A pipeline for the creation of progressively rendered web 3D scenes

Alun Evans*, Javi Agenjo**, Josep Blat**

Affiliations

*GTM - Grup de recerca en Tecnologies Mèdia, La Salle - Ramon Llull University, Quatre Camins 30, 08022 Barcelona, Spain. Tel: +34 932 90 24 74

** Grup de Tecnologies Interactives, Universitat Pompeu Fabra, Roc Boronat 138, 08018, Barcelona, Spain. Tel: +34 93542 2173

aevans@salleurl.edu; javi.agenjo@upf.edu; josep.blat@upf.edu;

ORCID IDs:

Alun Evans: 0000-0002-5713-0282

Josep Blat: 0000-0002-5308-475X

Abstract

We present an end-to-end pipeline for the export of 3D scenes from content creation tools to a real-time rendering engine in an embeddable web-page, including a novel system for compression/decompression of textured polygonal meshes. We show that the compression/decompression outperforms the best state-of-the-art non-progressive alternative, especially as bandwidth increases, showing that web-specific techniques should consider the whole user pipeline. Our pipeline also includes progressivity, which is paramount for a good interactive user experience, and permits full user interaction with lower resolution versions of the 3D scenes, while progressively higher resolution data is downloaded. Finally, we discuss how our method may be used in the future to facilitate the transfer of animated meshes.

Keywords

3D, Web, Graphics, Pipeline, Compression, Mesh

1. Introduction

The web has become a truly multimedia experience. We have moved away from the concept of 'web pages' and embraced the idea of 'web applications' – multimedia rich, client-server systems, which allow users to engage in both of the original goals of the web: browsing and editing content. The nature of the web means that several communities contribute regularly to its ongoing development, and the multimedia research community is no exception, with recent efforts contributing in the fields of virtual reality [1], video labelling [2], and remote rendering [3].

One factor in common with many of these improvements is the HTML5 standard and its associated APIs. One such API is WebGL, which permits access to the GPU from the web browser, and is supported by all major browsers (whether on desktop or mobile). Since its initial release in 2011, there has been a steady rise of applications, technology, and research concerning 3D graphics on the web [4]; and web developers now have a choice of several higher-level engines (such as three.js [5]), which use WebGL to facilitate the development of interactive 3D web applications.

In the entertainment industry, 3D scenes are invariably created and/or edited in one of a variety of professional modelling and animation software packages, such as Autodesk Maya and 3ds Max [6]. Exporting resources (or *assets*) from these packages in a format suitable for further processing and final use is frequently a central part of an *asset pipeline*: a workflow to transform artistic ideas into their final 3D form. Asset pipelines form the backbone of most (if not all) Digital Asset Management (DAM) systems [7], and the design and creation of the pipeline is frequently one of the first tasks of any digital media production [7].

Web-based productions also benefit from the presence of an efficient asset pipeline; yet they must consider an additional step in the pipeline, which is a central issue facing any interactive 3D web application: that of *data transmission*. 3D data tend to be quite large, and any asset must be exported in a suitable format, uploaded correctly to a server, and be fully downloaded to the browser, before it can be rendered. Thus, a key parameter within any asset pipeline for an interactive 3D web application is the time taken to *transmit and decode* the data, as any delay has a negative effect on user experience [8, 9].

A typical 3D scene consists of a variety of assets [10, 11]:

- 3D *object* data, usually represented as a triangular mesh or implicit surface
- *Materials*, which specify the colour of an object and how it should interact with light
- *Textures*, 2D image files which store per-pixel information to be used by the materials
- *Animations*, which specify the spatial movement of 3D objects through time
- Other components, such data relating to *lights* and *cameras* (position, direction, etc.)

Exporting these data to a web application presents different levels of challenges. Data regarding materials, cameras, lights and the like, is essentially metadata and can be specified as a lightweight text file, easily compressed and transmitted. Textures are stored as 2D images, whose compression and transmission is a well understood problem that has resulted in the very common standard formats (such as .jpg and .png) used every day on the web.

This leaves data related to 3D objects and animation. A very common representation of 3D objects is the triangular mesh [10], which features geometry position and connectivity data (at a minimum). In its raw format, a large mesh may occupy a large amount of data; thus the compression of meshes has been very well studied in the literature [12]. Mesh data can be encoded using single-rate methods, where all data is compressed and decompressed as a whole, or progressive methods, where a 3D mesh can be constructed continuously from a coarse to fine representation, as more data is retrieved.

In the Web 3D community the issue of 3D data representation is highly topical, because most established mesh encoding techniques are optimized purely in terms of bits per vertex (bpv) or rate-distortion (R-D) performance, and ignore the important trade-off between compression rate and decompression time [12]. It is only recently that researchers have noted that, given that web-based applications involve real-time transmission of data, for the final user experience, the decompression time in the JavaScript layer of the browser is of equal or greater importance than the compression rate [13–16].

In this paper, we present results which support this statement. We present a novel mesh compression method, and furthermore propose a structure for an asset pipeline which exports entire 3D scenes to the web. We also present an implementation of that pipeline, and demonstrate its advantages over a non-pipeline approach.

The main contributions of the paper are:

- A holistic, application-level view of 3D graphics on the web, in the form of an end-to-end pipeline which exports full scenes directly from a variety of popular modelling packages, uploads them to an account-controlled server, and outputs a sample WebGL rendering application that can be embedded into any web-page, or used as part of a custom application. Our results demonstrate that using a such a pipeline reduces completed export time for 3D scenes by 37%.
- A single-rate mesh compression method which prioritises decompression rate over file size. It improves on Google's WebGL-Loader [13] by employing efficient index buffer storage and normal vector compression; and improves on Google's recent Draco [17] mesh compression system by demonstrating clearly the benefits of fast decompression vs. overall file compression.
- A progressive compression/rendering method, which uses spherical Fibonacci points to store vertex normals at lower resolutions. Our method achieves considerably faster (4x speed improvement) transfer of meshes compared to similar techniques [14], and features support for multiple meshes, materials and textures, unlike other approaches [16].

For the particular problem of mesh encoding/decoding, we quantitatively and qualitatively compare our results to previous approaches, and evaluate according to metrics of compression and decompression performance, progressive vs single rate coding, and flexibility to deal with full scenes and more complex materials. For single rate encoding, we employ the McGuire 3D mesh dataset [18] to evaluate our technique against the state-of-the-art.

Beyond this, central to our approach is the concept that meshes are merely part of a more complete package of data which needs to be downloaded and processed, and as such any mesh transmission system should be designed taking the whole graphics pipeline in consideration. In the conclusion, we discuss the further advantages of our approach, for example when dealing with more complex materials or animations, particularly with regards to future work.

2. Related Work

2.1 Transmission of 3D data for the web

3D web pages are relatively uncommon, and for several years were mostly represented by declarative technologies developed in the academic domain [19, 20], or application specific virtual worlds requiring custom installations [21]. However, 3D web applications have been growing in popularity since the release of WebGL in 2011. WebGL is a web-specific version of the OpenGL graphics API (more specifically of the restricted embedded systems API, OpenGL ES 2.0), and allows access to dedicated graphics processing hardware (the GPU) directly from the browser (via JavaScript). It is now fully supported in the latest versions of all major browsers. WebGL and associated HTML5 APIs (such as WebAudio[1]) are in many respects enabling technologies, as they break down the barriers for the development of browser-based interactive multimedia applications. Nevertheless, they also open up new research challenges for the best way to transmit and interact with hybrid data (be it 3D, 2D image/video, audio, or text) [22, 47-49].

3D data is typically large, and transferring it to a remote client for rendering is a persistent problem for all web 3D applications. This is particularly relevant for our work, in that the multimodal visual data is stored in files which reach many hundreds of megabytes in size - simply “waiting for them to download” does not provide an optimal user experience. While a naive approach might be to simply compress the data using any number of established and powerful algorithms, Limper et al. [15] show that straightforward data compression may not necessarily be the solution, as the decompression time in a browser-based context may outweigh any benefits gained in terms of compressed data, particularly as bandwidth speeds increase. For a more complete overview of these issues, and the current state of the art with respect to web-based 3D, including techniques of remote rendering and progressive transmission, we refer the reader to a recent survey paper [4].

The majority of the previous work in the field of web 3D transmission focuses on the compression of meshes [12], which consist of position and index data, with an optional normal vector, colour, and texture coordinate data. In parallel, 3D point clouds are increasingly available and used, due to the widespread use of 3D scanning (see for instance [23–25]), and volumetric rendering is central when dealing with 3D representation of medical data [26].

It is also important to note that the community is beginning to realise that the present and future of 3D on the web consists of the transmission of an entire 3D scene (as discussed above), as opposed to meshes in isolation, as this is more representative of the needs of a typical web 3D application (for example, a game or an interactive experience). For example, Zampoglou et al. [27] attempt to address this issue by using the MPEG-DASH standard to encode and transmit an entire X3D scene. This is an interesting approach, as the MPEG-DASH encoding schema is in theory designed to cover adaptive streaming of all information types. In practice, however, the schema is highly tailored to audio and video, and 3D data requires some formatting in order to ‘fit’. Other content distribution techniques for web-based 3D graphics (such as server based rendering or hybrid server/client approaches), along with their positive and negative points, are surveyed extensively in [4].

1 http://www.w3.org/TR/webaudio/

2.1 Compression

In the following two subsections, we survey work related to data compression for web-based 3D graphics. We first survey single rate compression (where compression and decompression are carried out in a single instance), before surveying progressive compression (where data is compressed in a manner such that it can be decompressed at increasing levels of detail).

2.1.1 Single-rate compression

Data stored as binary geometry has the advantage that it can be directly transferred to the GPU for rendering. This approach is taken by the X3DOM framework [20] which uses a simple binary encoding format with 16-bit integer quantization, where floating point data is stored as an offset to a common point - or points, such as bounding box limits. Dequantization can be done on the GPU, which means that processing in the slower JavaScript layer is completely skipped. Nevertheless, this also means that no real compression can be carried out on the data. In [28], this issue is addressed by proposing a `Sequential Image Geometry' format, where mesh data is stored within an RGB image format, and then compressed using a lossless technique such as PNG, which, after transmission, can then be decompressed using low-level code within the browser, and uploaded as texture data to the GPU. An advantage of this approach (using a compressed image as a data vector) is that it takes advantage of the fast (native code) decompression of the image data which is permitted by the WebGL/GPU interface (thus skipping data decompression in the JavaScript layer). But passing vertex data directly to the GPU makes it impossible to use an index buffer, which leads to increased memory overhead.

Thus, to take advantage of the index buffer, it seems unavoidable that some decompression should be carried out on the CPU. For non-web applications, a commonly used open binary mesh format is OpenCTM [29], which is highly portable, provides good compression rates, and is relatively fast to decompress in a desktop context. It is built on entropy reduction and Lempel-Ziv-Markov chain algorithm (LZMA) entropy encoding, which combines the classic LZ77 algorithm with Markov chains. Vertex positions are stored as offsets to a network of cells, and delta coding is used to store only the difference of each vertex to the cell center. Delta coding is also used to store connectivity data. The use of delta coding reduces the entropy and therefore supports good compression when using LZMA. A JavaScript implementation of OpenCTM decompression is also now available [30], which makes the algorithm available for web-based 3D applications. The primary disadvantage of using this technique for the web is that has been demonstrated recently that LZMA decompression in the JavaScript layer of the web browser is roughly an order of magnitude slower than in native code [31].

Perhaps the most widely known technique to compress mesh data for web-based applications is Google's WebGL-Loader[13, 32]. The technique relies on bounding box quantization of vertex positions, and using delta coding to transmit the differences between values, as above. It can be decompressed very quickly in JavaScript, with some aspects (such as the dequantization) being pushed to the GPU. A vertex cache re-ordering algorithm [33] is used to cluster faces sharing the same vertex data to deal with mesh connectivity. Further vertex buffer reordering is then carried out to permit the implementation of a “high water mark” coding algorithm for the index buffer, where the value stored for each index represents the difference to the highest value seen in the buffer up to that index. Delta and high-watermark codings generate codes of different byte lengths, and thus are only effective when the file format used to transmit the data supports variable byte encoding. Thus, WebGL-loader stores data using UTF-8 strings, which support variable byte encoding and, crucially, are decoded not by the JavaScript layer, but by the underlying native-code of the browser application. This fast decompression was the primary motive to use of the UTF-8 format, yet it comes with several downsides, most notably the block of surrogate pairs (reserved codes which permit encoding of multi-byte codes) which limits any vertex buffer to a maximum 55,296 values. WebGL-loader overcomes this by splitting larger meshes into chunks smaller than this value. Besides being an inelegant restriction, a disadvantage of this strategy is that extending the technique into progressive transmission (where lower resolution meshes are downloaded and rendered first) becomes complicated to implement, due to the need to constantly track the current state of the various sub-meshes.

Google recently released a new mesh compression system, called Draco [17]. Draco offers several improvements over WebGL-loader, not least the removal of dependency on the UTF-8 format, and improved compression performance. However, this improved compression clearly affects the decompression performance, which is considerably worse than WebGL-loader (as demonstrated in the result section below). Furthermore, Draco does not support progressive encoding/decoding, which we discuss in the following section.

2.1.2 Progressive compression

Research into user interaction on the web led Nielsen [8, 9] to propose three response time limits to keep in mind when optimizing web and application performance:

i) 0.1 second – the limit for user feeling that the system reacts instantaneously
ii) 1.0 second – the limit for user's flow of thought to stay uninterrupted, even though the delay is noticed
iii) 10 seconds – the limit to keep the user's attention. Longer delays will often see a user leave the site immediately

Based on these limits, for a web-based 3D rendering, it seems that a progressive encoding scheme, where lower resolution versions of the mesh are rendered while higher-resolution data continues to download, opens the door to a significantly better user experience. The decision on whether to use a progressive encoding technique depends greatly on the application: when user-experience is paramount (such as public display of large models, catering to a variety of bandwidths), or when view-dependent level-of-detail is required (i.e. as the camera view focuses on specific areas, more detail of the mesh in these areas is loaded), it is logical to engage a progressive technique.

Seminal work on progressive meshes was carried out by Hoppe [34], where triangles and edges are sorted according to a series of rules, and edges are collapsed (their constituent vertices merged) one-by-one, thus reducing mesh complexity. Each vertex split operation stores a code, which can then be used to recreate the geometry when decoding. Turning to the web, progressive mesh decoding is difficult to implement in JavaScript, as it requires the engineering of a robust mesh representation structure which can be parsed quickly. Lavoue et al. [35] tackle this with an efficient implementation of the half-edge structure in JavaScript, which enables them to implement a version of the valence coding technique of Alliez and Desbrun [36].

Limper et al. [14] sacrifice efficient compression rates for a fast and highly progressive system. Their method involves reorganising the geometry buffer to not waste bandwidth. It provides an elegant solution which is integrated into the X3DOM framework. The work presented in [16] uses [31] to implement a hierarchical patch-based method for progressive and view-dependent visualisation of large meshes. The system features fast decompression and efficient compression, but it currently only supports visualisation of cultural heritage, and does not support the texturing and materials required by many digital entertainment scenarios.

2.2 Coding methods

The technique used to encode/decode the data is critically important to any compression algorithm involving a real-time application, as the decompression time (i.e. how fast the data can be made usable, after being initially accessed) can impact greatly on user experience (as per Nielson's limits, discussed above [8]). Entropy coding, where frequently occurring patterns are represented with few bits, and rarely occurring patterns are encoded with many bits, is one of the most common and effective methods of lossless compression. Its decoding has long been seen as a bottleneck in many compression algorithms; indeed, as [31] shows, JavaScript implementations of several entropy decoding algorithms are an order of magnitude slower than their C++ counterparts - a statement which is supported by the experimental results presented later in this paper. The only scenario in which entropy coding can be used effectively is the HTTP transfer layer, where the gzip (entropy coding) algorithm is implemented at a very low level (both on the server for compression, and on the browser software for decompression). Thus, as custom entropy coding on the web involves a high JavaScript decoding cost, it means that other strategies, such as delta or the high-watermark coding methods mentioned above, are more suitable for 3D web purposes.

3. Mesh encoding and transmission: theoretical approach

In this section, we present the theory behind our approach for progressive mesh transfer for the web, which prioritises fast decompression over absolute file compression to improve on the state of the art. Our encoding method is designed to be *client agnostic*. While other approaches require the use of a custom framework [27] or specialised shader code [14] to be used, we have specifically created our progressive transmission technique in a way that can be integrated into any existing pipeline, and use any client-side rendering code.

3.1 Mesh attribute encoding

Attributes are properties of the mesh (such as position vector, normal vector, texture coordinates, colour etc.). We use common quantisation [12] techniques to reduce the number of bits required to store vertex positions. The data format we employ for transmission (see below) means that negative integer values cannot be stored. Thus, delta encoded quantized values are interleaved as described by [13]. De-quantization can be carried out either in JavaScript or directly on the GPU, however our tests show that using the GPU for this purpose provides only a negligible decrease in decompression time. Furthermore, dequantizing in JavaScript removes the need to write custom shader code, which permits use in any client application, and allows easy integration with popular web-based APIs such as Three.JS, which manage their own material/shader pipelines.

3.1.1 Octahedral normals

Other than position, the most common attribute to be transmitted is the vertex normal vector (usually calculated by averaging each vertex's surrounding face normals, but for meshes generated for digital entertainment, it is common for the artist to specify custom normals [11]). Normals are usually represented by three component unit vectors. This means that they can be quantized to positive integer values and encoded as per-position attributes. While such quantisation of normals is quite precise (e.g. encoding a 3D unit normal vector using 11 bits per axis leads to an accuracy of three decimal places), we employ a discretisation strategy which leads to better compression results. By projecting the normals onto a unit octahedron, and unfolding the faces of that octahedron into a plane, it is possible to encode the normal as a two-component, instead of a three-component, vector, on a scale of 0-1 (the idea of projecting onto planar geometric surfaces was first proposed by [37]). This scale can then be quantized to a predetermined bit depth; if 8 bits per component are used, the normal can be stored in a maximum of 16 bits, effectively discretising the normal vector to one of 2^{16} values. In [38], the authors show several examples comparing this discretisation with uncompressed 3-component normals; with barely perceptible differences in rendering quality. The two components of the octahedral normal can be delta encoded as we do for other attributes.

3.1.2 Fibonacci Normals as an Alternative

The obvious advantage of encoding the normals in two 8-bit components is that the compression performance is considerably better than when using three 11-bit components (used for the quantized XYZ normal). Nevertheless, if we could somehow reduce the number of components required from two to one, we should be able to obtain further improvements in compression performance. The simplest method of storing normals as a single component is to use an array of vectors, and store (for each vertex) the index in this array. To be useful in a general context, the ideal structure of this array would be the discretisation of a unit sphere, where each normal of the mesh is quantised to a point on this sphere.

The quality of the sphere discretization depends on the quality of the distribution of the sampling points. To select the sampling points we have resorted to the spherical Fibonacci (SF) points [39, 40] as these points have intrinsic high quality properties regarding the spherical cap discrepancy and the inter-samples distance [41, 42]. Moreover, the SF point sets have already shown to outperform state of the art solutions in different problems where a uniform distribution of points over the sphere is required (e.g. [41–44]). In practice, the SF points are generated by using the Fibonacci ratio to evenly distribute the points over a spiral which covers the sphere from its north to south poles, and the points are stored in an array.

There are several theoretical advantages of mapping normals to Fibonacci sphere points, with regards to compression, which we now discuss. As mentioned above, it enables the normal to be stored to a single value which represents the index in a table of Fibonacci points on a sphere. This table can be generated quickly at run-time, thus leading to very fast and efficient manner to transmit the normals. Furthermore, transmitting a single component also enables us to more precisely control the balance between precision and compression performance. At lower precision (fewer Fibonacci points generated), the delta encoding permits the normals to be encoded in a single-byte, which is clearly an improvement over the 2-byte octahedral encoding.

The research question regarding the use of Fibonacci normals is whether they produce sufficient quality during rendering. In the results below we carry out several tests in order to answer this question.

3.1.2 Texture coordinates and other attributes

Texture coordinates can be encoded by quantising to a desired bit depth, and delta encoding as above. 11-bit quantisation will generate pixel-level precision in a 2048x2048 pixel texture, which is considered sufficient for web applications (larger, 4096x4096 textures are rarely used due to the transmission overhead). One potential issue with texture coordinates is that of texture seams, where the same vertex shares multiple texture coordinates (whose use depends on the face being rendered). Texture seams present an interesting problem in terms of pure mesh compression research, however from the perspective of coding a fast rendering application, the desired configuration is to present the GPU with consistent, aligned memory across all attributes, each index accessed according to a single element buffer. As such, vertices on texture seams are duplicated in our current system. For most meshes, this results in a small increase in final file size, but this is compensated by the simplicity of the approach, which fits in with our stated design goal of creating a client-agnostic system.

In this paper, we do not present results featuring other attributes such as tangent vectors or vertex colours. However, using the techniques presented above, any attribute could be added and encoded for transmission, without any major restructuring of the overall method.

3.1.3 Connectivity

Triangle reordering algorithms are typically used to cluster indices of triangles that share the same vertices, which is useful for vertex-caching purposes on the GPU. This reordering also has an understandably high effect on the efficiency of delta compression, as it reduces the average delta value within the index buffer, which leads to encoding in fewer bytes. WebGL-loader uses Forsyth's triangle reordering algorithm [33]. We implemented this algorithm, as well as one other well-known alternative, the Tipsify algorithm [45], as representatives of state of the art triangle reorganising algorithms.

We further compress the index buffer by attempting to encode neighbouring triangles using 4 indices rather than 6. It can be proved that for any given set of 3 indices forming a triangle, *ABC*, if $A > B$ and $B > C$ then $C < A$. We can use this single bit of information to encode a `neighbouring triangle' flag in the index buffer. If, when reading triangle *ABC* from the index buffer, we see that $A < B$, then we can read a single further index, *D*, and draw triangle *ADB*. When encoding this information, we may have to re-specify the winding order of certain triangles to ensure that the $A < B$ test works as expected, but this is a minor inconvenience for the effective reduction is index buffer size. The gain of this technique clearly depends on the number of neighbouring triangles encoded in the index buffer, again emphasising the importance of the triangle reordering. When encoding the indices using this method, the high-watermark must be advanced by multiples of 3. This is because the index buffer may need to be re-ordered to ensure that the vertex winding order, for each triangle, permits multiple paired triangles in a row, which means that there is a maximum step in the high-watermark of 3 (as opposed to 1).

3.1.4 Per axis optimisation

In certain meshes, if the bounding box dimensions for a given axis are shorter than for any of the others, by an integer factor o more, then it is possible to reduce the quantisation bit-depth for that axis, without losing overall precision. For example, in the case of the Happy Buddha mesh, the x- and z-axes of the bounding box are less than half the size of the y-axis. Thus, these axes can be encoded with one bit less precision, and still maintain the same real-world precision as the y-axis.

3.1.5 Transmission format

The data format used for transmission is important as both delta and high-water-mark encoding generate variable-byte integer (varint) values, and thus a variable byte format would ensure optimum compression. Perhaps WebGL-loader's biggest drawback is its reliance on the UTF-8 format, as discussed in the Related Work section above.

An alternative data storage strategy, described briefly earlier, is to bit-shift the data into RGBA colour channels and save this using a lossless image compression format, such as a PNG, advantageously using one of several well-established optimization algorithms. There remains a considerable bottleneck for web-visualisation, however: that of reading the data from the image into the JavaScript layer. Our simple tests show that reading the pixel data for a 4096x4096 image (whether from a HTML5 canvas, WebGL context, or direct decoding) into a JavaScript Typed Array takes a very minimum of 500ms on current hardware, an unacceptable delay, as discussed in the results section below. Researchers in [28], presented earlier, avoided this bottleneck by not storing index data and decoding all the geometry on the GPU; however, this leads to reduced compression performance, and also means that the vertex transform caches on the GPU cannot be exploited, and no indexing can be used.

Thus, a custom varint format seems a better option. Such a format requires bit-shifting operations in order to read it correctly, but our hypothesis was that the bit-shifting speed currently available to modern browsers, thanks to the JavaScript Typed Array specification, would be sufficient. Our format is a simple base-128 varint format (stored with the *.b128 extension), where integer data is stored in groups of 7 bits, and every 8th bit is used as a flag to indicate whether the next byte in the sequence is a continuation of the same integer or not. Assuming a 32-bit CPU, this format allows 29 bits of precision for the index buffer, which should be enough for the majority of 3D meshes suitable for real-time rendering (29 bits permits over 536 million individual vertices). The format is also highly suitable for compression by the HTTP gzip implementation, with 'free' efficient compression and fast decompression, as discussed above.

Beyond the binary b128 file, we also store a JSON file with metadata regarding the mesh, and information regarding the material associated with the mesh. This is particularly useful when it comes to integrating the b128 into an established workflow, as described below.

3.2 Progressive Rendering

Progressivity is paramount for interactivity. Our system takes advantage of the fact that aggressive quantization of geometry results in many triangles collapsing to lines or even to single points. Thus, we harness this to create a system of progressive rendering where lower precision is downloaded first, to create a level-of-detail quantisation of geometry. To this we add differing amounts of normal compression at each level of data. We discussed above that Fibonacci normals provide excellent compression with small number of points, but led to bad visual results at larger numbers of points. However, at lower resolutions of geometry, the resolution of the mesh is low enough that highly quantised normals do not further reduce the perceived quality. Thus, we compress normals using the Fibonacci approach at lower resolutions, and keep octahedral normals (with better visual quality) at larger ones.

Of course, the number and precision of lower-resolution meshes can change depending on the user's needs. However, our empirical tests show that a 3-stage refinement process offers a nice balance between file size and user-experience, per the following stages:

- 1st stage: 7-bit bounding box quantisation, 256 Fibonacci normals, low resolution textures
- 2nd stage: 8-bit bounding box quantisation, 256 Fibonacci normals, low resolution textures
- Final stage: 11-bit bounding box quantisation, octahedral normals, high resolution textures

3.2.1 Client-side considerations

As discussed, we have designed this encoding schema in a way such that it can be used in any client-side renderer, using any shader. We have successfully implemented decoders in two 3D web engines, three.js [5] and WebGLStudio [46]. The actual decoding code is identical in either case, as it outputs vertex arrays which are ready to uploaded directly to the GPU.

The decoder downloads each level of detail in turn, interactively rendering the lower resolution meshes as it downloads the higher resolution data.

As alluded to in the list above, our progressive method also features a multi-resolution material system, this is described further below.

4. Asset Pipeline

As mentioned in the introduction, much of the academic literature on efficient representation of 3D content for the web focuses almost exclusively on mesh compression; and including, at most, support for vertex colour or texture coordinate attributes. Yet in the digital production world (such as videogame and digital cinema production, or architecture), models are invariably stored as several sub-meshes, each with different materials and shading groups. Our JSON/b128 format addresses these needs, as it permits the representation of such complex scenes; specifically:

- Multiple groups, each representing a different mesh
- Multiple shading groups, where different materials can be applied to different groups, or even different areas of a single group
- Material description featuring diffuse, emissive, and specular colour
- Diffuse, normal and specular texture maps
- Multi-resolution textures to fit the resolution of the stages of the progressive geometry

The final item in the list above refers to the likelihood, when using progressive techniques, that high resolution textures will not have downloaded when the first stage progressive geometry is available. This would result in rendering an un-textured geometry, which is not desirable. To avoid this scenario, our material technique supports multi-resolution textures for progressive meshes, where lower resolution geometry is rendered with lower resolution textures.

In this sense, our mesh compression system is designed from a very practical point of view: that it should be able to fit into an asset pipeline capable of exporting a full 3D scene to the web. An overview of the pipeline dataflow is show in Figure 1. Figure 2 shows a series of screenshots of the pipeline in action.

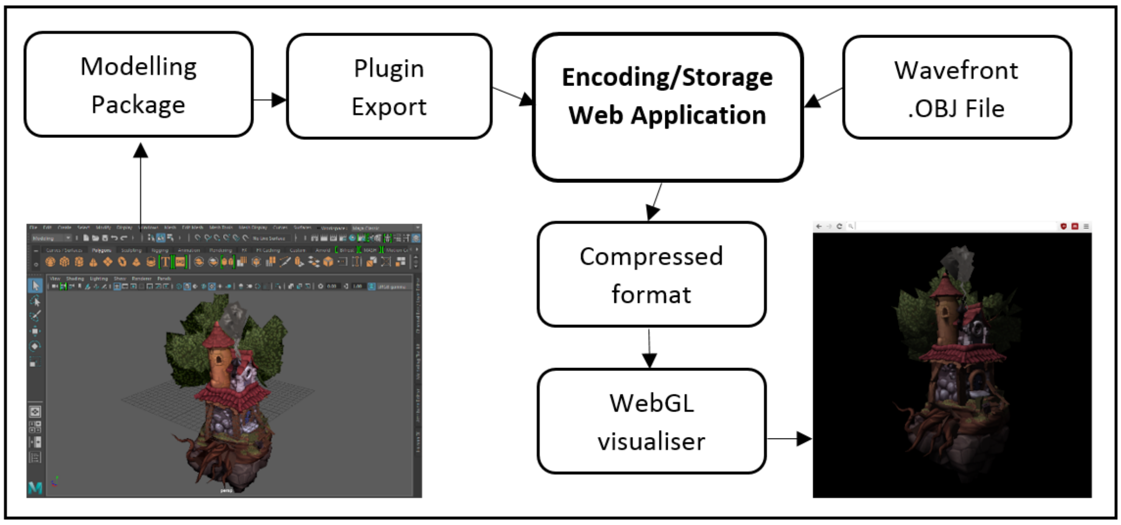
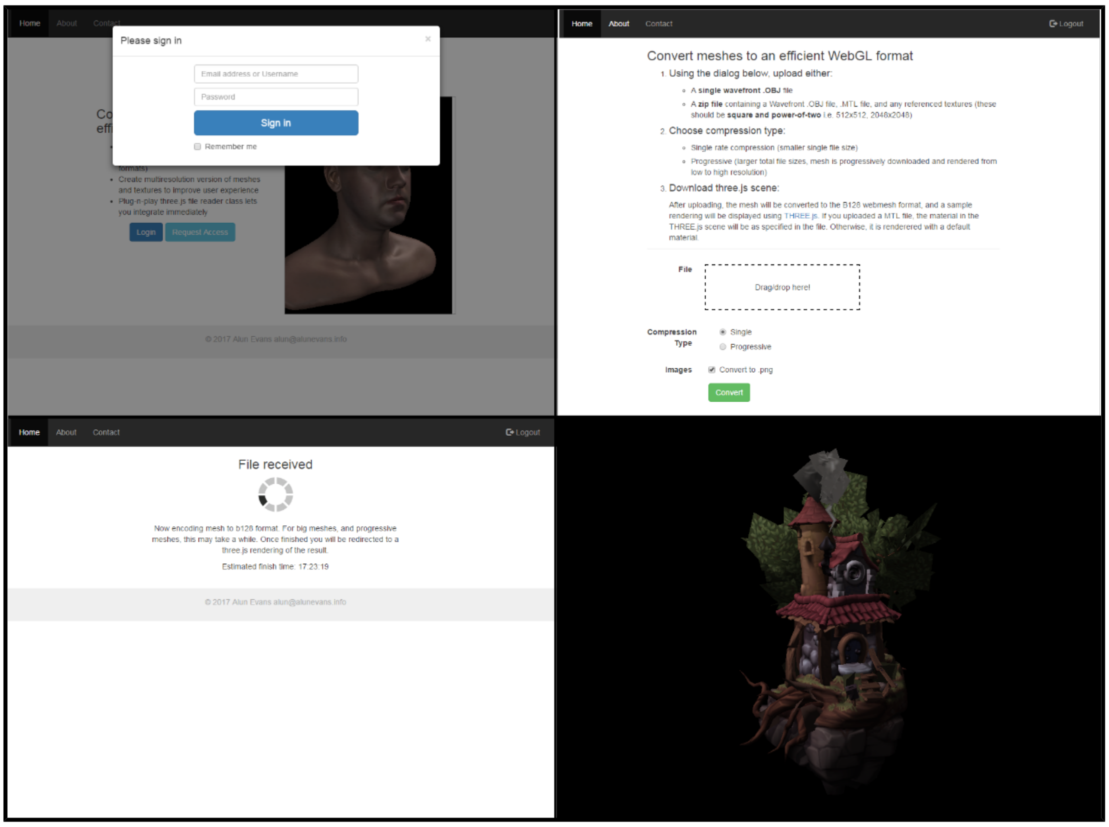


Fig. 1 Overview of asset conversion pipeline. Scenes are either exported from a modelling package using our custom plugin, or loaded directly from a Wavefront .OBJ file. Our web application uploads the content to a server, converts to our compressed format, and renders it using WebGL. [2]



Please sign in

Email address or Username

Password

Sign in

Remember me

formats)
- Create multiresolution version of meshes and textures to improve user experience
- Plug-n-play three.js file reader class lets you integrate immediately

© 2017 Alun Evans alun@alunevans.info

Convert meshes to an efficient WebGL format

1. Using the dialog below, upload either:
   - A **single wavefront .OBJ** file
   - A **zip file** containing a Wavefront .OBJ file, .MTL file, and any referenced textures (these should be **square and power-of-two** i.e. 512x512, 2048x2048)
2. Choose compression type:
   - Single rate compression (smaller single file size)
   - Progressive (larger total file sizes, mesh is progressively downloaded and rendered from low to high resolution)
3. Download three.js scene:

   After uploading, the mesh will be converted to the B128 webmesh format, and a sample rendering will be displayed using THREE.js. If you uploaded a MTL file, the material in the THREE.js scene will be as specified in the file. Otherwise, it is renderered with a default material.

File Drag/drop here!

Compression Type [✓ Single] Progressive

Images [✓ Convert to .png]

File received

Now encoding mesh to b128 format. For big meshes, and progressive meshes, this may take a while. Once finished you will be redirected to a three.js rendering of the result.

Estimated finish time: 17:23:19

© 2017 Alun Evans alun@alunevans.info

Fig. 2 Screenshots of the pipeline in action. Left->Right; Top->Bottom: Initial login page; upload page with drag-drop facility; holding page with finishing time estimation; final browser render.

2 ‘Small House Diorama’ © Glen Fox. Used according to CC BY_NC 4.0 license: https://creativecommons.org/licenses/by-nc/4.0/

4.1 Modelling Package Plugin

To ensure smooth workflow, we have implemented a plugin for three popular modelling tools: Autodesk Maya, Autodesk 3D Studio Max, and Blender. In the first two, the plugin exists as a dashboard button which the user can press, in the latter it is a context menu option when the user right-clicks on the viewport. In each case, the plugin exports the geometry, materials, textures, and, where possible, the lighting and camera settings. The various files involved in this export are archived into a single zip format, and an operating system hook (functional for Windows, Mac and Linux) brings into focus a file explorer window, opened at the folder where this zip file has been saved. This zip file is now ready to be uploaded to the encoding web application (see next section).

4.2 Web Application

The key component of the pipeline is the client-server web application, which is responsible for encoding scenes into our proposed format and storing both original and encoded files on the server. While it is possible to implement the encoding process directly as part of the modelling package plugin, we decided on a more general server-based approach, as this permits encoding of files/scenes created in other packages (for example, meshes output from photogrammetry techniques) for which we have not yet created a plugin. Figure 3 shows an overview of the components of the web application.

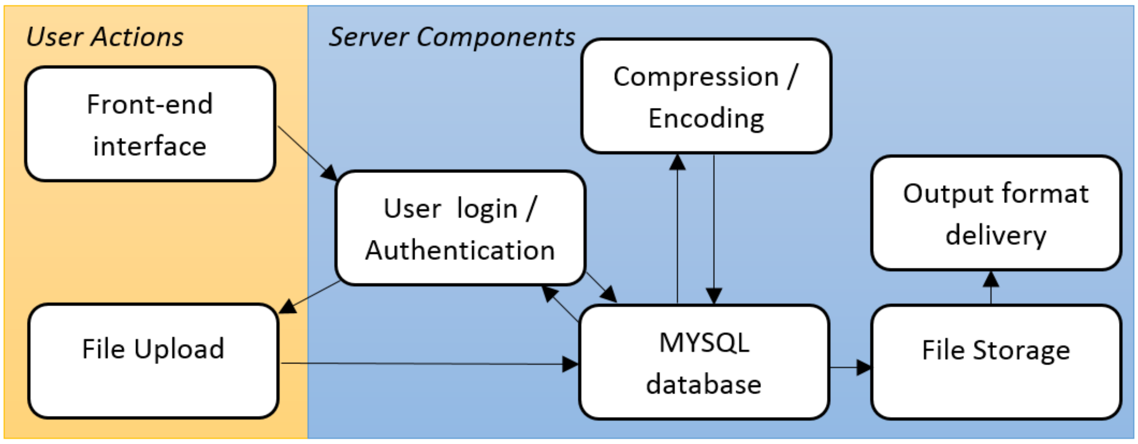


Fig. 3 Overview of the web application. User actions are shown to the left, while the components of the server are shown to the right.

4.2.1 Server Component

We chose to implement the server component as an original PHP application, as this provided us with complete flexibility regarding the structure and implementation. The initial user interaction with the server is via the account system: users must first login to the application before being able to upload files. Once uploaded correctly, the files are stored on the server, and a relevant entry is created in a MYSQL database. We use a concurrent database-filesystem as this permits us to control access to any uploaded data (data security is an extremely important issue in the audiovisual production industry). Our file upload system is implemented in a manner such as to make it highly suitable for this specific task:

- Each uploaded file is stored in its own unique directory (named according to a random 6-character string). This directory name further serves as the scene's unique identifier (uid) for future reference.
- If the file uploaded is a zip file, it is automatically decompressed inside the directory

- A (server-side) extension whitelist is employed such that only files with pre-approved extensions can be uploaded, or unzipped.
- Each user is assigned a quota of storage space; attempting to upload files which surpass this quota results in a displayed error message. Storage space can be managed via a control panel (see below).

4.3.2 Encoding Process

Once uploaded, the server starts the encoding process. The encoder is a compiled C++ binary application which converts input geometry to the format discussed in Section 3 above, and saves material and scene data in a json file. It is called, as a command line application, from the PHP script with the correct parameters for the recently uploaded file. Separately, the PHP process converts texture files (as described below) using the popular imagemagick suite of command line tools[3].

4.3.3 Front-end interface

The front-end interface of the application first requires the user to authenticate via a username and password. Once authenticated, users drag-and-drop or select scene files – these can either be zip files generated from modelling package plugin, or files saved in the Wavefront .obj format. The user can select whether they wish single-rate or progressive encoding, before clicking a button to upload the file. Once uploaded, the server begins the encoding process as described above; as this may take some time (particularly for complex scenes, see results below), the front-end page changes to display an estimated finish time. To manage space, the server features a storage space quota system, accessed through a dashboard interface that allows users to manage and delete previously uploaded scenes.

4.3 Multiresolution textures

During the creation of 3D assets, texture data is frequently stored in a lossless format, such as Targa or TIFF. Such formats are not natively readable by web-browsers. Thus, once uploaded the server, the web application converts all non-web-safe image formats to the web-safe Portable Network Graphics (.png) format (we choose png as opposed to jpg due to its support for transparency).

When encoding a scene for progressive rendering, the web-application also saves lower resolution versions of each of the texture files (default 50% resolution). These lower-resolution files are used with the lower-resolution version of the mesh which are initially downloaded during the progressive visualization.

4.4 Sample three.js application

Once the encoding process has finished, the web-page redirects to an interactive WebGL rendering of the encoded scene, created using three.js. From a code perspective, the scene is left intentionally simple, to best demonstrate to other developers how to parse and use the scene in its new, encoded format. The user can download the source code for the renderer, the encoded files, and the originally uploaded files.

5. Results

In this section, we present quantitative results of our mesh compression method, comparing against the state of the art. We then present quantitative evaluation of the benefits of using the entire pipeline.

3 https://www.imagemagick.org/

5.1 Metrics

The results of the mesh compression are discussed along the following parameters, which interplay, and should be balanced:

- *File size*: the size of the compressed file is clearly an important factor for web 3D applications, as larger files take longer to download.
- *Decompression time*: very important for interactivity; we measure times with which the entire mesh/scene is decompressed. *Compression* time is of lower priority; this said, it should be in the order of seconds (as opposed to tens of seconds) from a usability point of view [9].
- *Visual quality*: Our approach uses lossy compression, especially related to the normals, thus visual quality needs to be addressed.

The benefit of using a complete asset pipeline, as proposed in this paper, can be measured by comparing the time taken to use the pipeline, compared to the time it takes an experienced user to carry out each individual step of the pipeline in isolation, one after the other.

5.2 Experimental Setting: Application, Browser, Hardware and Meshes

5.2.1 Dataset

The detailed results in this paper (for single rate compression) are presented using three test meshes (see Table 1). With these three meshes, we provide a detailed (Tables 2-5) breakdown of the effects of the different elements of data compression which we employ.

Table 1 contains a summary of the data for these meshes, including the number of vertices and faces of each, and the amount of memory each mesh occupies (its binary size – this data is calculated from the fact that the vertex data is stored in 4-byte float format, and the face indices as 4-byte integers). Each of the meshes features index buffers; each vertex has a position and a normal attribute, but no colour information or texture coordinates (although, as mentioned above, our system supports them fully).

While these three meshes are used to test our work in detail, in order to obtain an integral and more extensive testing we use the McGuire Computer Graphics Archive [18], a collection of 34 meshes, to compare our work and the best performing single-rate encoding technique of the state-of-the-art (Google Draco). Not all meshes in the archive were used, either because DRACO was unable to read them, or their size makes them unsuitable for real-time rendering (the unused meshes are noted as such in Annex 1).

Table 1: Table with data regarding the meshes used in the primary evaluation. The binary size refers to the size of the mesh when stored in binary .ply format, including vertex normal, but without any other attributes (such as colours or texture coordinates).

Mesh	**#Vertices**	**# Faces**	**Binary Size**
Happy Buddha	540K	1.1M	27.8MB
Chinese Dragon	430K	870K	21.8MB
Stanford Bunny	35K	70K	1.8MB

5.2.1 Experimental Setting

The results presented here are measured using a very simple WebGL application with minimal HTML and CSS. The browser used was Google Chrome 46, although the system has been successfully tested to work with all major browsers, including mobile ones such as Mobile Safari.

The results from other research were obtained by scraping HTML/JavaScript code from their publicly available test sites, and re-hosting on our own server, thus ensuring a constant test environment and consistent results. The decompression times were measured on a 2.5GHz Intel Core i7 with a Geforce 650M graphics card with 2GB of VRAM, and using the native javascript performance.now() function.

5.3 Experimental Results

This section presents results demonstrating the effectiveness of the novel compression steps introduced by our compression algorithm (Tables 2 and 3). For comparative evaluation against the state of the art, see the following section (5.4).

5.3.1 Effect of paired-triangle index buffer compression

Table 2 shows that our novel paired-triangle index buffer method (described in Section 3.1.3) provides a considerable reduction in mesh size, over the basic method employed by WebGL-Loader and others. The paired-triangle method allows improved compression of the index-buffer, as it attempts to encode paired triangles using four indices instead of 6. The table shows that the technique provides a mean 15.3% reduction in mesh size.

Table 2: Index Buffer Compression: Effect of using paired-triangle index compression (using base compression + tipsify reordering + per-axis quantisation + delta-encoded attributes + high-watermark indices as a starting point). Sizes in MB.

Model	**Base (WebGL-Loader)**	**Paired-triangle technique (OUR)**
Buddha	6.7	5.8
Dragon	5.3	4.6
Bunny	0.458	0.397

5.3.2 Normal Vector Compression and Visual Quality

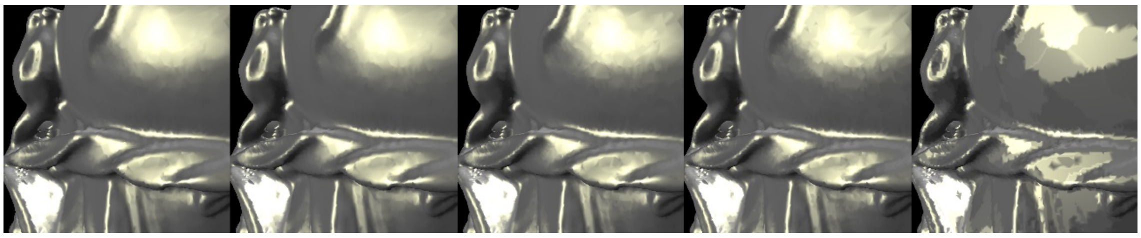


Fig. 4   Effects of normal encoding. The figure shows five close up renders of the buddha model using different normal encoding methods. (L -> R): Original normal, octahedral encoding, 8192 points on a unit sphere, 4096 Fibonnaci points, 256 Fibonacci points.

Figure 4 shows a pictorial comparison of the quality of the different normal compression techniques (see Sections 3.1.1 and 3.1.2) compared to the original (uncompressed) values. This figure should be viewed in conjunction with Table 3, which shows file sizes for the different techniques. The figure, which provides a closeup of critical zones for different techniques, shows that the octahedral method provides results which are nearly identical to the original, demonstrating its suitability for purpose. 4096 Fibonacci normals, which compress to a similar level to octahedral normals show a slight degradation, thus can be discarded. Indeed even when there are 8192 points on the sphere, the rendering quality is not as good as that of the octahedral method. With 256 Fibonacci points the quality is clearly unacceptable, but as it does provide better compression results, it seems suitable for a stage of a progressive rendering approach. In terms of quality, it is perhaps unsurprising that the octahedral method provides such results, as it means essentially discretising the 3-component normal to 16-bit precision (8-bits for each axis of the octahedron technique), and this provides much greater precision than the fibonacci technique.

Table 3 shows the resulting file sizes for compression using the different normal encoding methods discussed in this paper. When viewing this table in conjunction with Figure 4, we can see that the octahedral normal encoding method provides the best balance of quality vs compression. The results also highlight the principal disadvantage of the Fibonacci normal transmission: fewer than 13-bits precision (i.e. 8192 possible normal vectors) provides unacceptable quality, and greater than 10-bit precision does not compress as well as the octahedral normal encoding. Nevertheless, the excellent compression provided by 8-bit Fibonacci precision is useful when generating lower resolution versions of the mesh. Based on this this result, we use the Fibonacci method to encode normals at lower resolutions of our progressive method.

Table 3: Normal encoding: Effect on filesize of the use of different methods of encoding the vertex normals. Sizes in MB unless otherwise indicated. The numbers after the Fibonacci entries refer to the number of points generated on the unit sphere (i.e. maximum number of possible normals).

Model	11-bit Quantisation	Octahedral	Fibonacci 4096	Fibonacci 256
Buddha	5.8	4.9	4.5	1.3
Dragon	4.6	3.9	3.7	3.5
Bunny	0.397	0.330	0.313	0.294

5.4 Compression Comparative Evaluation

In this section, we evaluate our method against the state of the art in single rate compression (Section 5.4.1) and progressive compression (Section 5.4.2). Note that, for the single rate comparisons, we deactivate the progressive component of our algorithm – at no time is a single rate method evaluated against a progressive method. The quantisation bit-depth (the number of bits used to store each position variable) is set to 11 bits for each method.

5.4.1 Single Rate Techniques

Table 4: Comparison of single-rate compression performance for the happy buddha mesh between our method (using octahedral normal compression), OpenCTM [29], WebGL-loader [13], and Draco [17]. All results are taken after HTTP gzip compression. *10 submeshes, decoded in series.

	OUR (octahedral)	OUR (Fibonacci)	OpenCTM	WebGL Loader	Draco
File Size (MB)	2.7	2.6	3.5	5.1	1.7
Decompression (ms)	300	300	2000	100*	2055
Decomp rate (Mtris/s)	0.33	0.33	2.2	0.11	3.3
Full download @ 8mbps (ms)	3098	2935	5504	5217	3254
Mesh Split	No	No	No	Yes	No

Table 4 shows a comparison of two of our methods (using octahedral or Fibonacci normals) with what we have seen as best previous single rate encoding results. The table shows the overall comparison, according to the key metrics for web compression discussed in Section 5.1: file data size (in megabytes); decompression time (in milliseconds); decompression rate (millions triangles per second); and total time until first appearance (measured from the moment the page is loaded until the first frame of the mesh is rendered, at a clamped bandwidth of 8mbps); and an indication of the whether the method requires splitting the mesh (regardless of WebGL index buffer limitation).

As seen from the table, our methods improve significantly with respect to all metrics over OpenCTM except for mesh splitting.

Our methods are also better than those published for WebGL-Loader. It is worth stating that, since publication, the WebGL-Loader method has undergone improvements, which lead to compressed file sizes whose values are similar to those obtained with our method. Unpublished results for the Happy Buddha mesh, using an experimental version of WebGL-loader, are comparable to ours (2.7Mb for the final file size), yet it still relies on the UTF-8 data encoding and requires splitting of meshes larger than 55,000 vertices.

Table 4 shows that the best performing alternative technique to ours is Google's Draco [17] algorithm. To better evaluate the differences between the two techniques, we have carried out a comparison using a larger dataset (the McGuire 3D dataset, discussed in Section 5.2.1). Table 5 presents the mean values for file size, decompression time in JavaScript, and "time to display" (time taken from the moment the file is requested via HTTP, to the moment it is displayed on screen) at two different bandwidths, 50mbps, and 8mbps.

The results show that, while Draco obtains a 31% smaller file size on average, it is 2.2 times slower in data decompression. This contributes to our technique outperforming Draco in the crucial metric of "time to display", which is what directly affects the user experience. The difference is particularly marked at higher network bandwidths, as the cost of decompression outweighs the time to download the data.

Table 5: Comparison of our technique vs Google Draco [17], using mean values for the meshes of McGuire 3D graphics archive [18]. Full results are shown in Annex 1. The 'better' results for each metric are highlighted in **bold**.

Method	File Size (MB)	Decompression Time (ms)	Time-to-display @ 50mbps	Time-to-display @ 8 mbps
OUR	1.13	**281**	**915**	**2485**
Draco	**0.86**	626	1473	2997

To better illustrate how this affects the overall performance, Figure 5 presents a graph which compares our method with Draco, at different bandwidths. The data shows the overall time for a single test model (the Happy Buddha model) to render, from the moment the initial HTTP request is made. The data agree with those presented in Table 5 in that, at 8 mbps, our technique is marginally faster, yet as bandwidth increases, our method is considerably faster. This is because our method's decompression time (as seen in table 5) is 2.2 times faster than Google's technique, meaning that, as bandwidth increases, our method provides increasingly better results.

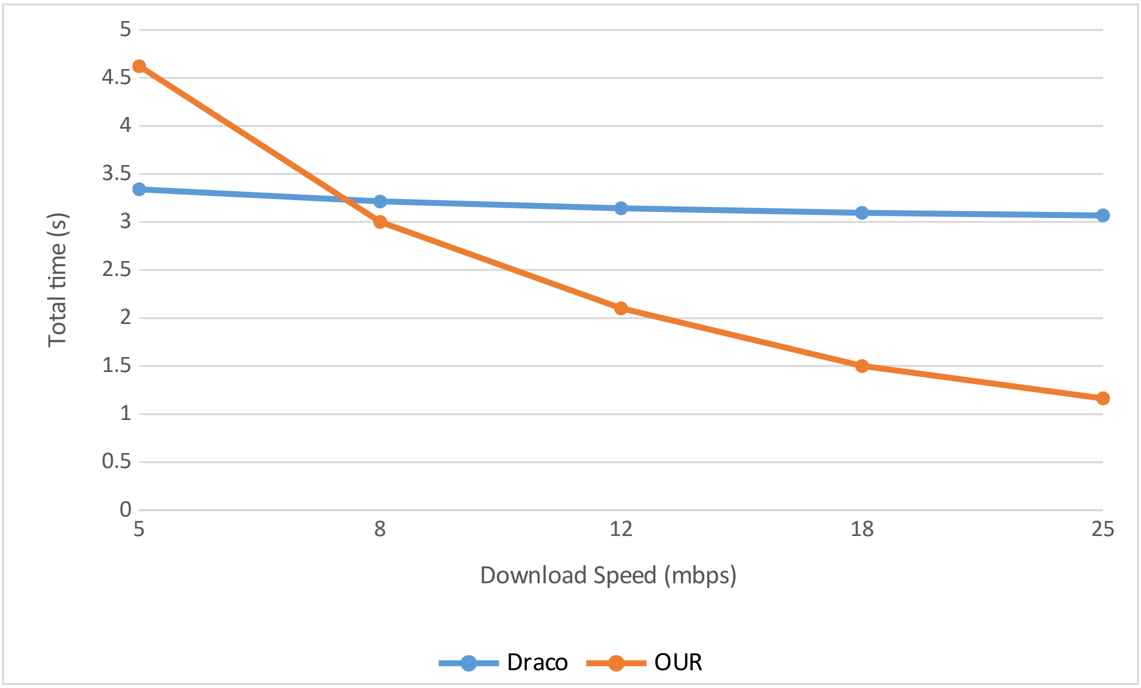


Fig. 5 Google Draco [17] vs our method (using octahedral normals), for the Happy Buddha model.

Perhaps the biggest limitation of both Draco and WebGL-loader is their lack of a progressive approach, which seem necessary for most interactive 3D web graphics applications. Our method allows for immediate transformation to progressive. Let us turn to evaluate this now.

5.4.2 Progressive Methods

Table 6: Comparison of the time taken to visualise the Happy Buddha mesh between our method, 3DHOP [16] and the POP Buffer [14].

	OUR	3DHOP	POP
Size (MB)	3.5	3.9	15
3Mbps	0.2/12.3	0.2/12.5	0.8/46.1
5Mbps	0.1/8.2	0.1/7.3	0.2/26.0
8Mbps	0.1/7.3	0.1/7.6	0.2/18.1

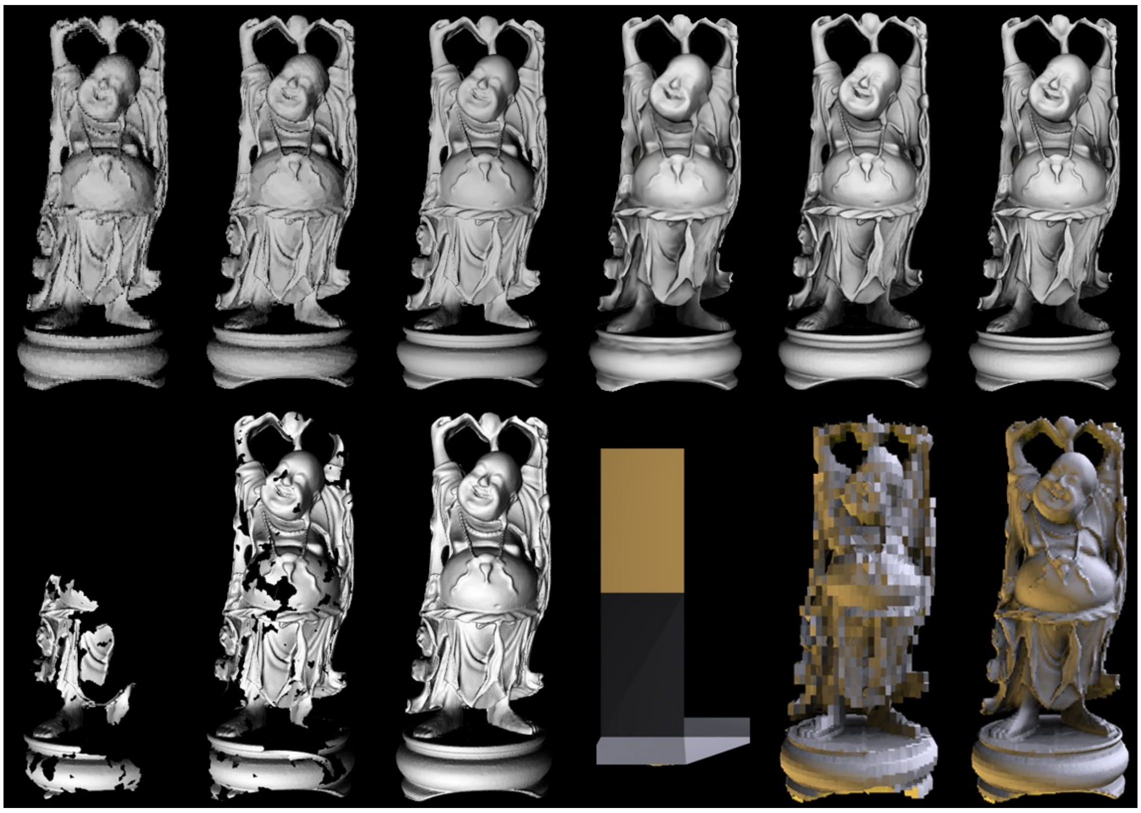


Fig. 6 Progressive loading of the untextured Happy Buddha model, a comparison of four different techniques downloaded at the same bandwidth. Each corner of the figure shows the results from (clockwise from top-left) our method, 3DHOP[16], POP buffer [14], WebGL-loader[13]

Table 6 shows a quantitative comparison of our (progressive method) results with those of 3DHOP and the POP buffer, the best ones according to our review. Here we use a simple metric, that of time taken to display the model at different bandwidths. We present two results in each table cell, the first is the time taken to display the initial, low-resolution model, the second is the time taken to finish downloading the complete model. In all cases, the low-resolution model is displayed in a fully interactive 3D scene from the moment it appears, and a user interface informs the user that further data is being downloaded, until the final mesh appears. The results show that POP buffer performs four times worse compared to our method and 3DHOP, although the visual effect of increasing resolution is very elegant. The 3DHOP method is also very elegant, and the results are similar to our method. As mentioned previously, 3DHOP currently only supports vertex colours, while ours supports a variety of materials by implementing a multi-resolution material system.

Figure 6 shows a visual comparison between our method, 3DHOP, the POP buffer, and WebGL-Loader In the figure, the three meshes shown (from left to right, for each technique) represent the rendering of model after 1 second, 5 seconds, 12 seconds (time from page refresh). In all but one case (the POP buffer), the right-most mesh represents the final mesh render. The visual results of the three progressive techniques, 3DHOP, POP buffer and ours, are quite similar in quality. While WebGL-Loader is not a progressive technique, the mesh is loaded in chunks (reflecting the limit of the UTF-8 format), and we include it to show the benefits of progressive rendering.

With respect to both 3DHOP and POP buffer, our progressive method improves compression performance at lower resolution through mapping vertex normals to spherical Fibonacci points.

Figure 7 shows three screenshots of our progressive method loading the Lee Perry Smith head model (from the McGuire 3D dataset [18]), complete with multiresolution diffuse and normal maps. The use of Fibonacci normals (lowest resolution image, far left) clearly affects render quality, yet the lower file size ensure that this model is displayed on the screen in a matter of milliseconds, while the higher resolution data is downloaded.

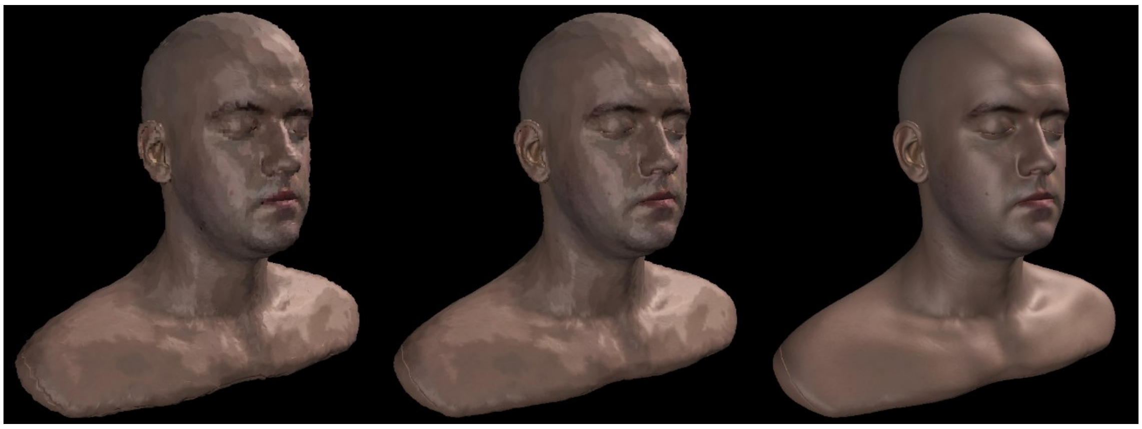


Fig. 7 Progressive loading of the Lee Perry Smith head model. Left -> Right: lowest resolution mesh, displayed first, mid-resolution mesh (with low resolution texture maps), final model and textures.

5.5 Online Pipeline

In this section, we compare the time taken to export the scene to a three.js web application, with and without our pipeline. Accurate quantification of this time without using the pipeline can be difficult as it depends on the size and complexity of the scene. To address this, we tested our pipeline with three very different scenes, described in Table 7 – the 'Batman' scene being the most complex, as it features several highly detailed meshes and 14 high-resolution textures. Figure 8 shows images of the Cartoon House and Batman scenes, rendered in the browser. All our measurements in this table were made with Autodesk Maya (both with and without our pipeline) and Adobe Photoshop for texture conversion (without pipeline). To simplify the comparison, we used the non-progressive version of our pipeline.

Table 8 shows a mean speed increase of 37% when using the pipeline. The standard deviation is quite high (34%), as there is clearly a large difference in performance. This is clearly linked to whether the scene features texture data, as the manual process of scaling textures is very time consuming.

Table 7: Scenes used to measure performance of pipeline

Scene	**# meshes in scene**	**Total # faces**	**Size as .obj**	**Size in .b128 (our format)**	**Textures**
Happy Buddha	1	1.1M	80MB	2.7MB	None
Small House	27	13,000	1.26MB	88.5KB	4 (albedo); 2048x2048; .tga format
Batman	4	2.5M	261MB	4.3MB	14 (albedo, normal maps, specular maps); 4096x4096; .tga format

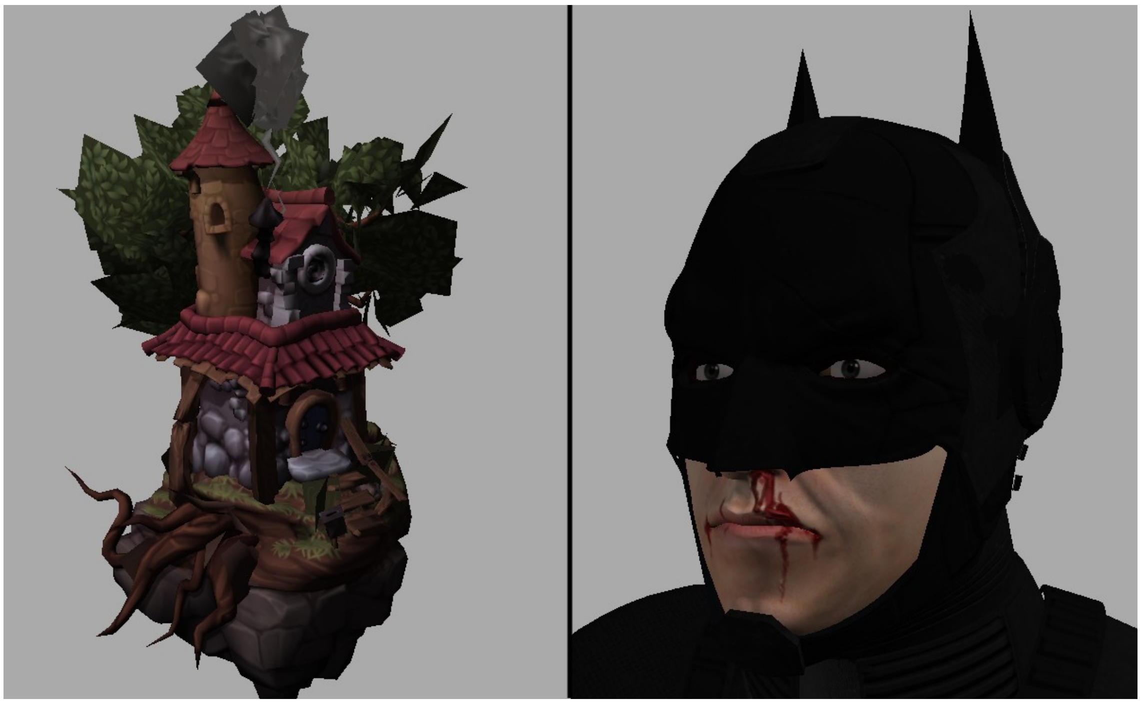


Fig. 8 The Small House (left) and Batman[4] (right) scenes which were used to test the pipeline.

4 The Batman © Dani Candil 2017. Used with permission.

Table 8: Comparison of times taken to export three different scenes to a pre-coded three.js web 3D application. All units (other than percentages) are seconds.

	Happy Buddha		**Cartoon House**		**Batman**	
Step	**Without pipeline**	**With pipeline**	**Without pipeline**	**With pipeline**	**Without pipeline**	**With pipeline**
Exporting scene to disk	36	65*	13	10*	112	350*
Conversion of multiresolution textures to web format	-	-	120	-	420	-
Compression for upload	44	-	3	-	251	-
Uploading to server @3mbps	66	66	10	10	693	693
Conversion to b128 format	-	77	-	30	-	330
Modification of threejs scene code	60	-	60	-	120	-
Full scene download (including textures) @ 8mbps	16	3	19	10	608	75
Total time	**222**	**211**	**226**	**60**	**2204**	**1448**
Percentage improvement	**5%**		**73%**		**33%**	
Mean percentage improvement	**37%** (standard deviation = 34%)					

* ('With pipeline' results include data compression – it is built into the exporter plugin)

6. Discussion

In this paper, we have argued about the importance of a holistic view of exporting of 3D content for the web. The principal concepts are:

- Exporting a scene for web use presents challenges which differ from those of the non-web case, specifically in the formats and mesh compression techniques required. Mesh compression, while clearly important, is just one part of complete 3D package to be processed by the asset pipeline. Texture conversion, multiresolution issues, online-conversion systems are all factors that influence the validity and usefulness of the pipeline.
- In addition, the type of mesh compression used should reflect the fact that bandwidth speeds are increasing at a faster rate than browser processing power. Thus, aggressive, probability based compression techniques lose out in terms of pure performance to those techniques that prioritise rapid decompression.

We have provided quantitative and qualitative results to demonstrate that our compression methods, both single rate and progressive, improve upon existing results. Moreover, the solution we present in this paper concerns an end-to-end asset pipeline, which allows users to export directly from their modelling package to a ready-to-use interactive WebGL rendering of the scene; we demonstrate that our pipeline provides clear speed benefits over a non-pipeline approach.

6.1 Improvement over the state-of-the-art

In this paper, we have provided comparative performance evaluation against three other techniques (Table 4) and extended this with detailed comparative evaluation against the best-performing of these techniques (Table 5, Figure 5, Annex 1). Comparative evaluation of our progressive method, against two methods from the state of the art, is present in Table 6 and Figure 6. Qualitative evaluation of the use of the pipeline is presented in Table 8. We now discuss this evaluation with reference to the improvement of our work over the state of the art.

In terms of single rate mesh compression (Tables 4 and 5, Figure 5), for some years WebGL-Loader has been widely accepted as the best method for the web, and since its original release has been improving. Yet it still relies on the UTF-8 file format, which, as discussed above, has several drawbacks. The base-128 varint format we use requires some bit-shifting at JavaScript level to decompress, but the total decompression rate is still greater than 3M triangles/second, that of WebGL-Loader. The most important advantage of the base-128 format is that is permits a 29-bit index buffer, which avoids the need to split the mesh, which simplifies greatly any further mesh manipulation or deformation in the 3D application (for example, for animation), and therefore is more suitable to more complex interactive 3D scenes, such as games.

Google recent release of Draco seems to indicate that it has shifted away from purely web-based mesh compression to a more unified approach. This has paid dividends as far as the compressed file sizes achievable with Draco are very impressive. However, the evaluation presented in this paper supports one of the central theses of our work: that decompression speed is more important than absolute file compression. Our experiments (Table 5, Figure 5, Annex 1) show that the unavoidable decompression time of Draco's JavaScript implementation results in a slower user experience at at all but very modest bandwidths.

Indeed, at all bandwidths, it can be argued that a progressive approach (Table 6, Figure 6) provides a more optimal user experience [9], as the user obtains an interactive rendering of the scene in very little time. While our approach is perhaps not quite as visually attractive as that of the POP buffer - which constantly updates the mesh in small increments, the results suggest that this does not merit the large overhead in file size, which leads to a longer time to download the final mesh. Our solution presents only three levels of detail; and this reduced amount of data, coupled with our compression of attributes, provides a faster experience for the user.

The 3DHOP method achieves a speed similar to ours - supporting a good user experience - by reparameterizing the mesh according to a patch-based method. This also provides an elegant and significantly faster update solution. Yet, 3DHOP does not currently support texture coordinates, and the recalculation of texture coordinates for the reparameterised mesh does not seem to be straightforward. Furthermore, any mesh deformation, such as blend shapes or bone-weight animation, would require considerable recalculation for any patch-based method. Thus, 3DHOP, conceived for digital heritage, seems unsuitable for digital entertainment applications.

6.2 Future Work

Our implementation of animation using mesh compression is at its initial stages, but it is worth discussing these efforts. The direct mapping between original and quantized vertices of our method should make any deformation relatively straightforward to encode accurately. Indeed, blendshape, or morph target, animation, should be relatively straightforward to add to our single rate compression technique, as it is simply a case of delta encoding blend offsets for those vertices which require it. For our progressive technique, it is slightly more complex, as we would need to maintain a mapping between the vertices across resolutions, in order to ensure that the correct vertices in the lower resolution meshes are morphed. Skeletal animation offers further challenges. While skeleton structure and bone transformations can be encoded fairly easily, ensuring that the weights for each bone are accurately represented for lower resolution meshes is an interesting problem.

In terms of pure compression, we intend to research re-triangulation algorithms that prioritise the longest possible chains of sequential triangles, which clearly benefit our paired-triangle index buffer compression. We plan to implement prediction techniques (such as parallelogram prediction) to improve geometry compression, and investigate methods to store multiple texture coordinates per vertex, to reduce the compression overhead at texture seams.

A potential weakpoint of our pipeline is that it does not support view-dependent downloading of higher resolution data. A further potential improvement, then, would be to incorporate spatial partitioning and download higher resolution data only when the viewport is focused on a particular area of the mesh. This would enable our system to deal with meshes and scenes containing many millions of polygons.

An interesting comparison of our work is

Finally, we will implement the encoding algorithm in JavaScript, allowing users to benefit not only from faster download of their assets from the web, but potentially from faster upload too.

References

1. de Paiva Guimarães M, Dias DRC, Mota JH, et al (2016) Immersive and interactive virtual reality applications based on 3D web browsers. Multimed Tools Appl 1–15. doi: 10.1007/s11042-016-4256-7
2. Ioannidou A, Apostolidis E, Collyda C, Mezaris V (2015) A web-based tool for fast instance-level labeling of videos and the creation of spatiotemporal media fragments. Multimed Tools Appl 76:1735–1774. doi: 10.1007/s11042-015-3125-0
3. Quax P, Liesenborgs J, Barzan A, et al (2016) Remote rendering solutions using web technologies. Multimed Tools Appl 75:4383–4410. doi: 10.1007/s11042-015-2481-0
4. Evans A, Romeo M, Bahrehmand A, et al (2014) 3D graphics on the web: A survey. Comput Graph 41:43–61. doi: 10.1016/j.cag.2014.02.002
5. Cabello R, Ulicny B, Koo J (2010) Three.JS. http://threejs.org/. Accessed 12 Sep 2017
6. Autodesk (2017) Autodesk. https://www.autodesk.com/. Accessed 14 Sep 2017
7. McIntyre L (2010) Building a DAM, one brick at a time. J Digit Asset Manag 6:344–348. doi: 10.1057/dam.2010.41
8. Nielsen J (1994) Usability engineering. Elsevier

9. Nielsen J (1999) Designing Web Usability. New Riders

10. Akenine-Möller T, Haines E, Hoffman N (2008) Real-time rendering. CRC Press

11. Gregory J (2009) Game engine architecture. CRC Press

12. Maglo A, Lavoué G, Dupont F, Hudelot C (2015) 3D Mesh Compression: Survey, Comparisons, and Emerging Trends. ACM Comput Surv 47:44. doi: 10.1145/2693443

13. Chun W (2012) WebGL models: End-to-End. In: Cozzi P, Riccio C (eds) OpenGL Insights. CRC Press, p 431

14. Limper M, Jung Y, Behr J, Alexa M (2013) The POP Buffer: Rapid Progressive Clustering by Geometry Quantization. Comput Graph Forum 32:197–206. doi: 10.1111/cgf.12227

15. Limper M, Wagner S, Stein C, et al (2013) Fast delivery of 3D web content: a case study. In: Proc. 18th Int. Conf. 3D Web Technol. ACM, pp 11–17

16. Potenziani M, Callieri M, Dellepiane M, et al (2015) 3DHOP: 3D Heritage Online Presenter. Comput Graph 52:129–141. doi: 10.1016/j.cag.2015.07.001

17. Google Google Open Source Blog: Introducing Draco: compression for 3D graphics. https://opensource.googleblog.com/2017/01/introducing-draco-compression-for-3d.html. Accessed 1 Feb 2017

18. McGuire M (2017) Computer Graphics Archive. http://casual-effects.com/data/index.html. Accessed 12 Sep 2017

19. Sons K, Klein F, Rubinstein D, et al (2010) XML3D. In: Proc. 15th Int. Conf. Web 3D Technol. - Web3D '10. ACM Press, New York, New York, USA, p 175

20. Behr J, Eschler P, Jung Y, Zöllner M (2009) X3DOM: a DOM-based HTML5/X3D integration model. Proc 14th Int Conf 3D Web Technol 127–136.

21. Chim J, Lau RWH, Leong HV, Si A (2003) CyberWalk: a web-based distributed virtual walkthrough environment. IEEE Trans Multimed 5:503–515. doi: 10.1109/TMM.2003.819094

22. Liu Y, Nie L, Liu L, Rosenblum DS (2016) From action to activity: Sensor-based activity recognition. Neurocomputing 181:108–115. doi: 10.1016/j.neucom.2015.08.096

23. Evans A, Agenjo J, Blat J (2014) Web-based visualisation of on-set point cloud data. In: Proc. 11th Eur. Conf. Vis. Media Prod. ACM, p 10

24. Blat J, Evans A, Kim H, et al (2016) Big Data Analysis for Media Production. Proc IEEE 104:2085–2113. doi: 10.1109/JPROC.2015.2496111

25. Kim H, Evans A, Blat J, Hilton A (2017) Multi-modal Visual Data Registration for Web-based Visualisation in Media Production. IEEE Trans Circuits Syst Video Technol 1–1. doi: 10.1109/TCSVT.2016.2642825

26. Cavalcanti MGP, Rocha SS, Vannier MW (2004) Craniofacial measurements based on 3D-CT volume rendering: Implications for clinical applications. Dentomaxillofacial Radiol 33:170–176. doi: 10.1259/dmfr/13603271

27. Zampoglou M, Kapetanakis K, Stamoulias A, et al (2016) Adaptive streaming of complex Web 3D scenes based on the MPEG-DASH standard. Multimed Tools Appl 1–24. doi: 10.1007/s11042-016-4255-8

28. Behr J, Jung Y, Franke T, Sturm T (2012) Using images and explicit binary container for efficient and incremental delivery of declarative 3D scenes on the web. In: Proc. 17th Int. Int. Conf. 3D Web Technol. pp 17–25

29. Geelnard M (2010) OpenCTM, the Open Compressed Triangle Mesh file format. http://openctm.sourceforge.net/. Accessed 12 Sep 2017

30. Mellado J (2014) js-openctm. https://github.com/jcmellado/js-openctm. Accessed 15 Mar 2017

31. Ponchio F, Dellepiane M (2015) Fast decompression for web-based view-dependent 3D rendering. In: Proc. 20th Int. Conf. 3D Web Technol. pp 199–207

32. Blume A, Chun W, Kogan D, et al (2011) Google body: 3d human anatomy in the browser. In: ACM SIGGRAPH 2011 Talks. ACM, p 19

33. Forsyth T (2006) Linear-speed vertex cache optimisation. https://tomforsyth1000.github.io/papers/fast_vert_cache_opt.html. Accessed 12 Sep 2017

34. Hoppe H (1996) Progressive meshes. In: Proc. 23rd Annu. Conf. Comput. Graph. Interact. Tech. SIGGRAPH. pp 99–108

35. Lavoué G, Chevalier L, Dupont F (2013) Streaming Compressed 3D Data on the Web using JavaScript and WebGL. In: Proc. 18th Int. Conf. 3D Web Technol. pp 19–27

36. Alliez P, Desbrun M (2001) Progressive compression for lossless transmission of triangle meshes. In: Proc. 28th Annu. Conf. Comput. Graph. Interact. Tech. ACM, pp 195–202

37. Praun E, Hoppe H (2003) Spherical parametrization and remeshing. ACM Trans Graph 22:340–349. doi: 10.1145/882262.882274

38. Cigolle ZH, Donow S, Evangelakos D (2014) A survey of efficient representations for independent unit vectors. J Comput Graph Tech 3:1–30.

39. Hannay JH, Nye JF (2004) Fibonacci numerical integration on a sphere. J Phys A Math Gen 37:11591.

40. Swinbank R, James Purser R (2006) Fibonacci grids: A novel approach to global modelling. Q J R Meteorol Soc 132:1769–1793.

41. Brauchart JS, Dick J (2012) Quasi--Monte Carlo rules for numerical integration over the unit sphere. Numer Math 121:473–502.

42. Marques R, Bouville C, Ribardière M, et al (2013) Spherical Fibonacci point sets for illumination integrals. Comput Graph Forum 32:134–143. doi: 10.1111/cgf.12190

43. González Á (2010) Measurement of areas on a sphere using Fibonacci and latitude--longitude lattices. Math Geosci 42:49–64. doi: 10.1007/s11004-009-9257-x

44. Keinert B, Innmann M, Sänger M, Stamminger M (2015) Spherical fibonacci mapping. ACM Trans Graph 34:193. doi: 10.1145/2816795.2818131

45. Sander P V, Nehab D, Barczak J (2007) Fast triangle reordering for vertex locality and reduced overdraw. ACM Trans Graph 26:89. doi: 10.1145/1276377.1276489

46. Agenjo J, Evans A, Blat J (2013) WebGLStudio – a Pipeline for WebGL Scene Creation. In: Proc. 18th Int. Conf. 3D Web Technol. Pp 79–82

47. Liu Y, Zhang L, Nie L, Yan Y, Rosenblum D (2016) Fortune Teller: Predicting Your Career Path. In: Proc 30th AAAI Conference on Artificial Intelligence. Pp 201-207

48. Liu L, Cheng L, Liu Y, Jia Y, Rosenblum D (2016) Recognizing Complex Activities by a Probabilistic Interval-Based Model. In: Proc of the 30th AAAI Conference on Artificial Intelligence. Pp 1266-1272

49. Liu Y, Nie L, Han L, Zhang L, Rosenblum D (2015) Action2Activity: Recognizing Complex Activities from Sensor Data. In: Proc. 24th International Joint Conference on Artificial Intelligence. Pp 1617-1623

Annex 1: Single rate compression using McGuire 3D dataset

This table shows the full data for each of the meshes [18] used to compare our single-rate compression algorithm vs that of Google Draco [17]. This data is used to provide the mean values in Table 5 above. *Compressed Size* refers to the size of the file downloaded from the server, after gzip compression. *Decompression Time* is the number of milliseconds taken to convert the raw compressed data until it is added to the Three.js scene. *Time to display* refers to the milliseconds taken to download and display a model (measured from the moment the HTTP request is made, to the moment the uncompressed mesh is added to the Three.JS scene. The table contains values at clamped bandwidths of 50 mbps and 8 mbps. Some meshes in the dataset were not used, the reasons are included in the table.

Model	.obj size (MB)	OUR				DRACO			
		Compressed Size (MB)	Decompression Time (ms)	Time to display @50 mbps	Time to display @8 mbps	Compressed Size (MB)	Decompression Time (ms)	Time to display @50 mbps	Time to display @8 mbps
Happy Buddha	90	2.7	307	1412	3098	1.7	2055	3189	3254
Stanford Bunny	13.4	0.641	211	793	1427	0.328	312	915	1087
Cornell Box	Not used: geometry too simple								
Clouds	Not used: geometry not suitable								
Conference Room	26.6	0.406	130	544	947	0.109	378	1074	1373
Crytek Sponza	20.6	1.1	293	1149	2380	0.72	579	1246	2525
Cube	Not used: geometry too simple								
Dabrovic Sponza	5.5	0.173	107	352	541	0.114	110	835	1442
Chinese Dragon	45.3	2.6	466	1465	5134	1.6	1130	2508	5400
Erato	Not used: DRACO failed to load input mesh								
Fireplace Room	27.3	0.44	153	458	1082	0.916	494	1332	3516
Gallery	106.9	3.6	586	1693	6982	1.2	1319	2398	5335

Hairball	230.5	6.1	913	3253	11732	4.8	3959	6424	9720
Holodeck	0.305	0.024	43	208	204	0.016	23	498	1274
Horse Chestnut Tree	35.7	1.1	423	916	2521	1.9	851	2311	4734
Indonesian Statue	90.1	3.7	543	2333	7172	3.9	1819	3496	9331
Lost Empire	8.3	0.339	472	786	1567	0.662	822	1450	2623
Living Room	Not used: DRACO failed to load input mesh								
LPS Head	1.4	0.103	126	400	513	0.028	33	676	1844
Mori Knob	0.93	0.058	52	202	320	0.032	35	656	1487
Mitsuba Knob	6	0.296	98	480	781	0.124	133	709	1733
Power Plant	Not used: geometry too large for real-time rendering								
Road Bike	Not used: DRACO failed to load input mesh								
Rungholt	Not used: geometry too large for real-time rendering								
Salle de bain	Not used: geometry too large for real-time rendering								
San Miguel	Not used: both DRACO and OUR failed to load input mesh								
Scrub Pine Tree	0.047	0.049	12	120	134	.006	9	528	1270
Serapis Bust	13	0.575	184	727	1458	0.509	341	900	2359
Sibenik Cathedral	5	0.317	380	949	1177	0.164	150	457	1100
Sports Car	16.4	1.1	450	1558	2851	0675	470	1097	1854
Teapot	0.849	0.043	98	258	503	0.035	42	800	1453
Vokselia Spawn	Not used: both DRACO and OUR failed to load input mesh								
White Oak Tree	3.9	0.311	118	451	760	0.286	134	628	1477
Amazon Lumberyard	Not used: geometry too large for real-time rendering								
Mean		**1.130**	**280**	**915**	**2443**	**0.95**	**625**	**1473**	**2**