Using Images and Explicit Binary Container for Efficient and Incremental Delivery of Declarative 3D Scenes on the Web

Johannes Behr* Yvonne Jung† Tobias Franke‡ Timo Sturm§

Fraunhofer IGD, Darmstadt, Germany

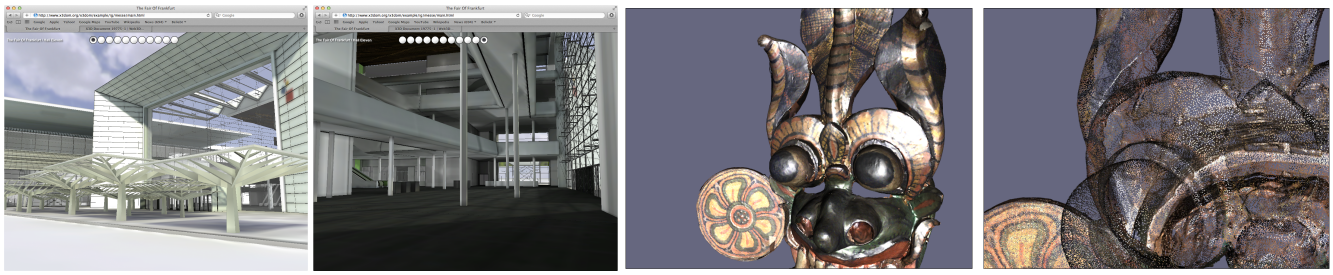

Figure 1: *Two different high-resolution models visualized with X3DOM in the Web Browser. Left: architectural walk-through model of Hall 11 of the Fair of Frankfurt – represented and rendered using our sequential image geometry (SIG) approach for fast content delivery and data compression on the Web. Right: 3D-scanned historical object rendered as triangle mesh and as point set using our SIG approach.*

Abstract

JSON, XML-based 3D formats (e.g. X3D or Collada) and *Declarative 3D* approaches share some benefits but also one major drawback: all encoding schemes store the scene-graph and vertex data in the same file structure; unstructured raw mesh data is found within descriptive elements of the scene. Web Browsers therefore have to download all elements (including every single coordinate) before being able to further process the structure of the document. Therefore, we separate the structured scene information and unstructured vertex data to increase the user experience and overall performance of the system by introducing two new referenced containers, which encode external mesh data as so-called *Sequential Image Geometry* (SIG) or Typed-Array-based *Binary Geometry* (BG). We also discuss compression, rendering and application results and introduce a novel data layout for image geometry data that supports incremental updates, arbitrary input meshes and GPU decoding.

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

Keywords: X3D, Declarative 3D, HTML5, WebGL, DOM, Web Integration, Image Geometry, Typed Array, Mesh Compression

1 Introduction

The expected user experience of web pages and web applications is quite different to traditional desktop or mobile applications: there is no explicit and visible application startup or data preparation and processing step. Everything starts with a single click, which leads to an instantaneous user experience, where inconsistent or partial presentations are accepted when loading the page (e.g. images are missing, layout is not yet final). This asks for very compact documents and data chunks that can be interpreted and used for the visualization before all data is fully delivered to the client.

With X3DOM, a DOM-based integration model for declarative (X)3D in HTML5 was proposed [Behr et al. 2011; Behr et al. 2010] which allows a seamless integration of 3D content into the HTML document model by utilizing standard Web APIs. The approach is one of the prototype systems used for the evaluation of use cases and requirements in the declarative 3D community group [W3C (Community Group) 2012]. However, for real world applications with large 3D data sets such declarative approaches, which integrate the 3D data into the HTML DOM, soon lead to huge HTML documents (see Figure 2). This in turn causes unpleasant, non-interactive user experiences due to long loading times and non-responsive web pages, because Web Browsers are not build to parse DOM attribute data sets beyond several megabytes in size.

Providing the data not just in one document but in multiple data-sets using the X3D compliant *Inline* mechanism [Web3D Consortium 2011] is often no suitable solution. First, an `<Inline>` element externalizes a complete subtree instead of the raw vertex attribute data, which requires the implementation of an additional parser architecture. Second, X3D *Inline* nodes by design hide their content from the application developer via a black-box interface, which contradicts to the requirement that the whole DOM tree needs to be dynamically manipulatable. Third, inlined files are like HTML typically ASCII-encoded to be human-readable and therefore the file size itself is an issue, even when editing the file. Since this vast amount of mesh data usually is not manipulated via DOM scripting but preprocessed in a modeling tool, binary compressed XML/X3D files that reduce the data transmission issue can also be an option here. However, though binary XML compression formats are streamable in general, they are not useful for progressive approaches because of the single document and parser model. Finally, to make the tree structure accessible to the application developer, all data (incl. the raw mesh data) will be provided as part of the DOM and therefore increases the document structure similarly.

*e-mail:johannes.behr@igd.fraunhofer.de
†e-mail:yvonne.jung@igd.fraunhofer.de
‡e-mail:tobias.franke@igd.fraunhofer.de
§e-mail:timo.sturm@igd.fraunhofer.de

Hence, we propose another solution: we do not split the scene-graph into sub-graphs, but divide the lightweight, structured information from the heavy, unstructured data (not only images but also vertex attributes). The light structured information such as transformation groups and materials (that usually makes up less than five percent of the file size) are put into the HTML / XML document and get processed and delivered as usual. The heavy unstructured data (that usually makes up more than 95 percent of a file) is stored very efficiently in binary containers which are transmitted on request. This is very similar to how the web already works today: lightweight HTML pages hold the DOM structure and script code and the heavy data containers, like images, videos, and sound, are requested afterwards on demand.

Key for this concept is a form of binary container, which is very efficient to transport and decode on the Web and useful even for partial or progressive transmission. Our main *contribution* is a new mechanism to store, retrieve and decode binary data of mesh geometry, separated away from the lightweight declarative 3D structure provided within the web document. We explore two different options for binary formats and show that using images with a very tailored data layout and compression scheme as mesh container leads to impressive results. We get close to 6 bytes per triangle for regular meshes (in case only positions and normals are given), which we decode on the GPU while rendering. We also show that dynamic updates via MPNG streams or videos can be used to efficiently update the vertex data. In addition, we compare this approach to standard binary data buffers, how we integrated both approaches into X3DOM and how both representations affect additional requirements such as compression, streaming and animation. We also provide a short discussion on browser behavior concerning time to load data for both encodings. We then conclude the paper with a discussion of results and applications.

The *benefit* of this work is twofold: first, structuring of 3D documents on the web is improved dramatically due to the clear externalization of unstructured data away from the scene-graph and (X)3D application. This separation is beneficial for human-readability and browsers, since parsers need not wade through heavy chunks of binary data within XML documents, also improving the loading time. Clearly, this affects the overall user experience: instantaneous loading and displaying 3D content is now supported just like in regular web documents. Second, with our new geometry container format we enable prioritized progressive transmission; large geometries can be streamed to the user instantaneously in a specific ordering with little effort. Therefore, we exploit existing browser technology, so that a 3D application appears as natural as possible to web engines parsing and loading binary and DOM data.

2 Related Work

There exist various possibilities to reduce the overall size of a 3D scene-graph in general and of mesh data in particular that are independent from a special encoding scheme. On the one hand, the amount of vertex data can be decreased by methods such as stripping [Behr and Alexa 2002], which also reduces the GPU load. On the other hand, compression algorithms can be utilized. Important examples for binary XML compression formats are x3db (a binary encoding for X3D [Web3D Consortium 2011] that is based on FastInfoSet [Sun 2004]), the W3C format EXI [W3C 2011], and the ISO standard x3z. All of them require decoding at the level of JavaScript, though for most of these encodings except for the latter no JavaScript library for decompression is available. Likewise, BIFS, which is part of MPEG-4, also allows compressing the 3D scene-graph. To overcome several issues like better mesh and animation data compression and transmission a new architectural model was proposed by [Jovanova et al. 2009].

Figure 2: *DOM holds structure and data of declarative 3d content. But, apart from very simple examples like this little mesh, usually more than 95% of the file size are unstructured data.*

In [Stocker and Schickel 2011] an interesting study of encoding sizes using the x3db binary encoding and GZIP / BZIP2 compression algorithms in X3D is presented, which shows that with X3D binary encoding followed by GZIP compression, bandwidth requirement of content delivery can be reduced by a factor of two. It was also shown that a scene-graph given as x3db loads around three times faster than ASCII-encoded X3D files and is also the fastest to parse. With the introduction of WebGL [Marrin 2011] the need for binary interfaces in JavaScript did arise as well. Therefore, the Typed Array specification [Khronos 2012] provides an API for the interoperability with native binary data by defining a generic buffer type, the `ArrayBuffer`, and typed array view types (e.g. `Float32Array`) that represent a certain view of it to allow indexing and manipulation of the data stored within the buffer. For binary download, an XMLHttpRequest[1] (XHR) can be used, which since 2011 directly supports `ArrayBuffers` as response type.

In real-time rendering, 2D images are usually used to store textures that are applied and interpolated during the shading process to flat surfaces (e.g. triangles) in 3D space. Normal maps and displacement maps give the illusion of more structured details, which are only useful for the shading process (cf. e.g. [Donnelly 2005]), but not for collision or picking. In [Schwartz et al. 2011], a WebGL-based framework for representing reflectance information via BTFs is proposed, which allows for the progressive transmission and interactive rendering of digitized artifacts by employing a progressive streaming approach for huge BTF data sets.

Analogously, we follow with a similar approach for lightweight geometry compression and transmission via sequential image geometries that likewise utilizes image compression techniques. Storing 3D meshes in image containers has been the target of extensive research, with investigation of regular meshes stored as height maps, as well as the remeshing of irregular triangle meshes into a 2D domain. 2D height maps stored as images usually provide so called 2.5D coordinates per pixel by defining a regular 2D grid with 1D height data. Therefore, they are especially useful for terrain rendering applications. Because of their regular structure, height maps can easily be resampled to gain level of detail capabilities.

Online streaming of large regular datasets with *geometry clipmaps* has been presented in [Losasso and Hoppe 2004; Asirvatham and Hoppe 2005], where multiple nested detail levels contain a view on the currently visible geometry centered around the viewer. The level of detail (LOD) decreases from the center to the edges. This approach has been largely used for terrain rendering applications and streaming large texture sets to the GPU (similar to Megatexturing). However, 3D surfaces are usually described by irregular

[1] http://dvcs.w3.org/hg/xhr/raw-file/tip/Overview.html

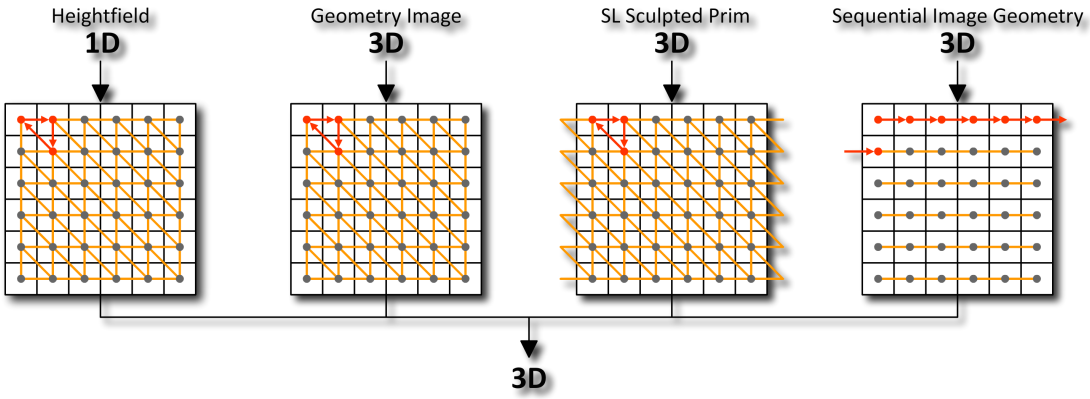

Figure 3: *Implicit mesh topology and drawing order in comparison: height maps (left) and the original geometry image method [Gu et al. 2002] use regular grids to sample the 3D data from the image; sculpted prims [SecondLife 2009] use textures, whose color values are interpreted as radial offsets. Our approach (right) uses images as binary container without a direct correlation to the mesh structure.*

triangle meshes, and since we want to transform arbitrary meshes into binary containers such as images, we therefore do not further investigate regular datasets within the course of this paper.

Remeshing of irregular geometry involves cutting it into charts and mapping these piecewise to a planar domain, which is then sampled by a (semi-)regular grid. Hoppe et al. [Gu et al. 2002] introduce the *geometry image*, a 2D array of size $N \times M$ that stores the xyz position information as rgb values. Geometry images aim to store an entire surface in a single image with an accompanying normal map without waste-space. Instead of cutting the geometry into several disjoint chart-like pieces, the proposed algorithm reparameterizes the mesh after every cut in an iteration until no further improvement can be made in order to create one seamless square chart or atlas. A number of other methods exist to resample geometry into images with arbitrary genus or with genus 0 [Praun and Hoppe 2003]. A rather simple approach with similar properties as geometry images are the Sculpted Prims in SecondLife [SecondLife 2009], which are textures, whose pixel color values are interpreted as radial xyz offset to a sphere, where values less than 127 are negative offsets and values greater than 127 are positive (see Figure 3).

Resampling arbitrary geometry to regular structures such as images has several benefits, which include improved compression and multiresolution geometries using mipmapping via up-/ downsampling operations. Remapping is not effective for all geometries though and may introduce geometric stretch. Undersampling of high-frequency surface features can occur if the geometry image is not large enough [Gu et al. 2002]. Additionally, geometry images can be created from pre-cut patches into a regularly packed format [Purnomo et al. 2004] (a regular structure of sub-atlases). To combat issues like the support for others than only genus-zero surfaces as well as geometric distortion and stretch, in [Sander et al. 2003] irregular multi-chart geometry images are proposed. After partitioning the model into charts, pieces remain as disjoint charts and some waste pixels remain. To improve border-handling by preventing cracks during reassembly of the geometry, a "zippering-algorithm" is introduced, which identifies and unifies boundary samples of each chart with its neighbors.

LOD can be implemented for geometry images. Ji et al. [Ji et al. 2005] create a seamless texture atlas with a quadtree structure. In a two-pass rendering process, an LOD is first selected in a fragment shader and the resulting buffer is subsequently used in a second pass responsible for culling away unnecessary data. GPU-based LOD for geometry images has been proposed in [Hernández and Rudomin 2006]. Through mipmapping a series of LODs is readily accessible at runtime, which are selected based on the distance of the object to the camera. To further improve the results, a special selector-map contains precalculated mipmap levels for selection. In [Maglo et al. 2010] a framework for streaming large 3D datasets in scientific visualization is presented, including a method for progressive mesh compression with LOD management in X3D.

3 Concept and Design

The overall goal is to develop a compression technique which works well with the declarative 3D approach, allows for progressive transmission and decoding to improve the user experience and works well with today's client and server standards. Increasing the interactivity is the major goal. Handling larger objects is also desirable but not necessary for all relevant use cases.

3.1 Separating Structured and Unstructured Data

Embedding the 3D scene data into the DOM and therefore into the HTML document is the fundamental idea of the declarative 3D approach. This kind of data is in all real-world scenarios significantly larger than the information usually stored in HTML documents. Objects with millions of points and normals are very common and therefore soon lead to documents in the gigabyte range. This is not only problematic for today's web browsers but really destroys the user experience since the user has to wait without any feedback from the browser until the full page is loaded.

The interesting quality of the scene file is now that it is not uniformly structured. For most cases the scene structure, which holds all the elements like cameras, lights, groups and geometry objects, including their simple attributes hold up to 5 percent of the actual document and in-memory amount. This is the scene which usually holds all the necessary structural information to be used during runtime. On average, 95 percent of the document are used to store vertex and vertex-related data sets. The positions, normals, texCoords, etc. are unstructured data blobs, which are stored as long lists of float values in the document and memory (see Figure 2), and which typically are never touched by the web developer.

3.2 Reference Binary Container for Unstructured Data

As mentioned, in most applications the vertex attributes are not directly manipulated during runtime. They are used to be transformed and rendered, but only in a very few cases it is necessary to change

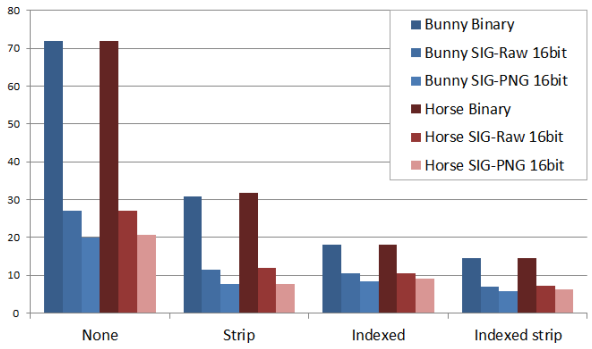

Figure 4: *Bytes per triangle stored for the Stanford Bunny (blue) and the Georgia Tech Horse model (red): sequential image geometries with PNG compression can reduce the file sizes substantially.*

a specific vertex attribute during runtime. Therefore, our approach is now to separate this static unstructured data and move it into external binary container, which can be downloaded during runtime on request. This is very similar to how HTML works already. We have the usual HTML document that includes the document structure and text, whereas the heavy elements like images and videos get referenced and downloaded when needed.

Therefore we need an external format which can hold meshes or vertex attribute data. The final format stores only vertex information since it should be really flexible to be useful for different applications, and the goal is also to map those files to GPU structures directly. Hence, the corresponding HTML element for specifying the mesh should be able to reference external binary data elements.

4 Mesh Data Encoding

To handle binary resources on the Web we basically have two possibilities: on the one hand, we can think of a generic asset dictionary. Here, the raw data is directly loaded to TypedArrays while the data assignment is done in JavaScript. This of course requires having several arrays per external data file, and thus multiple files per scene. On the other, we can use images and videos. The `int` and `float` arrays (e.g. coordinates, normals, and generic attributes) are encoded in RGBA images. This leads to around one to two images per array, and thus multiple images per scene.

We use images and ArrayBuffer dumps as binary container, as these are the options we have right now to load and decode vertex data to the GPU without too much CPU interference in the JavaScript layer. Therefore, two encodings are provided: *binary geometry* (BG) and *sequential image geometry* (SIG). The following sections describe how the information is stored and what the different properties of both methods are.

4.1 Images and Videos

First, we describe how we use `<img>` and `<video>` HTML elements as generic binary container and how they serve as a powerful abstraction for efficient data encoding for Web apps. In this regard, images are related to potentially static (e.g. image) and dynamic (e.g. video) regular grids of pixels.

4.1.1 Sequential Image Layout and Mesh Topology

All image-based 3D rendering methods, which are related to this work, produce 3D primitives (e.g. triangles) from 2D image data. This includes height fields, Geometry Images [Gu et al. 2002], and the so-called Sculpted Prims [SecondLife 2009]. They all use a specific implicit mesh topology to derive the 1D or 3D data from the images and to create those primitives (cp. Figure 3). All these cases are based on regular grids which can resample the image data to a lower or even higher resolution. This is a great application for images, but a really hard to fulfill requirement for meshes. Input meshes have to be pre-processed and must be converted to regular grids, which unfortunately cannot be done for every type of base mesh [Sander et al. 2003].

We propose another approach which does not transform and resample the original mesh to a regular structure, but uses the image to store unlinked vertex data (e.g. coordinates and normals). The pixel neighborhood on the images does not correlate to the neighborhood on the mesh topology. Therefore, up- or down-sampling is not supported with this image type. However, this structure does not have the limitations of the previously described techniques and is very well suited for patching, compression, transmission, and rendering with no special requirements on the input mesh. All vertex-data is stored in a simple fixed sequential order, which is the reason for the name of the layout: Sequential Image Geometry (SIG).

The content can be fully described by HTML (or HTML with our proposed X3DOM extensions) together with image and/or video resource data. One of the most important aspects is that utilizing images as data container allows decompression for free, although only lossless compression like PNG is useful right now. Even more, we also get streaming updates for free since WebGL and X3DOM both directly support the HTML `<video>` tag as texture source. Furthermore, web servers and browsers are well optimized to handle a large number of images and parallel downloads of images, which in turn guarantees a great user experience.

4.1.2 Vertex Data Layout and Encoding

Most previous research on Geometry Images (e.g. [Gu et al. 2002]) just state that they encode xyz coordinates in rgb values without a concrete encoding scheme. With the goal to provide a solution which works well with todays web standards, we have to deal with the limitations of 8 bit images. The HTML standard does not define a specific image or video encoding, but the image formats currently supported by all common browsers are GIF, JPEG, and PNG [Hickson 2012] – all of them with 8 bit per color channel. With RGBA textures we therefore get 32 bits per texel. This is not precise enough for encoding normals, texCoords, and especially positions in one image each. Therefore we propose a simple put progressive encoding to complete the SIG layout.

First, coordinates and texture coordinates are linearly normalized to their corresponding bounding boxes, which results in harmonized coordinate components in the range of $[0, 1]$. The bounding boxes are stored in the HTML / X3D document as part of the structured information and can be used (e.g. for culling) before the vertex data is actually loaded. 2D texture coordinates and normals are usually uncritical since they can be stored quite efficiently in 32 bit, because one RGBA texture can hold 16 bit uv coordinates or alternatively 16 bit θ and ϕ angles for normals in spherical coordinates.

The normalized xyz coordinates however need much more than 8 bits to be useful. Storing x, y, and z in separate textures would give us up to 32 bits for every axis but would increase drastically the amount of textures needed before we can render anything and would not support any form of progressiveness. Therefore we decided to map xyz to RGB but store the different bytes needed in separate textures. So, the first coordinate texture holds the first byte of the coordinate, the second texture the next one and so on. This allows us to render already a raw and not yet fully correct shape with the first texture (cp. Figure 5, left) and supports any precision since

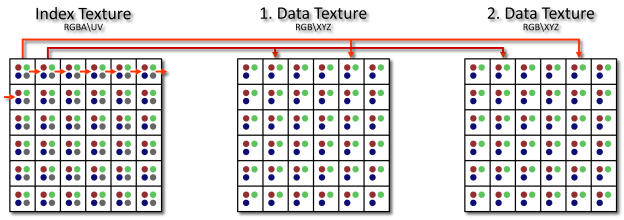

Figure 6: *Using an index texture to access the first and second byte of the corresponding vertex data textures (here the coordinates).*

we are not limited to 32 bit. The actual 3D vertex positions are normalized and quantified to $2^{n \cdot 8}$ values, where $n > 0$ can be error-/user-controlled and would be chosen as part of the data preparation process. Generally, multiple images can be used to distribute precision. This encoding inherently supports LOD and streaming of precision (e.g., closer objects get higher precision). However, for most cases, 16 bit are close to the original data for visualization purposes (see Figure 5) and therefore allow to drastically reduce the amount of memory needed for storage (compare Figure 4).

An additional (but optional) index image is supported to create a 2D single index access mechanism. This map reduces the amount of data to transmit even further (see Figure 6) and can be 8 bit uv (i.e., 16 bit indices) or 16 bit uv (i.e., 32 bit indices), and therefore supports up to 65536^2 vertices instead of 256^2. This comes along with some cache limitations while rendering, as will be described in section 5.5. Nevertheless, this encoding is simple, fast and works with any type of mesh, while the content is fully described by HTML5 including image or video resource data.

4.2 Binary Geometry Data Handling

With the Typed Array specification [Khronos 2012], using binary data is straightforward in JavaScript, because the whole attribute encoding is simply achieved by dumping the vertex data containers to binary files. For binary download, an XHR with *responseType* set to "arraybuffer" is sent to open the binary file. On load, the XHR *response* contains the ArrayBuffer object from which a typed array view (mostly a Uint16Array or a Float32Array) needs to be created, which in turn can directly be used to set the buffer data of the currently bound VBO as shown below. This data only resides in GPU memory and can directly be used for rendering.

```
gl.bufferData(gl.ARRAY_BUFFER,
          new Float32Array(xhr.response), gl.STATIC_DRAW);
```

4.3 Graphics Primitives and Optimization

For both encodings, Binary Geometry (BG) and Sequential Image Geometry (SIG), we support various graphics primitives per mesh, including points, lines, triangles and triangle strips. Every mesh can use any numbers of primitives with corresponding vertex count or length to represent a single draw call. We usually use a simple but fast striping method [Behr and Alexa 2002] to create triangle strips and stitch the resulting strips (by introducing degenerate triangles for concatenating strips) in order to increase the number of triangles rendered in the one call and to decrease the byte-per-triangle count, with or without additional index map. This method allows us to draw every mesh with two draw calls – one for the stitched triangle strips and one for the remaining triangles. This simple but efficient method allows reducing the amount of bytes needed to store triangles even further. Table 1 provides a comparison of relevant encodings, where for generating the x3db encoding the Instant Reality [FhG 2012] transcoder tool was used.

(a)

	X3D	X3DB	BG	SIG raw	SIG PNG
raw tris	11,135,415	2,298,303	5,000,472	1,875,177	1,388,904
tri-strip	4,638,412	1,910,977	2,150,664	806,499	535,483
ind. tris	2,977,736	1,130,963	1,252,722	730,212	584,695
ind. strips	2,269,225	937,259	1,055,238	492,728	405,799

(b)

	X3D	X3DB	BG	SIG raw	SIG PNG
raw tris	16,324,694	2,026,318	6,981,552	2,618,082	2,009,258
tri-strip	6,647,483	2,742,016	3,087,816	1,157,931	741,350
ind. tris	4,399,737	1,539,600	1,745,436	1,018,161	883,318
ind. strips	3,411,800	1,304,490	1,420,958	693,683	607,876

Table 1: *Number of bytes for (a) 69,451 triangles bunny and (b) 96,966 triangles horse model, both with coordinates and normals (as X3D XML-encoding, X3D binary encoding with zlib compression, BG, SIG - 16 bit raw, and SIG - 16 bit with PNG compression).*

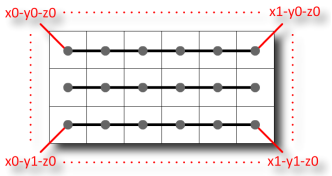

Figure 7: *Using the x- and y-position of the implicit mesh vertices as texCoords to directly map a vertex to its corresponding texel.*

5 Rendering

In this section, we describe the SIG and BG rendering methods, because they are critical for the overall results. However, the current and final extensions to any declarative approach (see section 6) additionally must be able to render regular grids for traditional image data like height maps and regular geometry images (cp. Figure 3).

5.1 SIG Rendering with GPU Decoding

The actual rendering method of the SIG data depends very much on browser and rendering-backend use. For the WebGL-backend we utilize the standard `<img>` and `<video>` tags to asynchronously download the pixel data to the browser and decode the information on the GPU or CPU. Most modern GPUs allow texture access in vertex shaders (which was introduced with Shader Model 3.0) and therefore support the GPU method. The Flash-backend always uses the slower CPU version (cp. section 5.2) since Stage3D[2] does not support texture access inside of the vertex shader at all.

For an efficient and fast rendering we use the *vertex texture fetch* feature of OpenGL ES 2.0 / WebGL [Marrin 2011]. This allows us to read data from textures inside a vertex shader. With this data, we can then transform the vertices and manipulate or set vertex properties like position, color, normal, and texture coordinates. As mentioned in section 4.1.2, even indexed rendering is possible by using a special index texture (see Figure 6). Thus, we can transfer the whole SIG decoding and most of the buffer generation process from the slower JavaScript layer directly to the GPU. By allowing multiple textures per vertex property (see section 6 for the node interface), the precision (as already described in section 4.1.2) grows until the GPU's vertex texture limit is reached.

To invoke the vertex shader appropriately, we render one very simple base mesh that directly provides the texture coordinates for ac-

[2]http://www.adobe.com/devnet/flashplayer/stage3d.html

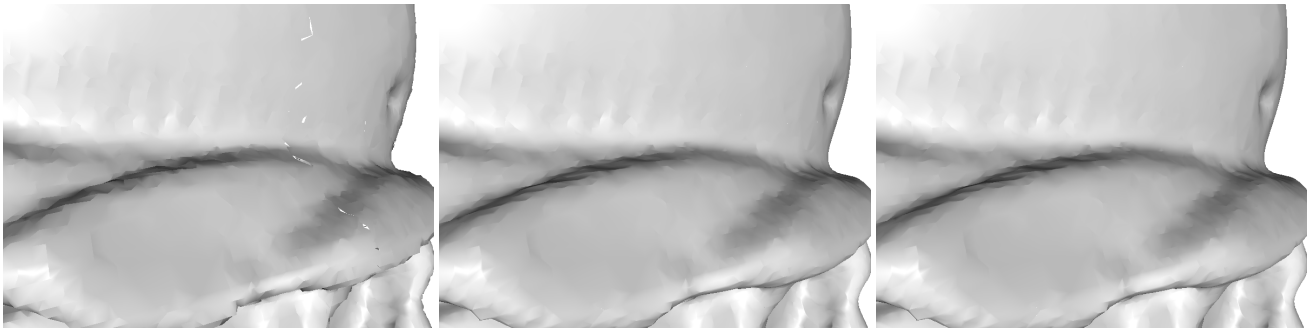

Figure 5: *Close-up on the Buddha model for 8 bit (left) vs. 16 bit encoding (middle) of the coordinate images. Especially for the positions, low precision leads to severe errors such as cracks within the mesh and at borders between mesh patches described by another coordinate image (where the bounding boxes are per patch). The rightmost image shows the 32 bit binary data encoding given as arraybuffer.*

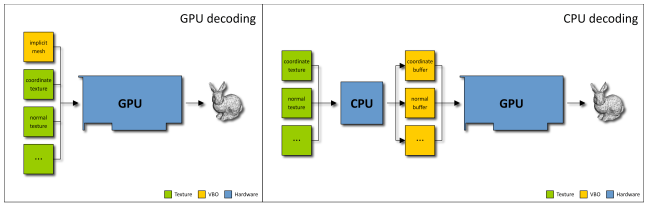

Figure 8: *SIG rendering with (left) and without GPU decoding.*

cessing the correct texel for the corresponding vertex, which is visualized in Figures 7 and 3 (right). Hence, only one Vertex Buffer Object (VBO) for these implicit mesh vertices (see Figure 7) needs to be generated by the JavaScript layer, which reduces the number of vertex buffer switches on rendering, since for all image geometries (assuming they don't have different chunk sizes) the same implicit mesh (i.e., the same VBO) is bound. All other data is directly uploaded as texture to the GPU. This not only decreases the processing time but also allows to store more mesh-data on the GPU since we keep the normalized and quantized data and decode and generate the 3D coordinates and other vertex properties on the fly on the GPU while rendering.

The decoding in the vertex shader is split up into two main parts. First, the xy positions of the implicit mesh vertices, which provide the texture coordinates as described above, are transformed such that the center of the texel is sampled instead of the corner to get the exact value of the data textures. After that, the full precision is calculated by shifting the summed coordinate samples of one vertex 8 bits to the left (i.e., multiplied with 256) and then adding the new precision sample of the vertex multiplied with 255. This is done in a loop until the desired precision n is reached. This sum is then divided by $2^{n \cdot 8} - 1$ to obtain the normalized result, which finally is transformed to its original value with the help of the bounding box of the respective mesh patch, which is separately stored in the geometry node's 'position' and 'size' fields. For the 2D texture coordinates a similar operation is applied, though 16 bit precision can be encoded in one RGBA texture. The normals however are stored in the rgb channels comparable to an object-space normal map. The colors obviously do not require any decoding. After that, further rendering works as usual.

5.2 SIG Rendering without GPU Decoding

With Flash or WebGL on older systems and some mobile devices that don't support Vertex Texture Units on the GPU, we provide a SIG software implementation as fallback. Here, all the data textures are decoded on the CPU and single VBO's are created for positions, normals, colors, etc., which are then loaded to the GPU similar to other geometry nodes (e.g., *IndexedTriangleSet*). This implementation is much slower (at least on recent machines) and we loose all the advantages of the pure GPU version, but at the moment it is the only way to guarantee that SIG runs on all systems. Figure 8 shows the difference between both implementations.

5.3 BG Rendering

As outlined in section 4.2, no decoding of binary containers given as *ArrayBuffer* is required, since they can be transferred directly to the GPU. Hence, rendering is very simple: the respective typed array views directly map to the required VBO data structures. The standard rendering code can be used out of the box. However, while rendering is very fast and easy, accessing the binary encoded data (BG as well as SIG) is much more difficult than for the traditional X3D-SAI [Web3D Consortium 2011] or DOM-based approach, where all index and attribute data are part of the XML representation. Though modifying single data values from large vertex attribute multi-fields is quite unusual in a web application anyway, one can still fall back to more traditional geometry representations as trade-off between speed and parameterizability.

5.4 Priority Controlled Rendering

All relevant use-cases and applications include more than a single mesh and therefore easily hundreds of external SIG or BG data references. Therefore, a simple question has to be answered: in what order should the chunks be downloaded and used for rendering? FIFO processing could be an option but does not always lead to the desired user-experience for progressively appearing of objects. Therefore, we propose a simple but efficient priority-controlled rendering mechanism, which combines the content- or content-type-specific user preferences with runtime behavior. We use three factors which influence the final chunk priority to control the ordering (compare Figure 9).

Content There could be a specific object in the scene, which should be loaded earlier, and therefore the user can set a factor per specific SIG or BG node tag.

Chunk type There is a specific chunk-type (e.g. positions or normals) which should be loaded first. This can be controlled by a global per scene and type factor. The standard values define the following order, where the first coordinate set (first 8 bit) are loaded at the beginning, afterwards normals and texCoords, and than the rest of the coordinate data. This leads to a nice progressive refinement behavior.

Size and view-frustum While rendering, the priority of yet not downloaded content is controlled by two additional parameters: the size of the options in world coordinates and whether

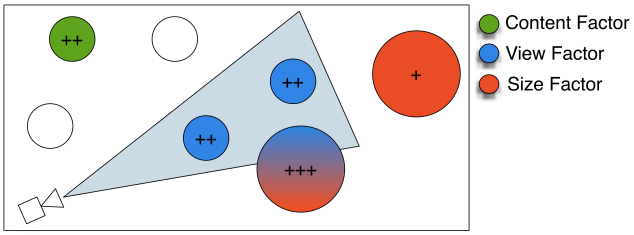

Figure 9: *Content, view and size controlled priority refinement.*

the object is inside of the view-frustum and therefore visible. The default factors increment the priority for both cases, to increase the chance that objects of the scene, which are most relevant, are downloaded earlier.

We are still experimenting with different formulas, but all those factors lead to a single priority number that is actually used in the download manager to determine which request is committed to the browser next. The download manager commits only a small number of request at once to assure an interactive overall user experience.

It is important to note that the requests to the browser are not guaranteed to be processed in a specific order. There are browser-caches, thread-managers, different file sizes, and server-side download mangers, which all influence the final result. Therefore the download manager is more used to provide a general behavior but cannot guarantee a specific order at all. However, we found that this is usually not a major issue. With Web applications, people are used to imperfect intermediate results (e.g. not all images are yet loaded on a standard HTML page) as long as the overall experience is pleasant and progressively improves.

5.5 Patch Creation

The overall goal of this effort is to increase the user experience by providing interesting and useful representations of the scene in a progressive manner more efficiently. Transferring a non-streamable single large binary representation which is loaded while a spinning wheel or even progress-bar is presented is not an option. Therefore finding a streamable and piecewise useful representation is key.

One simple way is to create and transfer patches as parts of the mesh. Interestingly enough all methods, regular Geometry Images, SIGs and BGs, need specific patch sizes to work as intended. The original Geometry Images method [Gu et al. 2002] only worked with genus zero meshes and had parametrization distortion issues. The original author proposed a multi-chart Geometry Image [Sander et al. 2003] which uses an atlas construction to map the surface piecewise onto charts of arbitrary shape. Our SIG method works with arbitrary large meshes but for streaming and rendering speed it makes sense to stick to the same grid size, which should be power-of-two, since non-power-of-two textures are not supported on most mobile devices. The BG encoding in addition has to deal with a hard API limit: WebGL (based on OpenGL ES 2.0[3]) and Flash 11 / Stage3D only support 16 bit indices [Marrin 2011]. Hence, indexed rendering only allows addressing a maximum of 65.535 vertices per draw call anyway.

Since the elements for SIG- and BG-based meshes do not really have to represent some closed surface patches, much simpler methods can be used to cut the triangles in sub-parts. We use simple KD trees to collect triangles in a minimal bounding box which share vertices for better cache and triangle-strip-generation operations.

[3] http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf

5.6 Interaction and Picking

Both implementations, SIG and BG, work really well with the existing render-buffer-based picking approach described in [Behr et al. 2010] for handling mouse or touch events and user interaction with the 3D scene. The BG implementation works almost out of the box (only the mesh's bounding box needs to be calculated on the CPU from the loaded `Float32Array` with the coordinates), in case the geometry's *Shape* node does not provide the 'bboxCenter' and 'bboxSize' hints. Similarly, for the SIG implementation only a few modifications of the – otherwise very simple – vertex shader used in the picking-pass are required to provide the coordinate data.

6 X3D Node Design

The current interfaces of both nodes, which we have implemented in X3DOM [Behr et al. 2010] and in the Instant Reality framework [FhG 2012], are tailored for performance, not generalized at all, and only used for prototyping. Any final interface must be harmonized to hold arbitrary binary references to vertex data (not just positions, normals, texCoords, and color). We only show their current state, which conceptually follows the respective X3D geometry nodes [Web3D Consortium 2011], for completeness' sake.

Our prototyped *ImageGeometry* node to represent SIGs, whose interface is shown below, reads the vertex properties from the attached image files. The 'implicitMeshMode' field defines the image mode as outlined in Figure 3 and defaults to our described SIG approach. As the name implies, 'implicitMeshCount' (which should be a power-of-two value) defines the size of the implicit mesh (and therefore the size of the attribute textures). The used primitive types are given via the 'primType' multi-field (e.g. triangle strips, triangles, or points). The multi-field 'vertexCount' describes how many vertices are used for each primitive type. To keep the number of draw calls to a minimum, both fields typically contain two values: the first bunch of primitives is described via triangle strips, whereas the rest of the mesh consists of those triangles that would introduce too many degenerated triangles in the striping pre-process.

The bounding box is described by 'position' and 'size', both of type *SFVec3d*, which is necessary to correctly scale and translate the normalized patch coordinates for the rendering and picking pass. The SFNode field index holds the optional index image (cp. Figure 6) to further reduce the size of the attribute textures, whereas the other MFNode fields refer to the other vertex attributes like coordinates, normals and colors. Note that in contrast to standard X3D geometry nodes the normals and texture coordinates must be provided, if lighting and texturing is needed.

```
ImageGeometry : X3DGeometryNode {
  SFString [in,out] implicitMeshMode   "auto"
  SFVec2f  [in,out] implicitMeshCount 256 256
  MFInt32  [in,out] vertexCount []
  MFString [in,out] primType    ["triangles"]
  SFVec3d  [in,out] position    0 0 0
  SFVec3d  [in,out] size        1 1 1
  SFNode   []       index       null
  MFNode   []       coord       []
  MFNode   []       normal      []
  MFNode   []       texCoord    []
  MFNode   []       color       []
  ...
}
```

Another experimental geometry node to test the different binary packaging methods is the *BinaryGeometry* (i.e., BG), whose node interface is shown next. The 'vertexCount' and 'primType' fields are defined as described above. Likewise, the index field as well as the other attribute fields hold the URL that refers to the respective

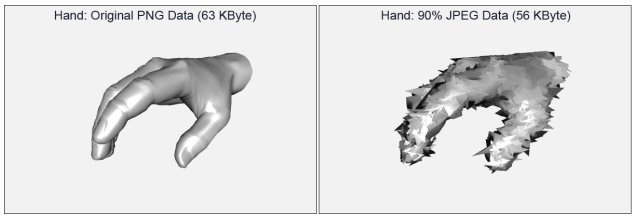

Figure 10: *Lossless vs. lossy compression of a 3D hand model.*

binary files to be loaded. Since ArrayBuffers can hold various attributes in an interleaved manner, the shown node interface is rather debatable, because in this case, one also needs to be able to specify the respective offset and stride into the array view.

```
BinaryGeometry : X3DGeometryNode {
  MFInt32  [in,out] vertexCount []
  MFString [in,out] primType    ["triangles"]
  SFString [in,out] index       ""
  SFString [in,out] coord       ""
  SFString [in,out] normal      ""
  ...
}
```

7 Discussion and Results

7.1 Image Compression

SIGs enjoy implicit compression from the image formats supported by browsers. RLE encoding can naturally reduce the size of a SIG container, as well as LZ DEFLATE lossless compression used by the PNG image format. Because SIGs come separated for index, vertex and normal data, different compression schemes can be used for each container. An interesting option we have not yet fully explored is normal compression via lookup tables in standard GIF files: geometry normals can be quantized into 256 directions, which are stored in the color table of a GIF image. The lookup table then simply indexes into this normal table.

As mentioned, we save geometry in several images, retaining their original structure and connectivity, since SIGs use images to store vertex and normal data in an indexed array. This is a major difference to other, seamless image formats [Purnomo et al. 2004; Gu et al. 2002]: vertex and normal data are not spatially coherent. A major downside to this storage type is that compression algorithms which exploit local coherence cannot work properly. For instance, JPEG or similar Fourier-based compression types will distort geometry quite radically. In figure 3 we show an example of an image containing vertex data compressed with a lossless scheme in direct contrast to a compression to 80% with JPEG.

Compressing SIGs therefore has to resort to classical non-lossy encoding such as RLE or LZW and hence provide only limited compression capabilities in their new domain. Because of these compression issues, streaming operations with lossy video codecs such as MJPEG can lead to distortions and loss of topology. To remain faithful to the original geometric properties, an MJPEG compressed movie stream of SIGs has to maintain a very low compression rate, which unfortunately defeats the purpose of the compression.

7.2 Dynamic Data Updates

Figure 11 shows two frames of a cloth simulation. The visualization is dynamically updated in real-time via MPNG image streams provided by a Tomcat server to efficiently update the vertex data. Likewise, videos with lossless encoding can be used for dynamic

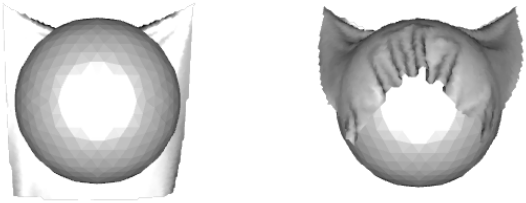

Figure 11: *Two frames of a cloth simulation that is streamed via MPNG to the client and visualized in real-time using SIGs.*

updates and animations. As previously discussed, this approach is also suited for streaming large geometries in an LOD-like way.

7.3 Load Timing Comparison

We have captured timings loading the Buddha model from a remote web server encoded as both BG and SIG, as well as the Horse and Bunny models, also encoded as BG and SIG, with Chrome 18 on a Windows 7 machine. All timings were taken with uncached, freshly started browsers. Averaging out variance due to network latency and other factors, the BGs will usually outperform the SIGs when loaded from a local resource with smaller models such as Bunny and Horse. However, the situation is turned on its head when we compared the results of a bigger model like the Buddha loaded from a remote server: here, the BGs usually take approximately 20% up to 100% (depending on the quality of the Internet connection) more time to load on average than their respective SIGs.

7.4 Image Operators

An interesting exercise is the application of image operators on SIGs. An adjustment such as contrast enhancement in image space translates to a mapping of points in 3D space. Therefore, image operators can be used for scaling, inversion other geometrically relevant operations on images containing coordinate information. An important aspect to first note is the spatial incoherence of vertex properties in SIGs (cf. section 7.1): compression algorithms seeking to exploit neighborhood similarities cannot be used in conjunction with our approach. This also affects the outcome of spatial filters applied to vertex property images. E.g., blur operators such as a *Gaussian blur* usually distort or destroy the geometry at hand, as do other filter operations such undersampling, averaging or detail enhancement and are therefore not useful as 3D operators.

The following operators have useful effects on SIG vertex containers: brightness (translation), contrast (resize along axis), gamma (stretch geometry along extremities), negative (invert positions, flip face directions) and color channel flipping (swizzel operator on vertices). The last two operations are format independent (LDR or HDR) that generate one-to-one mappings in color space. These operations retain the general structure of the geometry but may flip face directions from clock-wise to counter-clock-wise. A special class of image operators only work with set thresholds (usually given by an LDR representation) such as *Solarize*: a specific bound is chosen which limits the space of values and truncates or remaps values. In case of image solarization, a geometric volume limits the expansion of the model to an arbitrary threshold and reverses value growth beyond this point, making geometry grow into itself.

7.5 Applications

Figure 1 (left) shows a walk-trough application realized with SIGs in X3DOM. To ease the application development by excluding the

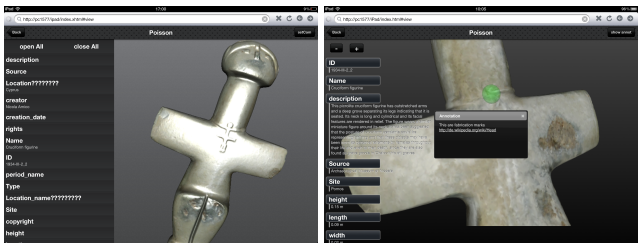

Figure 12: *Cultural heritage data viewer on iPad 3 WebGL-enabled browser. This use case is explicitly build with binary geometry (BG) models, since mobile graphics chips do not yet support vertex textures units and therefore no GPU-based SIG decoding.*

vertex attribute data from the HTML file on the one hand and to compress this large geometric data set on the other hand we used our image geometry approach to represent and render the vertex attribute data by utilizing images as binary data containers. A Cultural Heritage application running on an iPad is shown in Figure 12. For this use case we explicitly use BG models, since currently only few mobile graphics chips support Vertex Textures Units needed for GPU-based decoding of SIGs. Another CH example with a high-resolution mesh of a laser-scanned historical object that is compressed via the SIG approach is shown in Figure 1 (right), where it can be seen that point rendering is also supported.

8 Conclusion and Future Work

The SIG and BG approaches both support the overall goal of more efficient data container for declarative 3D scenes. Both work with arbitrary point, line or triangle primitives and are much easier to encode, decode and render compared to the original Geometry Image approach proposed by Hoppe et al. SIGs usually lead to smaller files but need hardware support for Vertex Texture Units for efficient rendering. BGs work perfectly with WebGL and Flash even on older or mobile devices and are even easier to generate.

We will direct future work into various areas of research. The current design and implementation just presents the current state of this experiment. The current X3D / X3DOM node design for instance is also only for prototyping and needs to be harmonized into a final geometry container that holds any type of external binary reference, which may include a data flow concept to configure the actual CPU / GPU processing load. Furthermore, we would like to include some form of hierarchical refinement for images, better striping methods to reduce the illegal triangle count, and some form of interleaved vertex data encoding to support even better progressive updates.

Acknowledgements

The described work was carried out in the project *v-must*, which has received funding from the European Community's Seventh Framework Programme (FP7 2007/2013) under grant agreement 270404.

References

ASIRVATHAM, A., AND HOPPE, H. 2005. Terrain rendering using GPU-based geometry clipmaps. In *GPU Gems 2*. Addison-Wesley, ch. 2, 27–45.

BEHR, J., AND ALEXA, M. 2002. Fast and effective striping. In *Proc. of OpenSG Symposium 2002*. Report 02i015-ZGDV.

BEHR, J., JUNG, Y., KEIL, J., DREVENSEK, T., ESCHLER, P., ZÖLLNER, M., AND FELLNER, D. 2010. A scalable architecture for the HTML5/ X3D integration model X3DOM. In *Proceedings Web3D 2010*, ACM, New York, USA, 185–193.

BEHR, J., JUNG, Y., DREVENSEK, T., AND ADERHOLD, A. 2011. Dynamic and interactive aspects of X3DOM. In *Proceedings Web3D 2011*, ACM Press, New York, USA, 81–87.

DONNELLY, W. 2005. Per-pixel displacement mapping with distance functions. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, ch. 8, 123–136.

FHG, 2012. Instant Reality. http://www.instantreality.org/.

GU, X., GORTLER, S. J., AND HOPPE, H. 2002. Geometry images. In *Proceedings SIGGRAPH '02*, ACM, New York, NY, USA, 355–361.

HERNÁNDEZ, B., AND RUDOMIN, I. 2006. Simple dynamic lod for geometry images. In *Proceedings of GRAPHITE '06*, ACM, New York, NY, USA, 157–163.

HICKSON, I., 2012. HTML5 (W3C Working Draft). http://www.w3.org/TR/2012/WD-html5-20120329/.

JI, J., WU, E., LI, S., AND LIU, X. 2005. Dynamic lod on gpu. In *Computer Graphics International 2005*, 108 – 114.

JOVANOVA, B., PREDA, M., AND PRETEUX, F. 2009. Mpeg-4 part 25: A graphics compression framework for xml-based scene graph formats. *Signal Processing: Image Communication 24*, 1-2, 101 – 114. Advances in three-dimensional TV and video.

KHRONOS, 2012. Typed array specification. http://www.khronos.org/registry/typedarray/specs/latest/.

LOSASSO, F., AND HOPPE, H. 2004. Geometry clipmaps: terrain rendering using nested regular grids. In *ACM SIGGRAPH 2004 Papers*, ACM, New York, NY, USA, SIGGRAPH '04, 769–776.

MAGLO, A., LEE, H., LAVOUÉ, G., MOUTON, C., HUDELOT, C., AND DUPONT, F. 2010. Remote scientific visualization of progressive 3d meshes with x3d. In *Proceedings Web3D 2010*, ACM, New York, NY, USA, Web3D '10, 109–116.

MARRIN, C., 2011. Webgl specification. https://www.khronos.org/registry/webgl/specs/1.0/.

PRAUN, E., AND HOPPE, H. 2003. Spherical parametrization and remeshing. In *ACM SIGGRAPH 2003 Papers*, ACM, New York, NY, USA, SIGGRAPH '03, 340–349.

PURNOMO, B., COHEN, J. D., AND KUMAR, S. 2004. Seamless texture atlases. In *Proceedings of the EG/ACM symposium on Geometry processing*, ACM, New York, USA, 65–74.

SANDER, P. V., WOOD, Z. J., GORTLER, S. J., SNYDER, J., AND HOPPE, H. 2003. Multi-chart geometry images. In *Proceedings of the EG/ACM symposium on Geometry processing*, Eurographics Association, Aire-la-Ville, Switzerland, SGP '03, 146–155.

SCHWARTZ, C., RUITERS, R., WEINMANN, M., AND KLEIN, R. 2011. Webgl-based streaming and presentation framework for bidirectional texture functions. In *Proceedings VAST 2011*, Eurographics, 113–120.

SECONDLIFE, 2009. Sculpted prims: Technical explanation. http://wiki.secondlife.com/wiki/Sculpted_Prim_Explanation.

STOCKER, H., AND SCHICKEL, P. 2011. X3d binary encoding results for free viewpoint networked distribution and synchronization. In *Proceedings Web3D 2011*, ACM, New York, NY, USA, Web3D '11, 67–70.

SUN, 2004. Fast infoset. http://java.sun.com/developer/ technicalArticles/xml/fastinfoset/.

W3C (COMMUNITY GROUP), 2012. Declarative 3D for the Web Architecture. http://www.w3.org/community/declarative3d/.

W3C, 2011. Efficient xml interchange (exi) format. http://www.w3.org/TR/2011/REC-exi-20110310/.

WEB3D CONSORTIUM, 2011. Extensible 3d (X3D). http://www.web3d.org/x3d/specifications/.