach. In these tests, the browser process crashed completely. Thus, the approach of actual instancing the models does not only result in a remarkable speed-up, but in the end also allows to display much larger scenes.

Instancing should not influence existing features of XMLElement. Especially XFlow data processing that is used for example for animating skinned characters should not suffer from dealing with the new format. Skinning operators in XFlow are entirely computed in JavaScript, thus evaluation needs to be efficient. To ensure that instancing does not introduce a bottleneck here, we have compared the frame rate for different numbers of instances, both when copying the geometry representation and when using the new instancing feature. The results are listed in Table 2. The frame rates are comparable for both approaches, thus our implementation of instancing does not introduce computational overhead.

9 Conclusion and Future Work

In this paper, we introduced a new mechanism for configurable instances of 3D models in the area of Declarative 3D for the Web. With the addition of merely 4 new types of DOM elements, the mechanism provides a compact interface to instance arbitrary 3D models with embedded dataflow processing for mesh animations and other features. The powerful extension mechanism allows for a flexible and detailed configuration of each instance, be it to specify the current pose of the animation, replace shading parameters, or to replace entire shaders or parts of the geometry. Additionally, we demonstrated how the flat structure of our assets simplifies the efficient implementation of the instancing process. Finally, we demonstrate how our new mechanism can be used in a range of use-cases and how it decreases memory usage and shortens page load times.

Of course, there still remain use-cases that collide with certain limitations in our instancing interface.

One limitation of the current implementation is lacking functionality to merge two independent assets. While it is possible to extend an `<asset>` element in arbitrary ways, e.g., by listing more `<assetmesh>` elements, it is not possible to merge all `<assetmesh>` entries of two or more `<asset>` elements. To tackle this limitation, we plan to integrate a concept of nesting `<asset>` elements within each other. This concept will also include an option to expose configuration properties of nested `<asset>` elements to keep the merged asset configurable as a whole. For instance we could place the asset of a propeller hat inside the asset of the person and expose the configuration value to rotate the propeller of the hat to the final asset.

10 Acknowledgement

The character models of the robots were provided courtesy of Disney Research Zurich. The character model of Team Fortress Sniper was provided courtesy of Valve Software.

The research presented in this paper has been generously supported by the Intel Visual Computing Institute, FI-CONTENT, FI-CONTENT, FI-CONTENT.
WARE/FI-Core, and FITMAN, which are part of the EU Future Internet PPP program, and the EU FP7 project VERVE.

References


