SRC - A Streamable Format for Generalized Web-based 3D Data Transmission

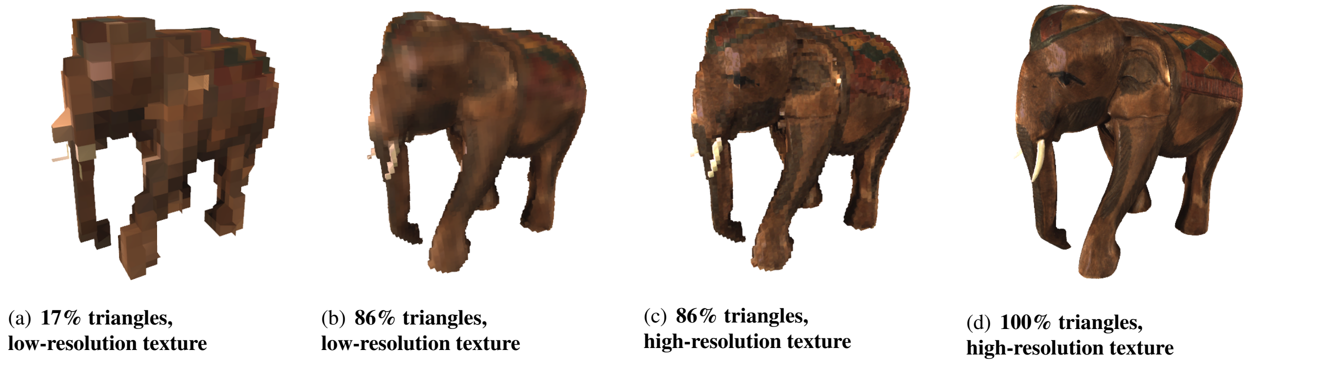

(a) **17% triangles, low-resolution texture** (b) **86% triangles, low-resolution texture** (c) **86% triangles, high-resolution texture** (d) **100% triangles, high-resolution texture**

Figure 1: *Streaming of mesh data, progressively encoded with the POP Buffer method, using our proposed SRC container format. We minimize the number of HTTP requests, and at the same time allow for a progressive transmission of geometry and texture information, using interleaved data chunks. Our proposed format is highly flexible, well-aligned with GPU structures, and can easily be integrated into X3D.*

Abstract

A problem that still remains with today's technologies for 3D asset transmission is the lack of progressive streaming of all relevant mesh and texture data, with a minimal number of HTTP requests. Existing solutions, like glTF or X3DOM's geometry formats, either send all data within a single batch, or they introduce an unnecessary large number of requests. Furthermore, there is still no established format for a joined, interleaved transmission of geometry data and texture data.

Within this paper, we propose a new container file format, entitled Shape Resource Container (SRC). Our format is optimized for progressive, Web-based transmission of 3D mesh data with a minimum number of HTTP requests. It is highly configurable, and more powerful and flexible than previous formats, as it enables a truly progressive transmission of geometry data, partial sharing of geometry between meshes, direct GPU uploads, and an interleaved transmission of geometry and texture data. We also demonstrate how our new mesh format, as well as a wide range of other mesh formats, can be conveniently embedded in X3D scenes, using a new, minimalistic X3D ExternalGeometry node.

CR Categories: I.3.2 [Computer Graphics]: Distributed/network graphics— [I.3.6]: Methodology and Techniques—Graphics data structures and data types

Keywords: WebGL, X3D, X3DOM, Compression, 3D Formats, Streaming

1 Introduction

Recently, various efforts have been made in order to design file formats for transmission of 3D geometry, for the use with high-performance 3D applications on the Web. The ultimate goal is to design a solution that scales well with large data sets, enables a progressive transmission of mesh data, eliminates decode time through direct GPU uploads, and minimizes the number of HTTP requests. Notable results include the WebGL-Loader library [Chun 2012], X3DOM BinaryGeometry Containers [Behr et al. 2012], and the GL Transmission Format (glTF)[1].

However, a problem that still remains is the lack of progressive transmission in almost all of those formats. While X3DOM provides an experimental implementation of progressive geometry transmission, it can only obtain batches of index and vertex data over multiple HTTP requests, which potentially becomes a huge drawback with larger scenes. The Khronos group's glTF proposal, on the other hand, is able to deliver an arbitrary number of mesh data buffers within a single file, but it completely lacks any mechanisms for progressive transmission.
The overall lack of a working format for progressive transmission of binary mesh data is also due to the missing support for progressive downloads of binary data in current implementations of the *XmlHTTPRequest* (XHR) specification. Luckily, there is already a W3C draft for the so-called *Streams API*, which will extend XHR and solve this problem in the near future. Figure 2 illustrates the advantage of using the Streams API over the current XHR. Still, there is no existing 3D format which is able to fully exploit this upcoming API.
Finally, there is no established format that allows an interleaved transmission of texture data and mesh data. As a consequence, the point in time at that a textured mesh is fully loaded depends on at least two different downloads, and is therefore pretty random.

Within this paper, we propose the *Shape Resource Container* (*SRC*) format, a new file format for progressive transmission of 3D geometry and texture data. Our contributions can be summarized as follows:

[1]https://github.com/KhronosGroup/glTF

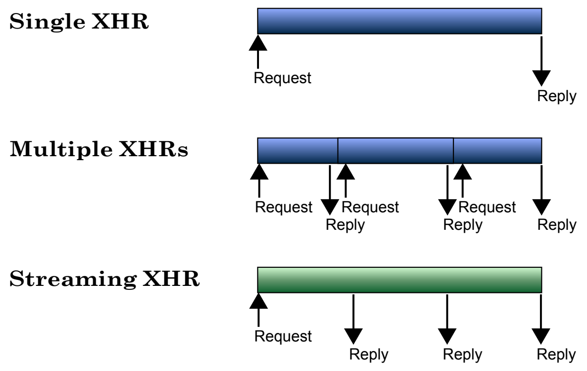


Figure 2: *Different methods of using XmlHTTPRequest. Requesting a single file prevents processing until the transmission has completed. Partial requests allow early processing, but at the costs of additional overhead for each request. The Streams API provides progress events, without any additional request.*

- We introduce *buffer chunks* as a new concept for progressive, interleaved transmission of indices, vertex attributes, and even textures, with an arbitrary small number of HTTP requests.
- We show how to efficiently speed up progressive texture retrieval, by including support for compressed texture data in our proposed format.
- We show how to embed our format, as well as other external geometry formats, into X3D scenes, by proposing a new, minimalistic X3D node.
- We demonstrate several use cases that are all covered by our format, showing its high flexibility, as well as its potential suitability for becoming a future standard.

Our proposed *ExternalGeometry* node serves as a very minimalistic interface to SRC content, and we believe that it could easily be translated in order to use our format within other declarative 3D solutions [Sons et al. 2010; Jankowski et al. 2013], with non-declarative 3D frameworks on the Web (like *Three.js*[2], for instance), or even within 3D desktop applications.

This paper is structured as follows: Within Section 2, we summarize the state of the most relevant previous work. This section also contains brief comparisons between our proposed format and the currently existing ones. Section 3 introduces our proposed SRC format in detail, and especially discusses its novel aspects. In Section 4, we describe several use cases that demonstrate the efficiency and flexibility of our proposed X3D node and mesh data format, showing that it serves as a generic geometry container in high-performance X3D applications. Finally, Section 5 concludes with a summary.

2 Previous Work

X3DOM Binary Geometry. A major drawback of existing declarative 3D mesh containers, in XML3D [Sons et al. 2010] as well as in X3DOM [Behr et al. 2009], is the missing ability to merge multiple drawable patches of a single mesh into a single shape. This might be necessary because of several reasons, like, for example, view-dependent streaming and geometry refinement, or WebGLs restriction of allowing only 16 bit indices during rendering. Consider, for example, the armadillo model from Fig. 3, which has been subdivided into three different chunks (in this particular case, the main reason was the mentioned 16 bit index limit). In X3DOM, this subdivision is also reflected in the declarative layer, by using three *Shape* nodes, each containing a separate *Appearance* node and a separate *BinaryGeometry* node [Behr et al. 2012]. As a consequence of this separation, the InstantReality AOPT and X3DOM visualization pipeline does not allow to encode and transmit the three sub-meshes all in one file. Furthermore, an X3DOM author must maintain three different *Shape*, *Geometry* and *Appearance* nodes, instead of just one. Finally, this tight coupling of the rendering representation with the scene-graph and the transmission format also potentially leads to large, cluttered HTML files.
Our proposed *ExternalGeometry* node and SRC format solve this problems by allowing a random mapping between the number of transmitted files and identifiable *Shape* nodes (cp. Fig. 7).

XML3D Mesh Data Composition. The XML3D declarative 3D framework includes a powerful data flow definition concept, entitled *XFlow* [Klein et al. 2012]. The concept is based on a *data* element, which represents a mesh data table with *data fields* (for instance, indices, vertex positions, vertex normals and vertex colors). Since all data elements can include other data elements, and since they may also add own definitions for single data fields, dynamic composition, overriding and re-use of mesh data among several mesh instances is possible. Still, there is no binary format for arbitrary pieces of mesh data, overriden attribute arrays are usually specified as strings. This in turn causes huge decode overhead, and it leads to unnecessarily large HTML files. Furthermore, a progressive transmission of mesh data, as it is enabled by our proposed format, is not possible within XML3D.

glTF. The *GL Transmission Format* (*glTF*), as proposed by the Khronos Group, is an optimized format for straightforward transmission and rendering of 3D assets. While COLLADA [Arnaud and Barnes 2006] has been designed as a format for asset exchange between 3D authoring tools, glTF is intended to be a delivery format, specifically designed for rendering, and not intended for further authoring.
A glTF scene description, transmitted as a separate JSON file, along with the texture images and binary mesh data containers, is always divided into several parts. The *buffer* layer contains a basic, raw data description, usually by referring to an external binary file, which is, on the client side, represented as an *ArrayBuffer* object, being the raw result of an *XmlHTTPRequest* that triggered the download. On top of that buffer layer, a *bufferView* layer manages several sub-sections of buffer objects, where each sub-section is usually represented as a separate GPU buffer on the client side. A buffer might, for example, be subdivided into two separate bufferViews that each map to a GPU buffer, one for index data and one for vertex data. On top of the bufferView layer, there is a layer with *accessor* objects (representing the graphics API's views on bufferView objects) that realize indices and vertex attributes. Two different accessors, one for normal data and one for position data, for example, might then refer to different parts of a single bufferView, potentially in an interleaved fashion. The highest hierarchical level of mesh data within glTF is represented by the *mesh* layer. A mesh entry always refers to one or more attribute accessors and index data, along with a material and a primitive type used for drawing (e.g., `TRIANGLES`).
Because of its straightfowrward, structured design, mapping very well to client-side GPU structures, glTF might seem like an ideal solution for many 3D Web applications. However, our aim, which was to have a flexible mesh data transmission format for the use within high-performance X3D scenes, could not been reached with glTF. Amongst others, this was due to the following reasons:

[2]threejs.org

- The glTF specification does not support any form of progressive transmission of mesh data.
- The glTF specification does not allow for an interleaved transmission of mesh geometry data and texture data, and it does not support any GPU-friendly texture encoding.
- The JSON-based scene description of glTF partially overlaps with existing concepts in X3D. Examples are lights, shaders and other material descriptions, and a node hierarchy.

At the moment, there are no profiles available in glTF, so there is no possibility to request a description which contains solely geometry data. Within this paper, we provide a solution to all mentioned problems by introducing our SRC format. We also present a new X3D node, entitled *ExternalGeometry*, which has a well-defined, minimal interface to the external mesh data description. We especially show how assets within our proposed SRC format can be referenced within the X3D description, and why the proposed concept is superior to existing approaches in XML3D, X3D and X3DOM.

Progressive Geometry Transmission. Progressive Meshes, as originally proposed by Hoppe et al. [Hoppe 1996], have been extensively studied within the past two decades, with a strong focus on compressing progressively transmitted mesh content, in order to optimize the rate-distortion performance [Peng et al. 2005]. Since it seems like an ideal candidate technology for the 3D Web context, there have been various efforts to port progressive meshes to the Web, for example, by integrating the method with X3D [Fogel et al. 2001; Maglo et al. 2010]. Recently, Lavoué et al. have proposed a progressive mesh method that aligns well with current 3D Web technology [Lavoué et al. 2013]. However, this method is currently only working with manifold geometry, and it might still introduce significant decode times, especially on mobile devices with low compute power.
As an alternative for a progressive, direct upload of downloaded mesh data to the GPU, without any decode overhead, there are currently two known alternatives. The first one is to converting the mesh to be encoded to a *Streaming Mesh* [Isenburg and Lindstrom 2005]. This approach reorders the input mesh data in such a way that it can be processed in a sliding window fashion, using a finite, fixed-size memory buffer. This is not only useful for out-of-core mesh processing algorithms, but it also gives clients a guarantee that indices never refer to vertices that have not been received yet. It therefore enables a simple progressive transmission of mesh data, by appending downloaded data directly to existing buffer content. In a similar spirit, the *POP Buffer* algorithm reorders mesh data with the aim of straightforward progressive transmission [Limper et al. 2013]. The reordering scheme is based on the degeneration of a large amount of triangles when performing aggressive quantization. By rendering triangle data with increasing precision, and sending, for each precision level, only the non-degenerate triangles, a progressive transmission of the whole mesh data is achieved. The second row of Fig.3 shows an example. Nevertheless, the high speed and lack of CPU-based decoding steps comes at the cost of a rather bad rate-distortion performance, compared to progressive mesh methods that explicitly adapt the topology of the mesh (e.g., compared to the approach of Lavoué et al. [Lavoué et al. 2013]).
For streaming meshes, there is currently no Web-based rendering library that applies this method for progressive transmission. The POP buffer method has been experimentally implemented inside the X3DOM library. However, the corresponding X3DOM *POPGeometry* node uses a set of child nodes to represent the different precision levels, and each chunk of triangle data is loaded from a separate file. This obviously leads to an unnecessary large number of HTTP requests, and it furthermore significantly increases the size of the application's HTML page.

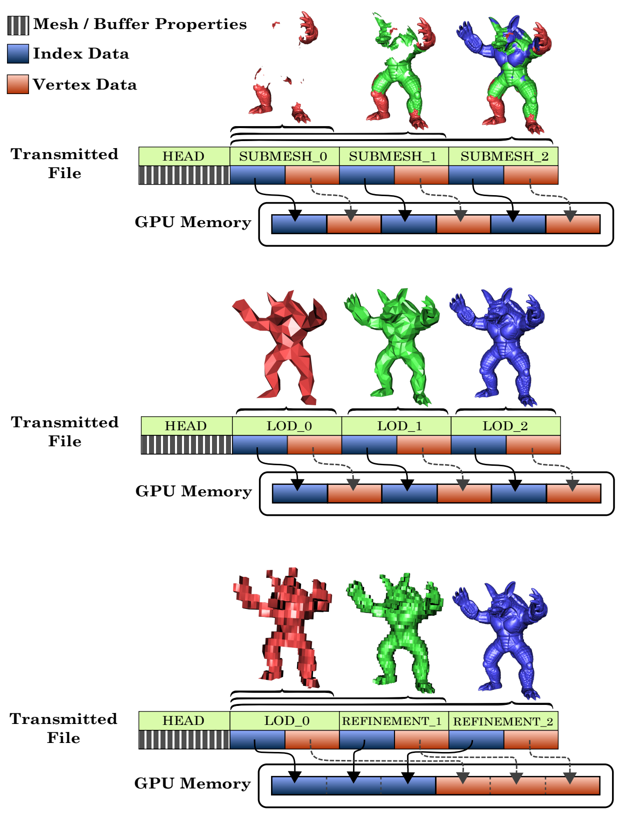


Figure 3: *Transmission and GPU storage of mesh data, for different data subdivision schemes. From top to bottom: Sub-Meshes, Discrete LOD, Progressive LOD. In the third case, all received chunks are progressibely concatenated to larger GPU buffers. This efficient, yet flexible coupling of transmitted data with its GPU representation is not possible with any existing transmission format.*

Texture Compression. Texture compression can drastically reduce the amount of memory textures require, which is especially helpful for transmission. Texture compression support of WebGL allows the direct upload of compressed texture data to the GPU without the need for an additional unpacking or decoding step. The Khronos Group has proposed WebGL extensions to support several texture compression formats. Currently, the most popular extension adds support for the patented S3TC texture compression algorithms [3]. This group of lossy compression formats, labeled DXT1 through DXT5, achieves a fixed compression rate of 6:1 [4]. According to WebGLStats[5], as of now, 77% of the WebGL-enabled browers, that visit their webpage, support this extension.

3 The SRC Format

Within this section, we present the most important features of our proposed SRC format. We first provide some motivation and features of the format itself, and we discuss its basic integration into X3D scenes. After this, we describe how our proposed X3D *ExternalGeometry* node can be used to achieve highly flexible and dynamic compositing of mesh data.

[3]http://www.khronos.org/registry/webgl/extensions/WEBGL_compressed_texture_s3tc/
[4]https://www.opengl.org/wiki/S3_Texture_Compression
[5]http://www.webglstats.com

3.1 Structured Mesh Property Encoding

While glTF does not fulfill all of the requirements for our proposed format, it has successfully served as a base for our thoughts, and we believe that we can easily motivate and explain our contributions by first considering a glTF description. This is especially true as both formats use a kind of structured, hierarchical description of the mesh properties. The great advantage of this hierarchical design is that it maps to GPU structures on the client side in a straightforward manner. It furthermore allows multiple meshes to freely share a random number of accessors to index and vertex data, which is not that easily possible with X3DOM's geometry containers, for example. Since it generally maps very well to rendering structures on the client side, we have decided to adapt some of the basic structures of glTF. However, we have identified several aspects of glTF that made it unuseable for our purpose (see Section 2, and we will show how to overcome this issues within the following.

Furthermore, we did not want to merge structural information about the scene into our 3D asset delivery format (which is a general problem with X3D, but in parts also applies to glTF). Therefore, we did not include, for instance, information about lighting, or about the scene hierarchy, into our SRC header.
Besides that, we have performed some modifications of existing glTF concepts, for example by differentiating between *indexView* and *attributeView* objects. For more information about the details of the basic structure of our format, the interested reader is referred to the appendix of this paper (Section 6).

We have also identified a problem related to the use of quantized position data. In glTF, there are *min* and *max* attributes available for each mesh attribute accessor, specifying extreme values within the corresponding buffer. However, this does not help in the case of converting normalized, quantized data (for example, in a 16 bit integer format) to an original floating-point range. Moreover, if cracks should be avoided, data within all sub-meshes must be quantized with the same bounding box scale [Lee et al. 2010]. During decoding, an additional offset vector is then used to translate the data to the original position. We have therefore decided to introduce two new attributes,*decodeOffset* and *decodeScale*, which are specified for each *attributeView* object. With the values of *decodeOffset* and *decodeScale* being denoted as vectors $\vec{d_o}$ and $\vec{d_s}$, and $\vec{p_q}$ being a quantized position read from a transmitted buffer, the decoding to floating-point position values $\vec{p}$ on the client side is performed as follows:

$$\vec{p}(\vec{p_q}) = \frac{\vec{p_q} + \vec{d_o}}{\vec{d_s}}$$

This process can take place on the CPU side, or on-the-fly during rendering (which is likely to be the most efficient solution in most cases). If *decodeOffset* and *decodeScale* contain values of *[0, 0, 0]* and *[1, 1, 1]*, the transmitted buffer values do not need to be decoded on the GPU, but can directly be used for rendering. For more details about the decoding of quantized positions, to achieve a crack-free composed mesh, the interested reader is referred to the work of Lee et al. [Lee et al. 2010]. In a similar way, the *decodeOffset* and *decodeScale* attributes can be used to decode quantized normals and texture coordinates. Note that the *min* and *max* attributes are not available for *attributeView* objects in our SRC header. Instead, we are specifying bounding box information with *bboxCenter* and *bboxSize* attributes, available for each *mesh*.

Our aim was that texture data can be included in a very similar fashion like mesh attribute data. Therefore, we have not only included a separate *textures* list, but also a list of *textureView* objects that access texture data from the file body. This not only saves us separate requests for each texture file, but it also enables a progressive transmission of texture data, and interleaved transmission with mesh attributes (see Section 3.2 and Section 3.3). Furthermore, we allow that the texture data, which is transmitted using one or more buffer chunks, is encoded in a format that can directly be uploaded to the client's GPU, without any decode time. This can either be a raw format, or an array with compressed texture data in an S3TC format. The resulting basic structure of the classes used for hierarchical mesh description is illustrated by Fig. 4.

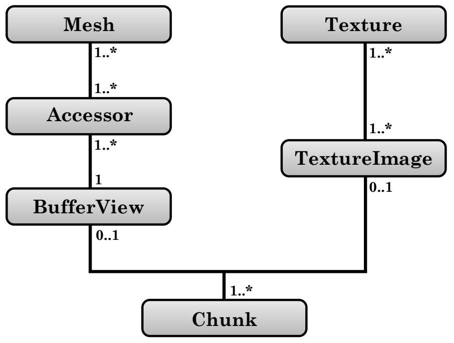


Figure 4: *Basic structure of our proposed SRC format. All structured information is delivered in the SRC header, binary data chunks represent subsequent sections of the file body.*

Another point that is also worth noting is the fact that we are always transmitting the header and body of our SRC format in a single file. This saves us an additional request for the header information, and we believe that this aspect is especially relevant for larger scenes. The header is usually of negligible size, therefore we currently encode it in a standard ASCII JSON format. This has the advantage that the client application can still easily use it, via a standard JavaScript call to *JSON.parse()*. For the future, however, we reserve the possibility to use different header encodings. Therefore, our the first three words of our file are a magic number, identifying our format, an identifier for the header format and its version, and finally the length of the header, given in in bytes.

3.2 Progressive Transmission of Mesh Data Buffers

As can be seen in Fig. 3, progressive transmission methods require that the final mesh data buffers are transmitted in an interleaved fashion. We note that the X3DOM *POPGeometry* node achieves a similar behavior, by maintaining a separate HTTP request for each refinement. This, however, introduces a tight coupling between the transmission layer and the rendering layer, which means that the number of LOD refinements will always determine the number of requests. Since our goal is to minimize the necessary amount of HTTP requests, we have decided to enable the transmission of all LOD refinements in a single file. This in turn requires us to change the base layer of our description hierarchy, introducing the concept of *buffer chunks*. In our SRC definition, a buffer chunk is simply a slice of a particular mesh data buffer (i.e., a slice of a vertex attribute buffer or index buffer).

In the trivial case, each mesh data buffer consists of a single chunk. Generally, we allow the encoding application to arrange the single slices of all mesh data buffers in a random order, wich makes it possible to interleave several buffers during transmission. A client application can, for example, initially receive a first batch of index data along with the attributes of the referenced vertices, and render an intermediate representation as long as the rest of the buffers is being progressively downloaded in the background. In general, our SRC format allows to include all use cases where mesh geometry and connectivity information is transmitted in a progressive manner, like POP Buffers or Streaming Meshes, for example.

3.3 Progressive Transmission of Textures

Since our concept of texture representation within our SRC format is also built on the concept of buffer chunks, as it is already used for mesh geometry, we allow that texture information is transmitted along with mesh geometry data in an interleaved fashion. Although we have not yet tested our format with a truly progressive transmission format for texture data, this could basically be achieved by using existing progressive image transmission methods, like the *Adam7* encoding scheme of the PNG format. We assume that a progressive representation of a texture is always given at resolutions that increase, in each dimension, by a factor of two, with each new texture image. This is mainly due to the usage of the separate texture images in the form of MIP map levels at the GPU. For each texture, we therefore store the length of each image that belongs to the texture within the SRC header, using an attribute entitled *imageByteLengths*. The last number in this list is always the size, in bytes, of the full-resolution texture image, while the others represent the sizes of the respective MIP map levels, starting with a MIP map size of 1×1 pixels. This way, client applications are able to progressively retrieve all MIP map levels, and use them to render intermediate stages during texture data retrieval (cp. Fig. 1)[6].

At this point, the reader might find that a progressive transmission of MIP map levels might be unnecessary for common textures, as a fast generation on the client's GPU is also possible. However, with the possibility to directly use compressed texture data, it becomes necessary to transmit also the pre-computed MIP map levels.

3.4 Integration with X3D

Integrating SRC Content with X3D scenes. We have designed our SRC format in a way that allows Web applications to use it as a self-containing geometry container. This is a huge conceptual difference to X3DOM's *BinaryGeometry* and *POPGeometry nodes*, where all information needed for interpreting the binary data files is encoded directly inside the X3D document (cp. Fig. 5), or as part of the binary data file extension (using extensions like '.bin+8' or '.bin+4'). Although X3D scene authors can decide to transfer geometry declarations to external X3D files, using the *inline* mechanism, this does not solve the basic problem that details of the geometry description, like, for instance, the data type of the vertex data buffers, have been included in the X3D declaration of the respective nodes.

We decided to follow a similar approach as employed by XML3D, in form of the *mesh* tag: basically, all our tag needs is a reference to a self-containing external file. This reference is specified using the *url* field, linking to a file which then contains all relevant geometry description and data.

An important aspect, which deserves special attention, is the possibility to use two fields of the surrounding Shape node, *bboxCenter* and *bboxSize*, which are describing the axis-aligned local bounding box of the corresponding model. The values could, for example, be computed by the authoring tool, which was used to export the X3D scene. The interpretation of those fields should be handled by the client, based on the following rules:

[6]We note at this point that an efficient implementation of this progressive texture rendering method might be depending on MIP level clamps, a feature that is not available with WebGL 1.0. However, it is already part of the Webl 2 specification.

```
<Shape>
  <Appearance>
    <Material diffuseColor='0.6 0.6 0.6'
              shininess='0.00234375'/>
    <ImageTexture url='"duck.png"'/>
  </Appearance>
  <BinaryGeometry DEF='BG_0' solid='false'
           vertexCount='12636'
           position='13.44 86.94 -3.70'
           size='165.47 154.04 115.25'
           primType='"TRIANGLES"'
           index='binGeo/indexBin.bin'
           coord='binGeo/coordBin.bin+8'
           normal='binGeo/normalBin.bin+4'
           texCoord='binGeo/texCoordBin.bin+4'
           coordType='Int16'
           normalType='Int8'
           texCoordType='Uint16'/>
</Shape>
```

X3DOM BinaryGeometry

```
<Shape>
  <Appearance>
    <Material diffuseColor='0.6 0.6 0.6'
              shininess='0.00234375'/>
    <ImageTexture url='"duck.src#tex_1"'/>
  </Appearance>
  <ExternalGeometry url='duck.src'/>
</Shape>
```

ExternalGeometry

Figure 5: *Two X3D encodings of the Collada Duck example file, comparing X3DOM BinaryGeometry (top) and our proposed SRC format and ExternalGeometry node. The BinaryGeometry node leads to a cluttered X3D (or HTML) document, as it contains many fields that are not interesting for the scene author.*

1. The *bboxSize* field should be considered to contain an unspecified size, if at least one of the three dimensions has a negative value (otherwise, the field is considered as specified). Following the X3D specification, the recommended way to communicate an unspecified size is to use a value of *-1 -1 -1* for the *bboxSize* field.

2. If the *bboxSize* field is specified, the *bboxSize* and *bboxCenter* fields are used to determine whether the mesh should be loaded. If the mesh has already been loaded, the *bboxSize* and *bboxCenter* fields can be used for visibility determination (e.g., view frustum culling).

3. If the *bboxSize* field is unspecified, the *bboxSize* and *bboxCenter* fields are completely ignored during scene loading and rendering, following the X3D specification. We allow the additional possibility that the X3D browser, as a fallback, performs a lookup for valid bounding box data in the header of the corresponding external file.

The proposed design allows the client application to decide at which point in time an SRC file, representing 3D information in a specific area of the scene, should be loaded. This is especially helpful to reduce the number of HTTP requests. We note that the second rule also implies that the *bboxSize* and *bboxCenter* fields, if valid, are used for culling, instead of using the internal *bboxCenter* and *bboxSize* fields of the SRC header. If a scene author wants to use the bounding box data from the external file instead, the third rule can be used for this purpose by setting the *bboxSize* field to the unspecified value *-1 -1 -1*.
While this redundancy is, in the context of X3D applications, basically not necessary, it keeps the SRC format self-containing, thereby allowing bounding box information in the SRC header to be used within other rendering frameworks. Please note that using the *size* and *center* fields is just an option - if they are invalid,

boundinx box information is retrieved from the SRC header. This enables the X3D-based Web application to dynamically include any external SRC file, without prior knowledge about its corresponding bounding volume.

Addressing SRC File Content. One of our intentions was that our file format should be useable as a self-containing geometry container, within a wide variety of application scenarios. To further control the number of HTTP requests, which must be issued by the client application to receive all currently relevant mesh data, we have decided to add the possibility to encode multiple meshes within a single file. This in turn demands some form of addressing scheme, that can be used to refer to specific parts of the file content.

For X3D, we propose a simple suffix scheme that enables us to refer to a specific mesh, or texture, within a file. Within our addressing scheme, a hash symbol must be used to separate the file name from the identifier of the respective mesh or texture. This way, X3D *ImageTexture* elements can refer to the texture that belongs to a specific part of geometry, and we can ensure that both are always loaded together. As the *ImageTexture* element accepts various image formats, a combination of a geometry file in SRC format with external textures is still easily possible.

The Source Node. A great advantage of data flow systems, like the XFlow System for declarative 3D content, is that they are able to combine data elements from various sources. In the case of XFlow, a mesh can be loaded from a file and re-used in several places. A custom attribute override can then be used to alter individual properties (like vertex colors, for example) for each instance, and to keep all of the other attributes from the original mesh. To allow such a dynamic behavior with our new container, there are generally two possibilites.

First, variants that exist within a single SRC container can simply be addressed by using the corresponding identifier when referring to the file. A SRC file could, for example, contain a single set of vertex positions and normals, three different sets of vertex colors, and three different meshes that realize the three variants. However, this scheme does not allow to dynamically combine data from various SRC files.

The second possibility to override single mesh properties, which also works when merging data from different SRC files, is to use a special declaration that assigns data from another source to a specific mesh property. To realize this declaration within X3D, we propose a *Source* node. The name is inspired by the *source* tag, known from HTML5 video embedding. However, there are some major differences between our *Source* node and this tag. While the tag lists alternatives for the corresponding video, our node may be used to provide different kinds of mesh attributes or indices, allowing for partial overrides of mesh data. Therefore, we also use an optional field entitled *name*, which identifies the corresponding attribute, if any. Second, our *Source* node uses the more X3D-conformant field name *url*, instead of *src*, to specify the location of the data to be loaded.

A *Source* node must always be the child of an *ExternalGeometry* node, and an *ExternalGeometry* node may have an arbitrary number of *Source* nodes as children. Fig. 6 shows an example, overriding a single *color* attribute of a mesh. The *name* field contains the name of the specific attribute, as it was specified in the original container file (in the example: in file *duck.src*). Note that we have used an extension of the mentioned addressing scheme, using a dot and an attribute name, to refer not only to a particular mesh, but to a particular attribute of that mesh. With our format it is, nevertheless, also possible to refer to an attribute that is encoded as a standalone file,

```
<Shape>
  <Appearance>
    <Material diffuseColor='0.6 0.6 0.6'
              shininess='0.00234375'/>
    <ImageTexture url='"duck.src#tex_1"'/>
  </Appearance>
  <ExternalGeometry bboxCenter='1 3 5' bboxSize='2 3 2
                    url='duck.src'>
    <Source name='color'
            url='duckAltColors.src#mesh_1.color'/>
  </ExternalGeometry>
</Shape>
```

Figure 6: *Overriding mesh properties, using our Source node.*

or to the first attribute within a file that has a matching name.

Furthermore, we allow several *Source* nodes to be nested. This way, we enable scene authors to recursively override several mesh attributes. The *ExternalGeometry* node acts as a top-level variant of the source node, with the difference that is has no *name* field. Nevertheless, its *url* field can be used to include a set of mesh data that is then partially overridden by child *Source* nodes. If multiple *Source* nodes are specified as direct children of a *ExternalGeometry* node, but the *url* of the *ExternalGeometry* is empty, the data from the *Source* nodes is interpreted as separate parts, which are jointly representing the corresponding mesh (see the right part of Fig. 7 for an example).

The following rules are used to determine which attributes from the file, specified via a *Source* node, using the (*url*) field, are used to override the original attribute data (coming from a parent node that is either a an *ExternalGeometry* node, or another *Source* node):

1. If the *name* field is empty (which is the default value), all mesh data from the file is used to override original index data and attribute data with the same identifiers, if any.

2. If the *name* field is not empty, the following rules are applied to replace a particular attribute of the original data, which has a name which matches the value specified in the *name* field:

   (a) If the *url* field contains a reference to a specific attribute of a specific mesh, this attribute is used (regardless of its name) to replace the original attribute.

   (b) If the *url* field contains a reference to a specific mesh (but not to a specific attribute), the first attribute of this mesh, with a name that matches the original attribute's name, is used.

   (c) If the *url* field contains only a reference to a file (but not to a specific mesh or attribute within the file), the first occurence of an attribute, with a name that matches the *Source* node's *name* field, is used.

This way, it also becomes simple to override attribute data with new data from a single file, which we believe is interesting especially for application scenarios like simulations and scientific visualization.

4 Experimental Results

We have included a first implementation of the *ExternalGeometry* and SRC exporter into InstantReality's AOPT tool[7]. A very basic version of the loading and rendering code has been included in the X3DOM framework. Furthermore, we have set up a project page, which will enable the interested reader to track progress of our im-

[7]http://www.instantreality.org

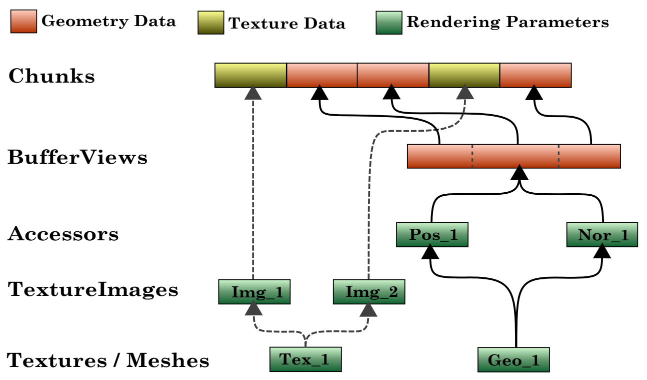


Figure 8: *Structure of the body of an example SRC file, containing the data shown in Fig. 1.*

plementation, including both, exporter and renderer[8]. Our current implementation of the SRC encoder and renderer is still in an experimental state. Nevertheless, after considering a wide variety of use cases, we are confident that our format is are able to cover the needs of many different 3D Web applications. Within this section, we illustrate some of them by example.

Sharing SRC Data. Since we want to have full control over the number of HTTP requests that is introduced by our 3D Web application, we have designed our format in such a way, that one SRC file may contain an arbitrary number of assets. The single meshes and textures, and even single mesh attributes, can be referenced from external documents (for instance, from an X3D scene) by using our proposed addressing scheme. Fig. 7 shows two example scenes, illustrating different use cases for accessing a shared SRC file.

Progressive Transmission. We also found that a great property of our SRC format is the fact that encoding applications can specify an exact, progressive order for the download of a batch of mesh data buffers, belonging to a mesh, by arranging the chunks in a corresponding order inside the file body. Fig. 1 shows some intermediate stages of streaming a scanned elephant sculpture model. The structure of the corresponding SRC file body is shown in Fig. 8. As can be seen, texture data and geometry data (in this case, non-indexed triangles) is transmitted in an interleaved fashion. Low-resolution texture data is transmitted first, and used throughout the first stages of geometry refinement. As soon as the geometry is available at a reasonable quality, the difference between the low-resolution texture and the high-resolution variant becomes visible. At this point, additional texture data is streamed, before a final geometry refinement take place.
Since the chunks are still plain binary containers, intended for direct GPU upload (in WebGL, for instance, using the `bufferSubData` function), the performance of the 3D Web application is not touched by introducing the concept of buffer chunks. Please note that we are able to pre-allocate each buffer, as soon as we have received the SRC header, and that a separate allocation of GPU memory for each incoming chunk is therefore not necessary.

Texture Compression. A novelty of our proposed format, compared to existing approaches in X3DOM, for instance, is the possibility to use compressed textures. This does not only result in a significantly reduced GPU memory consumption, but also has several advantages in a 3D Web context. Fig. 9, for instance, shows a rendering of the Siena cathedral. The whole scene uses 79 different textures, with a total size of 241 MB in PNG format. Compressed to DXT1 format, the total size of all texture files shrinks to 78 MB, with just a minimal notable difference. Furthermore, the startup time of the application can this way be significantly reduced.
We have noted that this approach is very similar to existing approaches for mesh geometry: by allowing a direct GPU upload of downloaded data, the startup time of the application is significantly reduced (cp. [Behr et al. 2012], for example).

[8]http://www.x3dom.org/src

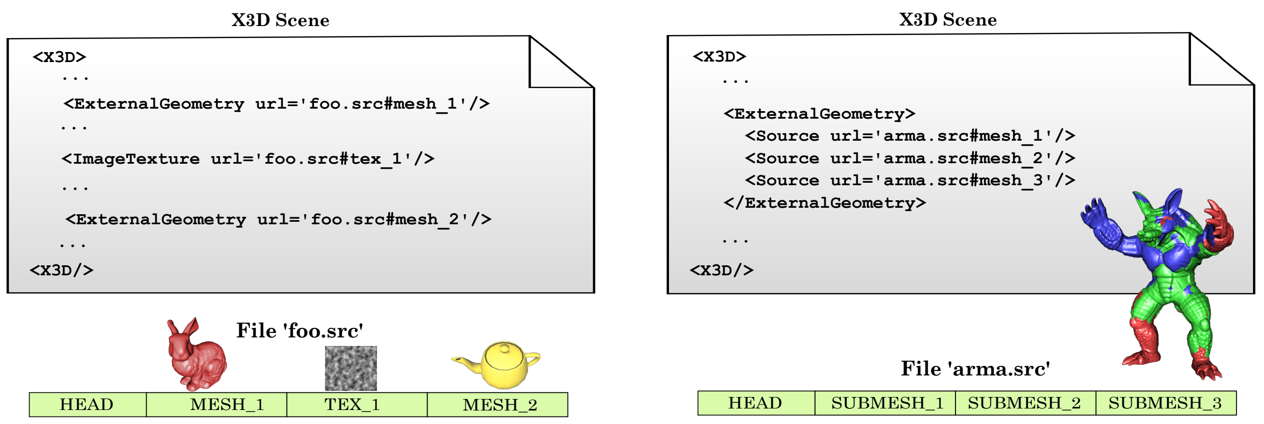
Figure 9: *Siena cathedral, rendered in a Web browser. In such cases, using texture compression is the most efficient way to reduce GPU memory consumption, download time and decode time.*

Externalizing Shapes Nodes. We have found that, instead of just externalizing data from *Geometry* nodes, it can become useful to externalize all data within a *Shape* node. This is especially important for large scenes, in order to reduce the overall size of the Web application's HTML page. Therefore, we have introduced an additional X3D node, entitled *ExternalShape*. As the name implies, this special kind of *Shape* node, that has no children, enables us to include a mesh, or all meshes from an SRC file, via its *url* field. To represent appearance information inside an SRC file, we allow the *mesh* objects inside the header to refer to a *material* object via its ID, just like it is done in glTF. Such a material description can then, for instance, include an X3D-compatible material description, as well as references to textures. With this powerful concept, we can even use *ExternalShape* nodes in a similar fashion like *Inline* nodes, to include multiple geometries with different materials, representing a single, large 3D object.

5 Conclusion

Within this paper, we have presented a novel, streamable format for transmission of 3D mesh data, entitled *Shape Resource Container* (*SRC*). Our format is built on latest technological developments, like the ability to stream binary data with XHR, and the use of compressed textures in upcoming WebGL 2.0 applications. We allow the authors of high-performance 3D Web applications to minimize the number of HTTP requests, by progressively transmitting an arbitrary number of mesh data chunks within a single SRC file. Furthermore, an interleaved transmission of texture data and mesh geometry is possible, allowing for full control over the order of progressive 3D asset transmission.

In order to use our format within declarative 3D scenes, we have discussed an integration of SRC content into X3D. Our proposed *ExternalGeometry* node can be used to include a random number of 3D mesh geometry instances into a single *Shape* node. Furthermore, content from a single SRC can be distributed to a random number of *Shape* nodes. Using a set of dedicated *Source* nodes as children of an ExternalGeometry node, potentially in a nested fashion, we can furthermore realize a wide range of mesh data composi-



```
<X3D>
  ...
   <ExternalGeometry url='foo.src#mesh_1'/>
  ...
   <ImageTexture url='foo.src#tex_1'/>
   ...
   <ExternalGeometry url='foo.src#mesh_2'/>
  ...
<X3D/>
```

```
<X3D>
   ...

   <ExternalGeometry>
     <Source url='arma.src#mesh_1'/>
     <Source url='arma.src#mesh_2'/>
     <Source url='arma.src#mesh_3'/>
   </ExternalGeometry>

   ...

<X3D/>
```

Figure 7: *Accessing different content within a SRC file, from various elements within an X3D scene. Our proposed format allows a random mapping between the number of files (and thereby the number of HTTP requests) and the number of identifiable assets within the scene.*

tion schemes. This allows for dynamic updates of single attributes, and it further enables scene authors to maximize full or partial reuse of mesh data among several instances.

Future work includes the investigation of different header encodings, for example, using *Binary JSON*[9], or Google's *Protocol Buffers* library[10]. Furthermore, we would like to continue our work on the concept of *ExternalShape* nodes. This includes the question, if our concept for mesh data compositing (using *Source* nodes) could be transferred to this context. Finally, an important topic for future research is the integration of parametric geometry descriptions [Berndt et al. 2005].

References

ARNAUD, R., AND BARNES, M. C. 2006. *Collada: Sailing the Gulf of 3D Digital Content Creation*. AK Peters Ltd.

BEHR, J., ESCHLER, P., JUNG, Y., AND ZÖLLNER, M. 2009. X3DOM: a DOM-based HTML5/X3D integration model. In *Proc. Web3D*, 127–135.

BEHR, J., JUNG, Y., FRANKE, T., AND STURM, T. 2012. Using images and explicit binary container for efficient and incremental delivery of declarative 3D scenes on the web. In *Proc. Web3D*, 17–25.

BERNDT, R., FELLNER, D. W., AND HAVEMANN, S. 2005. Generative 3d models: A key to more information within less bandwidth at higher quality. In *Proc. Web3D*, ACM, New York, NY, USA, Web3D '05, 111–121.

CHUN, W. 2012. WebGL models: End-to-end. In *OpenGL Insights*, P. Cozzi and C. Riccio, Eds. CRC Press, July, 431–454.

FOGEL, E., COHEN-OR, D., IRONI, R., AND ZVI, T. 2001. A web architecture for progressive delivery of 3D content. In *Proc. Web3D*, 35–41.

HOPPE, H. 1996. Progressive meshes. In *Proc. SIGGRAPH*, 99–108.

ISENBURG, M., AND LINDSTROM, P. 2005. Streaming meshes. In *Proc. VIS*, 231–238.

[9]http://bsonspec.org/

[10]https://code.google.com/p/protobuf/

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 *Proc. Web3D*, ACM, New York, NY, USA, Web3D '13, 39–45.

KLEIN, F., SONS, K., JOHN, S., RUBINSTEIN, D., SLUSALLEK, P., AND BYELOZYOROV, S. 2012. Xflow: Declarative data processing for the web. In *Proc. Web3D*, ACM, New York, NY, USA, Web3D '12, 37–45.

LAVOUÉ, G., CHEVALIER, L., AND DUPONT, F. 2013. Streaming compressed 3d data on the web using javascript and webgl. In *Proc. Web3D*, ACM, New York, NY, USA, Web3D '13, 19–27.

LEE, J., CHOE, S., AND LEE, S. 2010. Mesh geometry compression for mobile graphics. In *Proc. CCNC*, 301–305.

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

MAGLO, A., LEE, H., LAVOUÉ, G., MOUTON, C., HUDELOT, C., AND DUPONT, F. 2010. Remote scientific visualization of progressive 3D meshes with X3D. In *Proc. Web3D*, 109–116.

PENG, J., KIM, C.-S., AND JAY KUO, C. C. 2005. Technologies for 3D mesh compression: A survey. *J. Vis. Comun. Image Represent.*, 688–733.

SONS, K., KLEIN, F., RUBINSTEIN, D., BYELOZYOROV, S., AND SLUSALLEK, P. 2010. XML3D: interactive 3D graphics for the web. In *Proc. Web3D*, 175–184.

6 Appendix

Figure 10 shows an example of a JSON-based encoding of our SRC header. The file contains a single mesh with a single texture, where the geometrical data and the texture data is interleaved and transmitted progressively, as shown in Fig. 1.
Please note that we have used additional *meta* objects to specify application-dependent meta data. One global meta object contains general meta data about the file content, such as a short textual description. Other meta objects are directly attached to the mesh and texture objects. As we have used the POP Buffer method to progressively transmit the triangle data, it was necessary to specify also the progression levels (in vertices), which are associated with the mesh, via its meta object.

```
{
   "meta":{
      "description":"This is a simple example with an elephant"
   },
   "bufferChunks":{
      "chunk0":{
         "byteOffset":0,
         "byteLength":23448
      },
      "chunk1":{
         "byteOffset":23448,
         "byteLength":1849344
      },
      "chunk2":{
         "byteOffset":1872792,
         "byteLength":7669440
      },
      "chunk3":{
         "byteOffset":9542232,
         "byteLength":4767845
      },
      "chunk4":{
         "byteOffset":14310077,
         "byteLength":1551744
      }
   },
   "bufferViews":{
      "attributeBufferView0":{
         "byteLength":11070528,
         "chunks":[
            "chunk1",
            "chunk2",
            "chunk4"
         ]
      }
   },
   "textureViews":{
      "elephantTexView":{
         "byteLength":4791293,
         "chunks":[
            "chunk0",
            "chunk3"
         ],
         "format":"png"
      }
   },
   "accessors":{
      "indexViews":{},
      "attributeViews":{
         "attributeView0":{
            "bufferView":"attributeBufferView0",
            "byteOffset":0,
            "byteStride":16,
            "componentType":5123,
            "type":"VEC3",
            "count":2399,
            "decodeOffset":[
                28003.4827119521,
                -29173.7907980278,
                31671.6816747218],
            "decodeScale":[
                535.4424912549,
                271.0610593755,
                293.4133239205]
         },
         "attributeView1":{
            "bufferView":"attributeBufferView0",
            "byteOffset":8,
            "byteStride":16,
            "componentType":5121,
            "type":"VEC3",
            "count":2399,
            "decodeOffset":[-128, -128, -128],
            "decodeScale":[128, 128, 128]
         },
         "attributeView2":{
            "bufferView":"attributeBufferView0",
            "byteOffset":12,
            "byteStride":16,
            "componentType":5123,
            "type":"VEC2",
            "count":2399,
            "decodeOffset":[0, 0],
            "decodeScale":[656535, 65535]
         }
      }
   },
   "meshes":{
      "elephant":{
         "attributes":{
            "position":"attributeView0",
            "normal":"attributeView1",
            "texcoord":"attributeView2"
         },
         "indices":"",
         "material":"",
         "primitive":4,
         "bboxCenter": [51.69, -108.82, 106.83],
         "bboxSize":   [121.18, 239.37, 221.14],
         "meta":{
            "progressionMethod":"POP",
            "indexProgression":[],
            "attributeProgression":[
               138, 594, 2262, 8478, 32238, 115584,
               337764, 594924, 684972, 691878, 691908
            ]
         }
      }
   },
   "textures":{
      "elephanttex":{
         "textureView":"elephantTexView",
         "imageByteLengths":[
            23448,
            4767845
         ],
         "width":512,
         "height":512,
         "internalFormat":6408,
         "border":0,
         "type":5121,
         "format":6408,
         "meta":{}
      }
   }
}
```

Figure 10: *Example, using a JSON-encoded SRC header of a scene with a single model, shown in Fig. 1. Chunks of vertex attributes are transmitted interleaved with two texture images.*