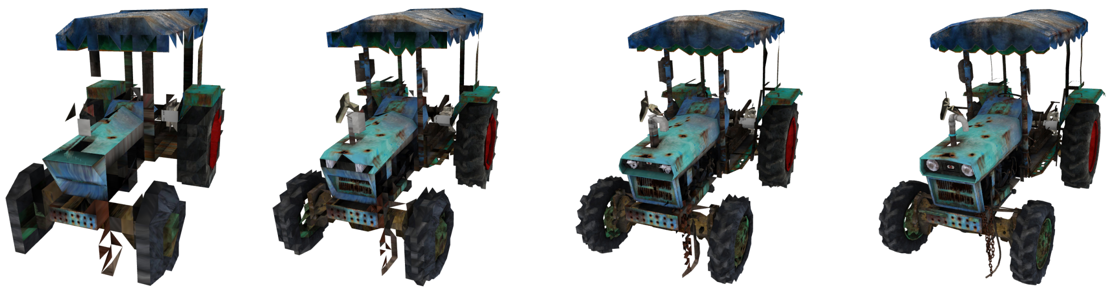
Pacific Graphics 2013
B. Levy, X. Tong, and K. Yin
(Guest Editors)

Volume 32 (2013), Number 7

The POP Buffer: Rapid Progressive Clustering by Geometry Quantization

M. Limper[1,2], Y. Jung[2], J. Behr[2], and M. Alexa[3]

[1] TU Darmstadt, Germany [2] Fraunhofer IGD, Germany [3] TU Berlin, Germany



(a) **4 bit, 5.5% triangles** (b) **5 bit, 16% triangles** (c) **6 bit, 36% triangles** (d) **16 bit, 100% triangles**

Figure 1: *Our fast progressive streaming method completely avoids CPU-based decoding steps, making it very attractive in Web-based and mobile environments. The full range of LOD representations has been created within only 9 ms.*

Abstract

Within this paper, we present a novel, straightforward progressive encoding scheme for general triangle soups, which is particularly well-suited for mobile and Web-based environments due to its minimal requirements on the client's hardware and software. Our rapid encoding method uses a hierarchy of quantization to effectively reorder the original primitive data into several nested levels of detail. The resulting stateless buffer can progressively be transferred as-is to the GPU, where clustering is efficiently performed in parallel during rendering. We combine our approach with a crack-free mesh partitioning scheme to obtain a straightforward method for fast streaming and basic view-dependent LOD control.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Computer Graphics—Three-Dimensional Graphics and Realism—Display Algorithms

1. Introduction

3D Web applications have gained much attention within the past few years. The advent of the first *WebGL* specification enabled a direct integration of hardware-accelerated 3D graphics into standard Web pages without the need for any specific plug-ins, and it resulted in a wide variety of new browser-based 3D graphics APIs [BEJZ09, DBPGS10]. Despite this trend towards high-performance 3D graphics on the Web, the fast and progressive transmission of 3D meshes within Web applications still remains a challenging problem. This might seem surprising, as there has been much work dedicated to the development of Progressive Mesh (PM) compression methods within the past decade, and beyond [PKJK05]. We think that there are several reasons for the fact that such formats have not yet been widely used in the context of 3D Web applications:

- Almost all of the existing PM methods optimize for Rate-Distortion (R-D) performance. However, this aim does not address the crucial tradeoff between compression ratio and *decode time*, which has only been mentioned in a few pioneering PM publications [Hop98, PR00] and the very latest results from the Web3D community [LCD13].

- Many existing PM methods make assumptions about the *topology* of the input mesh, for example that it is a manifold [AD01]. In contrast, a general format must be able to handle any kind of input mesh.
- Fast *encoding* is generally not considered a prior aim at all. Nevertheless, this aspect can become important in some 3D Web scenarios, for example in the context of 3D model community platforms, where servers quickly prepare new assets for transmission and online presentation.

As a consequence of all these points, common Web3D data formats have rather small compression ratios compared to PM methods, but keep encode time and especially decode times as small as possible [LCL10, BJFS12, Chu12]. This ensures an interactive user experience, and it is usually a good choice as long as the available bandwidth is not absolutely minimal (i.e., only a few MBits per second), which would justify advanced compression methods. Because of all these reasons, we argue that a progressive mesh transmission format for the Web must take into account different requirements than past PM algorithms. Those claims are also supported by latest research results from the Web3D community [LCD13, LWS*13].

Within this paper, we present a novel mesh encoding method which can be performed at interactive rates and is able to handle arbitrary triangle soups. It enables fast progressive transmission and basic Level-Of-Detail (LOD) features. We first introduce our algorithm and provide a brief discussion on the geometric properties of the intermediate representations. We then present a mesh partitioning scheme which avoids cracks between partitions with differing resolutions, and we discuss several important aspects like encoding time, rendering performance and memory consumption.

The key aspect of our method is a novel, stateless storage structure, which can be progressively transmitted to the client's GPU. This structure, called the *Progressively Ordered Primitive (POP)* buffer, provides an interlaced transmission of the input model's triangle data, comparable to the progressive Adam7 algorithm used by PNG images on the Web. While our method does not include sophisticated compression capabilities, it is very well-aligned to GPU structures and introduces *zero* CPU-based decode steps on the client side. This is especially crucial if devices require their precious CPU power for other tasks, or if they are simply technically limited in this domain. The approach is therefore particularly well-suited for Web-based environments and mobile clients.

2. Related Work

Quantization and Adaptive Precision. To compress mesh geometry for both, transmission and storage, many approaches employ quantization of vertex positions, as proposed in the pioneering work of Deering [Dee95]. While more sophisticated methods like quantization of spectral coefficients are clearly of superior quality [BCG05], uniform quantization in cartesian space is still the most popular approach in practice [JPP08] – likely because it is fast and simple. In the following, we will use the term *quantization* as a synonym for this quantization method.

Chow [Cho97] observed that integer quantization is similar to snapping vertex positions to a regular grid. He computes the error based on the granularity of a region. However, quantization is simply considered a static pre-processing step. Hao and Varshney [HV01] have shown how the dynamic use of quantized coordinates can speed up 3D transformations. Pool et al. [PLS08] experimentally confirmed these findings and provided a study on depth errors. Still, both approaches are ignoring the fact that many triangles might become degenerate after quantization. They therefore just complement LOD techniques by dynamically reducing the precision of vertex properties, while the POP buffer proposed in this paper inherently combines both approaches.

Purnomo et al. [PBCK05] use quantized vertex attributes for a compact, densely packed storage of mesh data in GPU memory. Decompression is performed inside a vertex shader during rendering. Still, they focus on the off-line creation of a static, simplified and quantized mesh representation, leaving dynamic aspects like LOD management and progressive representation aside.

In addition to storing quantized vertex data in GPU memory, Meyer et al. [MSGS11] also adapt the precision dynamically during runtime in order to reduce memory load. However, it requires costly dynamic updates of single bits for each vertex during runtime. In contrast, the proposed POP method uses a stateless buffer, which is simultaneously used by all LOD representations, and by all instances of a model.

Progressive Mesh Compression. Methods associated with the term *Progressive Meshes (PMs)*, as originally proposed by Hoppe [Hop96], encode mesh data in a compact and progressive structure based on sequential edge collapse and vertex split operators. As the CPU-based processing time of such approaches can become critical during runtime, game developers have early tried to port parts of the technique to the GPU [Sva99]. The focus of latter work is primarily shifted towards the optimization of R-D performance (see e.g. the survey of Peng et al. [PKJK05]), aiming at compact representations for transmission and mostly leaving the aspects of decode time and LOD rendering completely aside [PK05, ALAK11, LLD12]. Hu et al. [HSH09] presented a GPU-based rendering algorithms for PMs, which partially parallelizes the original method of Hoppe by using geometry shaders. Unfortunately, such advanced GPU programming features are not available in most mobile and Web-based graphics APIs, such as *OpenGL ES* and *WebGL*. In addition, a problem inherent to the basic design of all PM algorithms is the need to store and manage the connectivity information for each instance of a mesh separately. In contrast, our stateless POP buffer does not need to be modified at all, once it has been uploaded to GPU memory.

Discrete LOD and Vertex Clustering. As an alternative to PMs, *discrete LOD* methods completely avoid changes of the mesh data on the GPU. Several pre-computed versions of a mesh are used to represent different levels of detail. This also enables the use of multiple instances without additional memory consumption [LWC*02, Wil11]. The method of Sander el al. [SM05] allows for smooth transitions between LOD representations without popping artifacts. Still, it does not provide a truly progressive data structure, since the representations are still completely disjoint in memory. In contrast, the proposed POP buffer represents several LOD representations in a nested manner, which enables progressive transmission and avoids additional memory overhead.

While a wide variety of mesh simplification methods has been proposed in the past, including e.g. methods based on error-controlled edge contraction [GH97], this classes of algorithms are not really related to our clustering method. Therefore, we refer the interested reader to the still very good and detailed overview in the book of Luebke et al. [LWC*02]. The original *Vertex Clustering* approach, proposed by Rossignac and Borrel [RB92], groups vertices into uniform grid cells by checking their truncated coordinates. Vertices within the same cell are then collapsed to a single *representative vertex*, which could consider importance weights. After this first step, polygons that are degenerate get filtered out, resulting in a static, simplified mesh representation. Extensions to the original algorithm have been proposed by several authors, improving the quality of the results [LT97], enabling dynamic, view-dependent clustering [LE97] or out-of-core processing, without [Lin00] and with help of modern GPU features [DT07].

Schmalstieg and Schaufler [SS97] achieve progressive refinements by simply updating vertex indices within the indexed triangle list whenever the LOD changes. However, this introduces additional processing load. Furthermore, sharing the same triangle buffer for rendering multiple instances with different LOD becomes impractical. Willmott [Wil11] improves the result of the clustering process through several criteria improve the preservation of shape, thin features and attribute discontinuities, while still performing simplifications at interactive rates. However, his method does not enable progressive transmission, since it still creates several, disjoint LOD representations.

3. The POP Buffer

The method proposed within this paper builds on the most widely used geometry representation in modern rendering pipelines, which consists of index buffers and vertex buffers [SNB07]. Within this section, we describe how nesting and reordering of triangle data is realized within the stateless Vertex Buffer Objects (VBOs), and how the proposed reordering scheme realizes a straightforward structure for progressive streaming and basic LOD control.

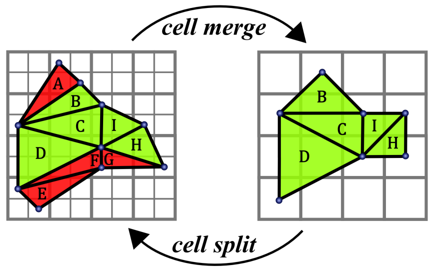


Figure 2: *Switching the grid resolution. Triangles marked in red become degenerate at the lower level and can thus be sorted out. Note that the grids are nested, so that degenerate triangles never reappear at lower levels or, conversely, triangles never degenerate at higher levels.*

3.1. Clustering

We assume each 3D model is given as a triangle mesh, with n vertex positions $\{\mathbf{v}_i\}$, with index $i < n$, and triangles $\mathcal{T} \subseteq \{(i,j,k) \mid i,j,k < n\}$. We start to obtain different LOD representations by computing an axis-aligned bounding box, represented by the minimal and maximal corner $\mathbf{b}_{\text{min}}, \mathbf{b}_{\text{max}} \in \mathbb{R}^3$, for the input mesh. Given a maximal number of bits q for the quantization, we transform this box to a uniform grid in $\mathbb{R}^3$ with integer grid points in $\{0,1,\dots,2^q-1\}$ for each coordinate, i.e. the integer lattice $(\mathbb{Z}_{2^q})^3$.

To map the original shape to the grid of integer coordinates, we apply the following transformation to each vertex coordinate

$$w_{i_c} = \left\lfloor \frac{2^q-1}{b_{\text{max}_c} - b_{\text{min}_c}} (v_{i_c} - b_{\text{min}_c}) + \frac{1}{2} \right\rfloor, \qquad (1)$$

where c denotes the index of coordinate direction. We note that all w_{i_c} are integers in the range $\{0,1,\dots,2^q-1\}$, as desired.

Let the quantization level be denoted $l \leq q$. Our main idea for vertex clustering is to only use the l most significant bits of w_{i_c}, which corresponds to using a reduced uniform grid with integers in the range $\{0,1,\dots,2^l-1\}$. We employ a truncation function $\tau_l(n) = \lfloor n/2^{q-l} \rfloor \cdot 2^{q-l}$, extracting the l most significant bits from a positive integer value n, which can also be implemented using simple bit operations. Based on the truncated integers, the inverse of the bounding box transformation becomes

$$v_{i_c} = \frac{b_{\text{max}_c} - b_{\text{min}_c}}{2^l} w_{i_c} + b_{\text{min}_c}. \qquad (2)$$

Using the truncation function, we are able to easily modify the uniform grid resolution in both directions: truncating one bit less than before is equal to doubling the grid resolution, and vice versa. We can therefore refer to these two operations as *cell merge* and *cell split*, as illustrated in Fig. 2.

3.2. Nesting

We make the following observations: if two points in space $\mathbf{p}, \mathbf{q}$ are mapped to identical points at level l' they necessarily share the same l' most significant bits. Consequently, they also have the same $k < l'$ most significant bits and are also mapped to the same point for all levels $k \leq l'$. Conversely, if they are mapped to different points for a level l'', they differ in their l'' most significant bits – and are mapped to different points for all levels $k \geq l''$.

This observation can be extended to edges and triangles. If the two endpoints of an edge in the triangulation are mapped to the same grid point, the edge is degenerate. If this happens at level l', then this is true for all levels $k \leq l'$; conversely, if the edge has non-zero length at level l'', this is true for all levels $k \geq l''$.

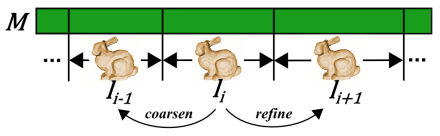
A triangle becomes degenerate once one of its edges is degenerate. For each triangle with index t, we denote the *smallest* level at which it becomes non-degenerate l_t. Since each triangle is degenerate for all levels $k < l_t$, and non-degenerate for all levels $k \geq l_t$, the levels l_t form equivalence classes over the set of triangles. Elements in a class form a set $\mathcal{Q}_l = \{t \mid l = l_t\}$. The nesting property makes identifying the non-degenerate triangles required at a certain level l particularly easy: $\cup_{k \leq l} \mathcal{Q}_k$.

3.3. Reordering

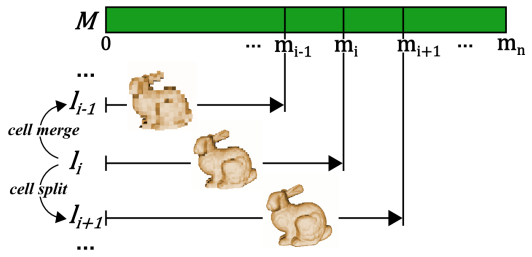
We call the level l_t for triangle t the *popup level* as, intuitively, the triangle appears at this level as the model is refined. Now we sort the triangles according to their popup levels. This results in one reordered sequence of the original triangles, which we call the *Progressively Ordered Primitive (POP)* buffer.

Discrete sorting can be efficiently performed in $O(n)$ operations (where n is the number of triangles), exploiting that the maximum number of equivalence classes is q. The whole sorting procedure simply reduces to creating containers for each level $k \leq q$ and then concatenating the containers.

Fig. 3 illustrates the POP buffer and compares it to the approach of Sander and Mitchell [SM05], where several static LOD representations are stored disjointly in memory. Since each detail level of the POP buffer reuses all data of lower detail levels, progressive loading becomes trivial: everything we need to do to refine our model is to push additional triangle data at the back of our buffer on the GPU. Furthermore, switching the LOD can be realized by adjusting a single parameter of the corresponding draw call, which just specifies



(a) Sander and Mitchell [SM05]



(b) POP Buffer

Figure 3: *The POP buffer in GPU memory, compared to the approach of Sander and Mitchell. Their method stores several LOD representations in disjoint subsections of a mesh data buffer M. In contrast, our approach reorders mesh data in such a way that the m_i elements of each buffer are fully contained within the m_{i+1} elements of the succeeding buffer.*

the amount of rendered primitives from the beginning of the buffer.

It is worth noting that the vertex and triangle data in each set $\mathcal{Q}_l$ can be freely sorted, according to the need of the application. Yet, sorting across the boundaries of sets is impossible. This limitation can result in reduced locality of the triangles in memory. We discuss the effects on framerate in Sec. 7.3. The practical aspects of progressive transmission and basic LOD management are discussed within the following sections 4 and 5.

4. Progressive Transmission

The greatest advantage of the proposed POP buffer is that it can be used for streaming applications in a very straightforward way: incoming vertices and triangles can simply be pushed to the back of the corresponding buffers.

At this point, the question arises how refinement of the quantization scheme is realized. One possibility would be to always explicitly update the quantized positions of all vertices in GPU memory, as soon as data from a new precision level is available. In that case, we would always only send the new bits for existing vertex positions, and all the currently used bits for new vertex positions. Nevertheless, this requires additional processing of incoming data, and additional GPU memory transfer. Such steps can be quite time-consuming, especially for larger models [MSGS11]. The situation is even worse if client devices with limited CPU power are

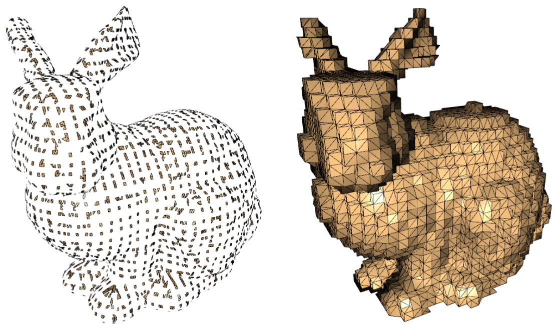


Figure 4: *An intermediate stage of interlaced triangle data transmission. Left: Raw triangle data for detail level $l = 5$, without vertex clustering. Right: Same data, with clustering applied during rendering.*

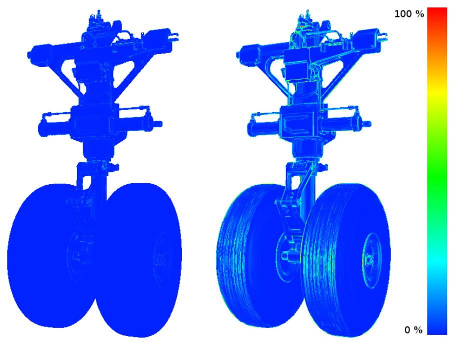


Figure 5: *Image-space error for level l according the bound provided by Eq. 5 (left) and a coarser level $l-3$ (right).*

used, and we also don't want our Web application to block user interaction during the decoding process (or to rely on multi-threading).

To overcome this limitations, we chose to always transmit the full-precision vertex positions, and to perform the quantization on-the-fly in a vertex shader during rendering. Obviously, this leaves some bits unused during early stages of transmission, but we found that the drawbacks of this method are clearly outweighed by its advantages, which are as follows:

- CPU-based decoding steps are completely avoided.
- GPU memory traffic is kept minimal.
- The POP buffer structures in GPU memory are *stateless*.

The last point has several interesting implications, especially for fast LOD selection and instanced rendering (see Section 5).

As can be seen in Fig. 4, the amount of vertices which are shared among the triangles is relatively small in the beginning, since the interlaced transmission scheme tends to spread non-degenerate triangles within each level over the mesh. Nevertheless, the fact that we do not explicitly merge collapsed vertices of the intermediate stages of the model has the great advantage that we do not need to manipulate the geometry or connectivity data at all, once it has been downloaded.

5. Dynamic LOD Control

Within this section, we describe how we select a matching LOD during runtime by using a bound on the geometric error, depending on the distance of each part to the view plane. Given the error bound, we explain how we avoid cracks along the partition's boundaries when rendering different sub-meshes of a large triangle mesh with individual LOD.

5.1. Error Estimation

We know that, at level l, we have dismissed $q-l$ bits of the representation of each vertex or, in other words, we have lumped all vertices in a box with diameter $\frac{\|\mathbf{b}_{\max}-\mathbf{b}_{\min}\|}{2^l}$ into one position. Because we choose the center of this box as the vertex position, the error at level l is bounded from above by

$$\varepsilon_l = \frac{\|\mathbf{b}_{\max}-\mathbf{b}_{\min}\|}{2^{l+1}}. \tag{3}$$

Given this bound on the size of the error in world coordinates, we transform it to screen space, following the derivation of Hao and Varshney [HV01]. We assume quadratic pixels, an aspect ratio of one, a viewport of dimensions $w \times h$, and field of view θ. We find the approximate size of one pixel projected into world coordinates at distance d to be

$$\eta = \frac{2d\tan(\theta/2)}{h}. \tag{4}$$

If we wish to hide geometric errors, we need to make sure that they are smaller than one pixel, i.e.

$$l > \left\lceil \log_2 \frac{\|\mathbf{b}_{\max}-\mathbf{b}_{\min}\|}{\eta} \right\rceil - 1. \tag{5}$$

By this choice of level, there is no need for blending vertices at the transition between levels, while still avoiding popping artifacts.

Nevertheless, smaller shading errors may be visible, even with a guaranteed sub-pixel geometric error. The reason for this lies in the shifted vertex positions: although we are using the full-precision (e.g., 16 bit) normal information at each point, the normals are not adapted to fit with the surface normal at the new position of each vertex after quantization.

Fig. 5 shows a comparison of image-space errors for two different quantization levels, revealing that shading errors occur especially at sharp edges. Still, the choice to keep the full-precision normal for each vertex provides a good preservation of discontinuities, especially compared to vertex clustering approaches that are explicitly unifying vertices (see the

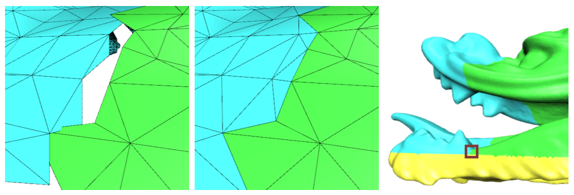


Figure 7: *Closing cracks between sub-meshes. We achieve crack-free borders by sorting a small number of protected vertices to the beginning of the vertex buffer.*

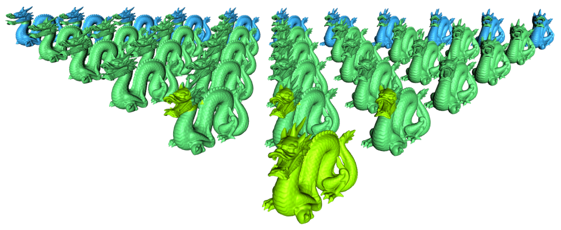


Figure 8: *Instanced rendering, with color-coded LOD. All 36 instances share a single, stateless POP buffer.*

discussion of attribute discontinuities in the work of Willmott [Wil11]).

However, as can be seen in Fig. 6, methods based on error-controlled edge collapses, like the one proposed by Garland and Heckbert [GH97], can provide much better results with the same triangle budget. On densely tessellated flat surfaces, for instance, such algorithms are able to remove many triangles without a visible change, while keeping important details in other regions of the mesh. In contrast, using a fixed quantization grid instead leads to blocky appearance, and small features get lost at lower precision levels. Nevertheless, progressive methods based on edge collapses in turn lack almost all of the advantages of the proposed POP buffer structure (for instance, handling triangle soups, zero decode time and instanced rendering).

As for the normals, we are always using the full-precision texture coordinates. Errors arise in the form of stretched texture regions. However, by keeping the original texture coordinates at the quantized positions, we can already guarantee that texture coordinates are never mistakenly moved after simplification, which is especially important when using a texture atlas. We found that this simple and practical approach provided results of surprisingly good quality. An example is shown in Fig. 1.

5.2. Mesh Partitioning and Crack Prevention

The appropriate LOD depends on the minimal distance d of an object. A large model, however, might span quite a large distance interval. This results in many vertices being quantized to a precision that is significantly higher than necessary. To prevent this, it is common to partition a mesh into several sub-meshes, and then computing an appropriate LOD for each sub-mesh independently. In our setting, we simply compute individual bounding boxes and then use the equations presented previously to bound the error for each sub-mesh.

A general problem that comes with mesh subdivision for LOD management are cracks in an originally closed surface [SM05, MSGS11], occuring when boundary vertices of sub-meshes are mapped to different positions in world coordinates. A common solution is to use the same quantization grid for each sub-mesh, along with roughly equally sized bounding boxes [SM05, LCL10]. However, we still encounter cracks in the mesh if the precision levels of adjacent sub-meshes differ, which can especially be visible when coarser levels are used during streaming.

To overcome this problem, we have decided to simply *protect* the positions of all vertices that are located at the borders the sub-meshes by always using the highest possible quantization level $l = q$. All protected vertices are flagged during preprocessing, and the computation of degenerate triangles consequently considers the high-precision coordinates for these vertices.

To identify protected vertices during rendering, we sort them to the beginning of the vertex buffer and provide their total number as an additional uniform variable in the vertex shader. Each rendered vertex can then simply check this number against its ID (e.g., the value of `gl_VertexID`, if available) to decide whether it should be displayed with the full precision of q bits. Fig. 7 illustrates the difference for a real-world example.

We note that the idea of protected vertices could also be used for other applications, for example preserving feature edges in a mesh. Nevertheless, this also decreases the amount of degenerate triangles at each level, and therefore limits the overall efficiency of the streaming process, which is why we decided to restrict this method to border vertices.

6. Instanced Rendering

A big advantage of the POP buffer is that it supports instanced rendering and streaming (unlike other progressive streaming and LOD techniques [SS97, Sva99, HSH09, MSGS11]). With the proposed method, each instance of a model only needs to manage a single integer value, representing its current level of detail. During rendering, we can then look up the number of primitives for this level and draw the corresponding number of elements from the POP buffer, using matching vertex shader settings for quantization (see Fig. 8 for an example).

It is worth noting that rendering a single geometry from dif-

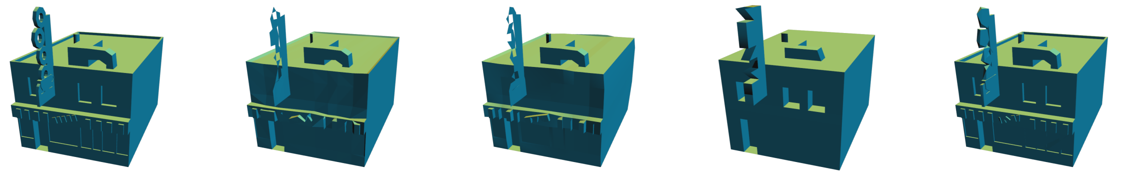


Figure 6: *Simplification to approximately 40% of triangles. (b)–(d) Vertex clustering methods. (e) Quadric-based simplification.*

Model	#Tris	Quant.	Reord.	Total
Building	1,896	0.1	0.2	0.3
Fandisk	12,946	0.3	1.7	2.0
Tractor	49,480	2.0	6.9	8.9
Bunny	69,451	1.9	9.3	11.2
Horse	96,966	2.9	14.5	17.4
Wheel	257,376	9.5	31.9	41.4
Dragon	867,522	23.5	135.9	159.4
Buddha	1,087,716	27.9	176.3	204.2

Table 1: *Encoding time, given in ms, for various models (input data quantization, reordering).*

ferent view points during another rendering pass (for example, for obtaining a shadow map or a picking buffer) can be done in exactly the same way. Many applications, such as collision handling or picking, might therefore greatly benefit from this approach, too.

7. Results

7.1. Encoding

In many scenarios, the encoding of meshes into a specific format has to be performed at interactive rates [Wil11]. An example could be a Web-based platform where all users can upload 3D assets, which are then instantly processed on a server for instant online presentation. The proposed POP buffer structure fits this purpose very well, since even large meshes can be processed within a fraction of a second, as can be seen in Table 1. Larger models have been previously subdivided. Our test machine was a MacBook Pro notebook with an i7 CPU, 2.4 GHz and 4 GB RAM, and we were able to reorganize triangle data into the proposed POP buffer structure at rates of up to 4 million triangles per second, using a sequential, CPU-based implementation.

As the fast labeling approach used by our method (see Section 3.3) is inherently parallel, we think that an optimized (e.g., GPU-based) implementation would achieve even faster run times.

Another interesting topic is how our algorithm relates to *Streaming Meshes*, as proposed by Isenburg and Lindstrom [IL05]. The method maximizes data coherency by reordering the mesh data, which allows mesh processing algorithms to be executed on out-of-core data volumes, in a *sliding window* fashion. The approach of reordering mesh data is quite similar to our method, but both algorithms rely on different criteria for reordering. While the final POP structure itself is therefore not compatible with their mesh format, we note that our encoding algorithm could also operate on a Streaming Mesh, in order to perform an efficient out-of-core construction of our proposed POP buffer structure.

7.2. Streaming

As a consequence of the interlaced triangle data transmission, the amount of new vertices is relatively high in the early levels, as also illustrated in Fig. 9. Fortunately, this drawback is compensated by the relatively small amount of triangles within those levels, and it is only valid for indexed rendering. Fig. 1 as well as the accompanying video demonstrate that a first impression of the shape is already available for a small fraction of the total data.

As can be seen in the rightmost chart of Fig. 9, the geometrical error vanishes quickly, since we are doubling the precision for each incoming batch of triangle data. Note that this chart does not represent an R-D curve, since it is independent from any encoding or compression scheme, which could be used at the cost of additional decode steps.

With the proposed approach, the time needed to *decode* data is always zero, therefore the time needed to *download* the levels is crucial. As the maximum precision level, we found $q = 16$ bits to provide a sufficient quality, therefore the bunny model, for example, has an uncompressed size of $34,834 \times (3 \times 2) = 209,004$ bytes for the vertex positions. It furthermore needs $69,451 \times (3 \times 2) = 416,706$ bytes for the connectivity, if 16 bit indices are used (like it is for example mandatory when using WebGL). In sum, the bunny mesh can hence be represented by $625,710$ bytes, and by some additional metadata (like the number of triangles within each level), which can be sent separately and has neglectable size.

Assuming that, for example, a common DSL connection with a bandwidth of 16,000 kbit/s is used, this means that the bunny mesh can be completely downloaded within 0.31 seconds. For any compression method, this means that it

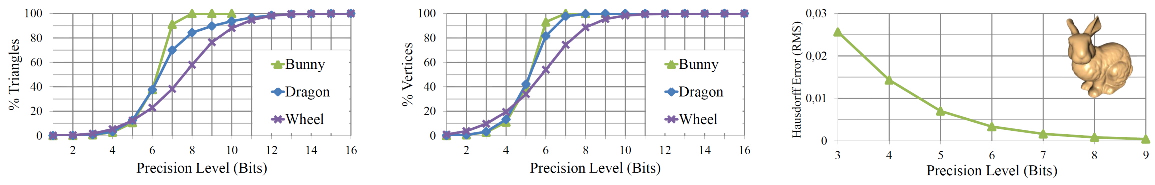


Figure 9: *The two leftmost charts depict the total amount of used triangles and used vertices at each precision level for different models. The rightmost chart illustrates the Hausdorff error for several levels of the quantized bunny model.*

has to be able to decompress at least 221,991 triangles/s in order to deliver the full mesh at the same time as our uncompressed approach. Most popular PM approaches, if they have investigated decompression speed, have reported significantly slower decompression times (e.g., Alliez and Desbrun [AD01] reported 5,000 triangles/s). The latest PM compression results we could find, provided by Maglo et al. [MCAH12], also only reported a decompression rate of 122,000 triangles/s, using an i7 CPU and 2.8 GHz.

As can be seen, even in the raw, uncompressed format, our method is able to deliver the triangle data in a progressive manner within an acceptable amount of time. This is especially true if mobile client devices are used, where the decompression performance is usually expected to be much worse than for desktop machines [LCL10, LWS*13].

7.3. Rendering

We have implemented the POP buffer in a lightweight, browser-based render client, solely relying on standard 3D Web technology, such as JavaScript and WebGL. This made it possible to easily test the efficieny of the POP buffer as a basic LOD method on different hardware platforms, including various WebGL-capable mobile devices (see Fig. 10).

Resulting frame rates for different triangle counts are given in Table 2. On all devices, we can ensure smooth user interactions by instantly switching to a lower precision level during camera movements (see the accompanying video for a brief demonstration). As can be seen, the proposed method resulted in a significant speedup in rendering time, compared to regular vertex buffers without any LOD, on all tested platforms. It can furthermore be seen that the speedup proved by our experiments is *not* due to limited fragment shading costs at far distances, as similar results have been obtained for varying precision with a fixed camera position.

One could also suppose that the GPU's ability to filter out degenerate triangles before the fragment processing stage would make our LOD scheme less efficient, or even obsolete. However, we did not measure any significant speedup when using only quantization and always rendering the full buffer. The reason for this is that our application is *vertex bound* (a basic assumption for most LOD methods), and that

Coverage	l	#Tris	PC	iPhone 5	iPad 2	Nexus 7
26.9%	9	913K	158	14	7	1.5
27.1%	7	367K	246	30	16	2.5
27.2%	6–7	294K	257	33	19	3
27.4%	6	141K	313	40	33	5

Coverage	l	#Tris	PC	iPhone 5	iPad 2	Nexus 7
26.9%	9	913K	158	14	7	1.5
4.0%	7–8	367K	248	30	16	2.5
0.5%	6	141K	315	40	33	5
0.3%	5	56K	321	40	34	8

Table 2: *Rendering times for the Happy Buddha model on several devices, measured in frames per second on a* 512×512 *viewport. The first column indicates the viewport coverage of the model. Top: varying precision at a static camera distance. Bottom: adaptive precision at varying camera distances.*

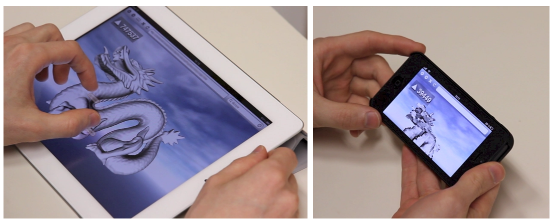


Figure 10: *Due to its lightweight, straightforward design, our progressive streaming and LOD approach works perfectly with mobile devices (here: iPad2 and iPhone5).*

degenerate triangles can only be identified *after* the vertex processing stage.

As already noted, our reordering method can be seen as an interlacing scheme, spreading the triangles of each refinement level over the whole mesh (see Fig. 4). This can potentially have a negative impact on the *average cache miss ratio* (ACMR) during rendering, and therefore also on the overall render performance. Optimizing for example the full Horse model with the *Tipsify* method [SNB07], we get an ACMR of 0.66. In contrast, all we can do in the context of the POP

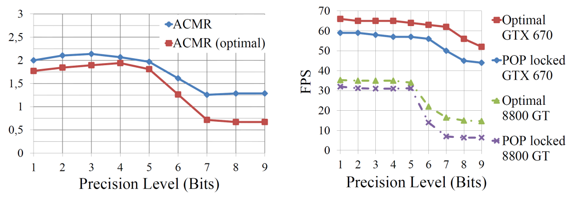


Figure 11: *Left: Cache miss rate for different levels of the Stanford Bunny. Right: Render performance, compared with cache-optimized data, using 401 instances of the model.*

buffer is optimizing the different buffer segments, resulting in a higher ACMR from 2.15 to 1.31 for the several levels. Fig. 11 shows the ACMR for all levels of the Bunny model, reference values have been obtained by optimizing the entire snapshot of the model at each level separately.

In the theoretical case, render performance can be linearly dependent on the ACMR [SNB07]. Modern GPUs, however, process batches of vertex data in parallel and are less sensitive to data locality [Kil08]. We see this effect in our framerate measurements: the total reduction of the vertex count during rendering was found to be far more important than the influence of caching mechanisms. The performance loss due to increased cache miss rate is illustrated in Fig. 11. The *locked* version of our POP buffer was using fixed precision levels (instead of true view-dependent LOD) to focus solely on cache miss performance during the comparison. As can bee seen, our findings indicate a loss of performance due reduced data locality of less than 20% on a modern GTX 670 GPU, and which is far better than the implied linear correlation with the ACMR.

8. Conclusion and Future Work

We have shown that a uniform quantization in cartesian space, based on the truncation of integer coordinates, can also be regarded as the application of a simple vertex clustering scheme on a uniform grid. As a consequence, we find that degenerate triangles can be progressively sorted out at lower levels of detail, leading to a compact and progressive mesh representation – the POP buffer. This buffer consumes the same amount of GPU memory as the original mesh data, yet allows for progressive rendering with minimal computational overhead.

In summary, the proposed method has the following benefits:

- As the concept of quantization is independent from any assumptions about topology or manifoldness, the proposed method is able to handle arbitrary triangle soups.
- The mesh representation is obtained by simply reordering the input mesh data according to some straightforward criteria. As a consequence, even large meshes can be automatically converted at interactive rates.
- The resulting LOD representations are *nested*. As a consequence, mesh data can be transmitted over networks in a progressive manner.
- Streaming applications like Web apps can *directly* send downloaded sections of the POP buffer to GPU memory, without any CPU-based decoding steps. This is especially interesting for setups where the client's CPU power is a critical resource.
- The POP buffer is *stateless*, meaning that it is not manipulated *at all* when switching the LOD. To the best of our knowledge, it is the first progressive 3D mesh representation with this unique property, which helps to minimize GPU memory traffic and is also very useful in the context of instanced rendering.
- The POP buffer has exactly the same memory footprint as a regular single-rate buffer.
- The method presented in this paper can be easily implemented, even with WebGL or GPUs that have a very limited feature set, making the integration into existing pipelines very simple and attractive.

Still, the proposed method has some specific drawbacks compared to more specialized approaches:

- PM methods provide advanced data compression capabilities. They deliver superior results if the client has powerful hard- and software for decoding, and if the available bandwidth is rather low. However, PMs can not be efficiently used with multiple instances of a mesh, and adjusting (at least) the connectivity data during LOD management also introduces additional processing overhead .
- Discrete LOD methods provide better visual results and a more sophisticated handling of attributes without using more triangles. Still, they generally consume more memory than the proposed method and need much more time to generate the discrete LOD representations. Furthermore, they do not allow for progressive streaming.
- Since we perform a reordering of the input triangle data, which can only be cache-optimized per LOD section, our method stands in conflict with cache performance optimization schemes. However, our experiments suggest that the loss in cache hit rate is insignificant compared to the speedup of our LOD method, and that optimizations assuming a FIFO cache also become less relevant.
- The proposed method is unable to handle non-rigid mesh animations (e.g., such used for skinned character models).

Overall, we think that our method might be very useful for fast, progressive streaming in Web-based and mobile setups.

In the future, we plan to investigate compression schemes which could be applied on top of our approach. In this context, the relation between decompression speed and available transmission bandwidth seems the most important aspect.

Acknowledgements: We like to thank Andrew Willmott for providing the building model. The tractor model has been kindly provided by Crytek. The fandisk model was taken from the AIM@SHAPE repository. The horse model is courtesy of the Georgia Institute of Technology. Other models are courtesy of the Stanford 3D scanning repository. We finally thank Christian Stein for assisting us to create the demo applications.

References

[AD01] ALLIEZ P., DESBRUN M.: Progressive compression for lossless transmission of triangle meshes. In *Proc. SIGGRAPH* (2001), pp. 195–202. 2, 8

[ALAK11] AHN J.-K., LEE D.-Y., AHN M., KIM C.-S.: R-d optimized progressive compression of 3d meshes using prioritized gate selection and curvature prediction. *Vis. Comput.* (2011), 769–779. 2

[BCG05] BEN-CHEN M., GOTSMAN C.: On the optimality of spectral compression of mesh data. *ACM Trans. Graph. 24*, 1 (2005), 60–80. 2

[BEJZ09] BEHR J., ESCHLER P., JUNG Y., ZÖLLNER M.: X3DOM: a DOM-based HTML5/X3D integration model. In *Proc. Web3D* (2009), pp. 127–135. 1

[BJFS12] BEHR J., JUNG Y., FRANKE T., STURM T.: Using images and explicit binary container for efficient and incremental delivery of declarative 3D scenes on the web. In *Proc. Web3D* (2012), pp. 17–25. 2

[Cho97] CHOW M. M.: Optimized geometry compression for real-time rendering. In *Proc. VIS* (1997), pp. 347–354. 2

[Chu12] CHUN W.: WebGL models: End-to-end. In *OpenGL Insights*. CRC Press, 2012, pp. 431–454. 2

[DBPGS10] DI BENEDETTO M., PONCHIO F., GANOVELLI F., SCOPIGNO R.: SpiderGL: a javascript 3D graphics library for next-generation WWW. In *Proc. Web3D* (2010), pp. 165–174. 1

[Dee95] DEERING M.: Geometry compression. In *Proc. SIGGRAPH* (1995), pp. 13–20. 2

[DT07] DECORO C., TATARCHUK N.: Real-time mesh simplification using the gpu. In *Proc. I3D* (2007), pp. 161–166. 3

[GH97] GARLAND M., HECKBERT P. S.: Surface simplification using quadric error metrics. In *Proc. SIGGRAPH* (1997), pp. 209–216. 3, 6, 7

[Hop96] HOPPE H.: Progressive meshes. In *Proc. SIGGRAPH* (1996), pp. 99–108. 2

[Hop98] HOPPE H.: Efficient implementation of progressive meshes. *Computers & Graphics* (1998), 27–36. 1

[HSH09] HU L., SANDER P. V., HOPPE H.: Parallel view-dependent refinement of progressive meshes. In *Proc. I3D* (2009), pp. 169–176. 2, 6

[HV01] HAO X., VARSHNEY A.: Variable-precision rendering. In *Proc. I3D* (2001), pp. 149–158. 2, 5

[IL05] ISENBURG M., LINDSTROM P.: Streaming meshes. In *Proc. VIS* (2005), pp. 231–238. 7

[JPP08] JOVANOVA B., PREDA M., PRETEUX F.: MPEG4 Part 25: A Generic Model for 3D Graphics Compression. In *Proc. 3DTV-CON* (2008), pp. 101–104. 2

[Kil08] KILGARD M. J.: Modern opengl usage: Using vertex buffer objects well. In *SIGGRAPH ASIA courses (Contributed Chapter)* (2008), pp. 13:1–13:31. 9

[LCD13] LAVOUÉ G., CHEVALIER L., DUPONT F.: Streaming compressed 3D data on the web using JavaScript and WebGL. In *Proc. Web3D* (2013), pp. 19–28. 1, 2

[LCL10] LEE J., CHOE S., LEE S.: Mesh geometry compression for mobile graphics. In *Proc. CCNC* (2010), pp. 301–305. 2, 6, 8

[LE97] LUEBKE D., ERIKSON C.: View-dependent simplification of arbitrary polygonal environments. In *Proc. SIGGRAPH* (1997), pp. 199–208. 3

[Lin00] LINDSTROM P.: Out-of-core simplification of large polygonal models. In *Proc. SIGGRAPH* (2000), pp. 259–262. 3

[LLD12] LEE H., LAVOUÉ G., DUPONT F.: Rate-distortion optimization for progressive compression of 3d mesh with color attributes. *Vis. Comput.* (2012), 137–153. 2

[LT97] LOW K.-L., TAN T.-S.: Model simplification using vertex-clustering. In *Proc. I3D* (1997), pp. 75–81. 3

[LWC*02] LUEBKE D., WATSON B., COHEN J. D., REDDY M., VARSHNEY A.: *Level of Detail for 3D Graphics*. Elsevier Science Inc., 2002. 3

[LWS*13] LIMPER M., WAGNER S., STEIN C., JUNG Y., STORK A.: Fast delivery of 3D web content: A case study. In *Proc. Web3D* (2013), pp. 11–18. 2, 8

[MCAH12] MAGLO A., COURBET C., ALLIEZ P., HUDELOT C.: Progressive compression of manifold polygon meshes. *Comput. Graph.* (2012), 349–359. 8

[MSGS11] MEYER Q., SUSSNER G., GREINER G., STAMMINGER M.: Adaptive level-of-precision for gpu-rendering. In *Proc. VMV* (2011), pp. 169–176. 2, 4, 6

[PBCK05] PURNOMO B., BILODEAU J., COHEN J. D., KUMAR S.: Hardware-compatible vertex compression using quantization and simplification. In *Proc. HWWS* (2005), pp. 53–61. 2

[PK05] PENG J., KUO C.-C. J.: Geometry-guided progressive lossless 3d mesh coding with octree (ot) decomposition. *ACM Trans. Graph. 24*, 3 (2005), 609–616. 2

[PKJK05] PENG J., KIM C.-S., JAY KUO C. C.: Technologies for 3d mesh compression: A survey. *J. Vis. Comun. Image Represent.* (2005), 688–733. 1, 2

[PLS08] POOL J., LASTRA A., SINGH M.: Energy-precision tradeoffs in mobile graphics processing units. In *ICCD* (2008), IEEE, pp. 60–67. 2

[PR00] PAJAROLA R. B., ROSSIGNAC J.: Squeeze: Fast and progressive decompression of triangle meshes. In *Proc. CGI* (2000), pp. 173–182. 1

[RB92] ROSSIGNAC J., BORREL P.: *Multi-resolution 3D approximations for rendering complex scenes*. Tech. rep., 1992. IBM Research Report RC 17697. 3, 7

[SM05] SANDER P. V., MITCHELL J. L.: Progressive buffers: view-dependent geometry and texture lod rendering. In *Proc. SGP* (2005), pp. 129–138. 3, 4, 6

[SNB07] SANDER P. V., NEHAB D., BARCZAK J.: Fast triangle reordering for vertex locality and reduced overdraw. *ACM Trans. Graph. 26*, 3 (2007), 89:1–89:9. 3, 8, 9

[SS97] SCHMALSTIEG D., SCHAUFLER G.: Smooth levels of detail. In *Proc. VRAIS* (1997), pp. 12–19. 3, 6

[Sva99] SVAROVSKY J.: Extreme detail graphics. In *Proc. Game Developers Conference* (1999), pp. 889–904. 2, 6

[Wil11] WILLMOTT A.: Rapid simplification of multi-attribute meshes. In *Proc. HPG* (2011), pp. 151–158. 3, 6, 7