Random Accessible Hierarchical Mesh Compression for Interactive Visualization

Clement Courbet, Celine Hudelot

Abstract
This paper presents a novel algorithm for hierarchical random accessible mesh decompression. Our approach progressively decompresses the requested parts of a mesh without decoding less interesting parts. Previous approaches divided a mesh into independently compressed charts and a base coarse mesh. We propose a novel hierarchical representation of the mesh. We build this representation by using a boundary-based approach to recursively split the mesh in two parts, under the constraint that any of the two resulting submeshes should be reconstructible independently.

In addition to this decomposition technique, we introduce the concepts of opposite vertex and context dependant numbering. This enables us to achieve seemingly better compression ratios than previous work on quad and higher degree polygonal meshes. Our coder uses about 3 bits per polygon for connectivity and 14 bits per vertex for geometry using 12 bits quantification.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations E.4 [Coding and Information Theory]: Data compaction and compression—

1. Introduction
As the precision of numerical simulations and 3D scanning devices increases, the size of 3D meshes increases continuously. Since the available memory, bandwidth and processing power are limited, it is often desirable to shrink the size of these datasets. Therefore, mesh compression has been an active field of research in the last decade. Very efficient algorithms providing single-rate as well as progressive compression of triangular and polygonal meshes have been described in the literature [PKK05, AG03].

More recently, an increased interest has been given to random accessible mesh compression. This technique enables to select the interesting parts of a dataset one wants to visualize. It can be applied to selectively decode an interesting part of a mesh for interactive view-dependent rendering or partial editing. This idea is also particularly interesting for very large meshes which do not fit in core memory. Using our hierarchical random accessible scheme, we are able to load a compressed version of a mesh and selectively decompress a required part while staying within the limits of available core memory.

This paper presents a new algorithm for random accessible mesh decompression, which has the following properties:

Hierarchical: Our method is built on a recursive split of the mesh into two independent parts.

Random Accessible: The recursive partitioning allows the reconstruction of any requested part of the mesh without decoding other, less interesting parts.

Polygonal: We are able to compress meshes with arbitrary polygons instead of only triangles.

Simple: Our scheme is very simple to implement.

2. Related Work
2.1. Mesh Compression
Single Rate: The research in Single Rate Compression of 3D triangular meshes has been very active since 1995, and very good reviews exist on the subject [AG03, PKK05]. Single rate compression targets high compression ratios, typically for storage. These approaches originally aimed at compressing triangular meshes, but they have also been extended to the compression of polygonal meshes of arbitrary degree [JS00, KADS02, LAD02, IA02]. These al-
algorithms are very successful at compressing connectivity, reaching an average cost of approximately 3.5 bits per vertex (bpv) [PPK05]. In comparison, coding the geometry is more problematic. Typical cost is 15 bpv at 12 bits quantization. Various approaches have tried to improve this by using geometry-driven approaches [KG00b,CCMW05], with up to 70% better ratio for spectral-based approaches [KG00a].

**Multiresolution**: Multiresolution or Progressive approaches aim at providing a coarse-to-fine description of the mesh, generally for transmission purposes. Most methods apply to triangular meshes and rely on edge collapse/vertex split operations [Hop96,PR00], although other methods exist [TGH98,AD01]. The edge collapse/vertex split operation has been extended to enable polygonal mesh simplification in [Ram03].

Another class of approaches concentrates on compressing only the geometry [LDW97,KSS00]. They get rid of connectivity by remeshing the object, and provide the best rate/distortion curves and compression ratios. Valette and Prost [VP04] have generalized these methods to irregular meshes, combining good geometry compression ratios with the possibility of keeping the initial connectivity. However, their method is hard to apply to general polygonal meshes as it requires to enumerate all possible decimations of a polygon. To our knowledge, there is currently no generalization to polygonal meshes.

All these methods generally provide progressiveness at the cost of a significant overhead compared to single-rate coders.

### 2.2. Random Accessible Compression

Recently, because of the needs for interactive exploration of massive datasets, random accessibility in compressed meshes has been given increased interest. Progressive coders aim at providing an immediate access to the global shape of the mesh. To access local fine details, the user has to wait for the decompression of the whole model. In contrast to this, random access enables immediate access to any arbitrary part of the mesh at the finest level of details. Various approaches have been described, that we divide into two groups: Block Random Access (BRA) approaches rely on single-rate compression of chunks of the original mesh, while Multiresolution Random Access (MRA) methods use progressive compression.

Among BRA approaches, Choe et al. [CKLS09] first segment a mesh into charts and globally code the wire-net mesh formed by the boundaries of those charts. Then they independently code each chart using the single-rate compressor of [LAD02]. That way, random access is provided in the unit of a chart, while the access to one vertex within it requires the decompression of the whole chart. Recently, Yoon and Lindstrom have presented a similar method [YL07] which groups triangles into clusters, but also preserves the order in which the mesh was initially stored. Thus, they enable random access without sacrificing cache coherence in the case where the access is locally structured. However, the compression rate is lower than [CKLS09] (≈8 bpv for connectivity). One may note that providing progressiveness with these methods is problematic because of the handling of chart boundaries.

On the other hand, some methods (MRA) have tried to combine the benefits of random access and progressiveness. Liu and Zhang [LZ04] extend the approach of [LDW97] to enable random access. Compression ratios are very good, but as their method is based on a remeshing of the input mesh, the connectivity cannot be retrieved. Kim et al. have proposed a multiresolution scheme that enables the compression of the connectivity and geometry of triangular meshes [KCL06]. Their method is based on a transformation [KL01] similar to vertex split. This transformation can be applied on any vertex of the coarser mesh to add a new vertex. This breaks the symmetry of Hoppe’s vertex split [Hop96] which had to be applied in the same order at encoding and decoding times. This way, using an appropriate searching structure, fine-grain random access can be provided. However, the compression ratio of this technique remains limited to about 12 bpv for connectivity and 21 bpv using 12 bits quantization for geometry, and the authors were unable to attain interactive frame rates.

### 3. Proposed Scheme

We propose a third approach to random access, which is based on a hierarchical subdivision of the mesh. This approach is somehow related to block random access. Instead of dividing the initial mesh into \( K \) charts like [CKLS09], we divide it into two balanced parts. These two parts are recursively subdivided until no further subdivision is possible, or when a given chart size is reached. We thus build a tree of subdivisions. Previous approaches [CKLS09,YL07] used an indexing structure (header) to enable individual access to the charts. Using an indexing structure embedded in the tree of subdivisions, we are able to determine the path to follow to decode only the required part of the mesh without decoding other parts of the tree, thus enabling random access. While the previous approaches needed to decompress a coarse mesh for random access to individual charts, our method enables random access directly from the compressed representation, without the need for the initial coarse mesh decompression.

The idea of a recursive chartification of a mesh for compression is not new. [ADS05] use a three-level chartification to compress a mesh. However, their method aims at fast neighborhood access rather than random access.

In this section, we describe how our algorithm builds and encodes the charts tree. Then, in the results section, we give...
3.1. Wires

Let $G$ be a 2-manifold mesh of genus 0. We define a wire as a sequence of connected vertices (i.e. vertices that are joined by an edge) of $G$ such that any vertex appears at most once. A closed wire is a wire whose first and last vertices are connected. A wire that is not closed is an open wire. If the mesh has a boundary, a boundary wire (or simply boundary) is a closed wire containing all the vertices in the boundary (and only them). A cut wire is a wire that joins two vertices in the same boundary wire which are not connected together in this boundary wire (see figure 1).

In addition, if two wires $W_A$ and $W_B$ share one endpoint but have no other vertex in common, we define the wire $W_A + W_B$ as the wire which contains all the vertices of $W_A$ and $W_B$ (and shares one endpoint with $W_A$ in the case there are several candidates, i.e. for a closed wire).

C as well as the indices $i_A$ and $i_B$ of $v_A$ and $v_B$ in $B_G$ are stored in the root node. Then the above process is applied recursively to $G_L$ and $G_R$.

![Figure 2: One step of the algorithm: splitting the mesh $G$ and its boundary. $C$ (green) is the cut wire, $G_L$ and $G_R$ are the two submeshes, and the original boundary $B_G$ is split into $W_A$ (red, dashed and dotted) and $W_B$ (blue, dashed).](image)

Decoding an element of the mesh: Let us suppose that $B_G$ is known. The process begins at the root of the tree: $i_A$, $i_B$ and $C$ define two regions bounded by $W_A + C$ and $W_B + C$.

We choose the region corresponding to the element to be decoded (★), and proceed recursively (figure 3).

![Figure 3: Decoding the element ★: We begin by numbering the root boundary, which has 14 vertices, from 0 to 13. We retrieve $v_A$ and $v_B$ from $i_A = 2$ and $i_B = 9$. We can then rebuild $C$ (here adding 3 vertices). We choose the left region (which contains the ★), and define the new boundary as $W[0,2] + C + W[9,13]$. We proceed recursively, until we hit an unsplittable polygon.](image)

3.2. A Boundary-based Approach to Coding Meshes

To take advantage of good compression ratios brought by wires, we store the mesh $G$ as a tree of wires. The approach is boundary-based: we begin by extracting a boundary wire $B_G$ of $G$ (the case where $G$ has no boundary is addressed in section 3.6). Then we split $G$ into two independent meshes $G_L$ and $G_R$ (see figure 2). It is easy to see that the vertices that belong both to $G_L$ and $G_R$ form a cut wire. Let $C$ be this cut wire, and $v_A$ and $v_B$ the end vertices of the cut wire. Note that $v_A$ and $v_B$ belong to $B_G$. Therefore we can partition $B_G$ in two wires $W_A$ and $W_B$ such that $B_G = W_A + W_B$, $B_G = W_A + C$ and $B_G = W_B + C$.

![Figure 1: An open wire (left, blue), a boundary wire (center, red) and a cut wire (right, green).](image)
the cut wire is straightforward: we apply a simple linear predictor followed by an entropy coding of the residuals. The distribution of $S_C$ is very biased towards zero. Most of the final cuts will have a length of zero. More generally, we can expect the size of a cut to be approximately $\sqrt{n}$, $n$ being the number of vertices in the current mesh. Therefore, the distribution of $S_C$ will be roughly geometric. Thus, $S_C$ is also suitable for entropy coding.

Figure 4: Typical distributions of $S_C$ (top), $\delta_B$ (middle) and $\delta_B$ (bottom).

3.4. Cut Selection

The compression ratio for a given cut depends on the length of the wire, the smoothness of its geometry, the closeness of $v_B$ to $o(v_A)$, and a good renumbering of the boundary vertices for compressing $i_A$. Thus, finding the best cut wire with respect to compression ratio is too slow for practical use. We must find heuristics to provide an efficient cut. A good cut wire should have the following properties:

1. Be short, so that the number of vertices in $C$ is small (to enable better entropy coding).
2. Cut the boundary in two parts as equal as possible, so that $\delta_B$ is biased towards zero (to enable better entropy coding).
3. Be smooth, to provide good correlation between the geometry of adjacent vertices.

In addition, the heuristic should not be too complex to compute, to reduce the time needed for compression.

We use the following heuristic to determine the best cut: For each index $i$ in the boundary, we note $v^i$ the $i$-th vertex, and $o(i)$ the index of $o(v^i)$. For all $i$, we compute the euclidean distance between $v^i$ and $o(v^i)$. Let $i_0$ be the index such that this distance is smallest (figure 5). We then renumber the vertices in the boundary away from $i_0$ in each direction. We denote the original numbering by a superscript and the renumbering by a subscript. Thus $v^i$ becomes $v_0^i$. We then grow $G_R$ and $G_L$ from the newly determined $W_A$ and $W_B$ until all the vertices in $G$ are visited (We call growing the process of augmenting a submesh $G_{sub}$ with the unvisited vertices of $G$ that have neighbours in $G_{sub}$). The resulting cut wire is the shortest path through the vertices that have neighbors in both $G_L$ and $G_R$. If no path was found from $v_0$

**Opposite vertex and context-dependant numbering:**

Due to the decreasing size of the boundary as we get closer to the leaves, the indices decrease accordingly. They also follow a geometric distribution (figure 4, middle), of which we can take advantage with entropy coding. However, we can greatly improve the compression of the indices by introducing two notions that we call **opposite vertex** and **context-dependant numbering**.

We remark that the cuts that are well suited to building a balanced tree, i.e. cuts that split the mesh into submeshes with similar number of vertices, also tend to separate the boundary in a balanced manner. We take advantage of this property. Instead of coding $i_A$ and $i_B$ separately, we code $i_A$ and $\delta_B$, $\delta_B$ is the difference in the position of $v_B$ in the boundary between the actual cut and the balanced cut (that is, the cut that would result in $W_A$ and $W_B$ having the same number of vertices). Given $v_A$, we call the vertex $v_B$ that would cut the boundary in a balanced manner the **opposite vertex** of $v_A$, noted $o(v_A)$. The compression ratio is very dependant on the actual cut. A careful selection of the cuts can lead to entropies of the order of 0.5 bits for $\delta_B$ (figure 4, bottom), versus 2.5 bits for $i_B$.

As we know how to succinctly code $i_B$, most of the indices bit budget is now allocated to coding $i_A$. The only available context data known at the time of decoding is the boundary information. Therefore, a scheme to code $i_A$ should only rely on this information. The numbering of the boundary vertices comes from the context of the parent subdivision in the subdivision tree, and is therefore arbitrary in the current context. Thus, all the values for $i_A$ are equiprobable, preventing us from decreasing the entropy below the geometric distribution of figure 4. Our scheme uses the geometric information of the boundary to renumber the vertices of the boundary in a way such that the probability of $i_A$ is constant anymore, but has a small variance around 0. This way, the entropy of $i_A$ drops from 2.5 to 0.5 bits. We discuss this point in the next paragraph.
to \( v_{k0} \), then we try to find cut wires between \( v_k \) and \( v_{k0} \) with \( k + l \) increasing, until we find a suitable cut.

As we visit vertices further from the original guess (which is balanced) in increasing order, property (2) is verified. We use the smallest geometric distance, therefore the resulting cut is generally small, and thus property (1) is verified. The smoothness of the geometry of the cut wire depends on the regularity (in terms of vertex valence) of the mesh because of the use of the growing algorithm. A submesh will grow faster near vertices that have a lot of neighbours, and more slowly around those that have few neighbours. If most vertices have the same valence, which is the case in general, the growing will be regular, and so property (3) is generally verified. The heuristic is in \( O(N^{3/2}) \) in the worst case, but often becomes \( O(N) \) practically as a path usually exists for the original guess.

**Figure 5:** Finding the basis vertices for renumbering: The opposite vertices with shortest euclidean distance are picked (here 2-9). The resulting renumbering is in green.

3.5. Handling Holes

The leaves of the tree of charts represent the original mesh polygons. To handle holes in the mesh and differentiate a hole from a face, an escape sequence is reserved in the coding of \( S_C \), which is used to indicate a hole, together with a bit that indicates if the hole is on the left or the right child.

3.6. Initial boundary selection

We use as initial boundary wire the largest available boundary. When no boundary exists, we use the vertices of an arbitrary face as boundary. Using the aforementioned heuristic, the first cut splits the mesh in two parts having roughly the same size.

3.7. Meshes of higher genus

Our algorithm is based on the property that a cut wire divides a mesh in two disjoined parts. This property holds only for meshes with genus 0, therefore our algorithm does not work on meshes with handles.

3.8. Random Accessible Memory Layout

If the goal of the algorithm was to achieve the best possible compression, the best way would be to compress the bit stream resulting from the above process using an entropy coding method (e.g. arithmetic coding). However, such a method is unable to provide random accessibility in the bit stream. Our random access scheme is based on the tree of charts representation of the mesh built earlier. The tree is stored in a depth-first manner, and the cut wire data is entropy coded using a scheme which enables individual symbol decoding (such as Huffman Coding). For more efficiency at lower levels, the tree is coded in an autumnal fashion \( [FM86] \). In addition to the cut wire data, we store in each node of the tree the information needed to reach the right child. Because of the depth-first layout, this information is only the size of the left subtree. This information can be stored in an efficient way. Consider a node \( G \) in the tree, let \( D_G \) be the size (in bits) of the tree of root \( G \) and \( D_C \) the size of the cut wire data stored at node \( G \). We know that the left subtree has a size \( D_{G_L} \) which is smaller than \( D_G - D_C \), so only \( \log_2(D_G) \) bits are enough to code \( D_{G_L} \). We can also retrieve the size of the right subtree \( D_{G_R} \) as \( D_G - D_C - D_{G_L} - \log_2(D_C) \). This way, at each node, the left and right children nodes can be decoded independently, by offsetting the memory pointer of 0 bits (left child) or \( D_{G_L} \) bits (right child). A leaf is simply a node of size zero.

4. Results

In this section, we provide results for our compression scheme. First, we evaluate the compression ratios provided by our method on several classical models. Then we show a
practical application of random-accessible compression. We describe an application-specific traversal method suitable for efficient view-dependant rendering of large models.

4.1. Compression Ratio

Table 1 details the compression ratios for our scheme. Compression ratios are better for quadrangular than for triangular meshes, typically 20 versus 27 bits per vertex. Indeed, there are roughly as many cut wires as faces, but there are twice as many polygons in a triangle mesh than in a quad mesh with the same number of vertices. Our coder uses about 3 bits per polygon for connectivity. Hence, the ratios for connectivity are roughly 6 bits per vertex for triangle meshes and 3 bpv for quad meshes.

As we use Huffman coding in our memory layout to enable random access, the results cannot drop below 1 bit. We include corresponding entropies for reference. Clearly, finding a better coding scheme remains an important work.

Table 1: Compression results for various classical models (T denotes models having mostly triangles, Q models having mostly quads). Note that cut wire length and indices are given in bits per cut wire. The total compression ratio is given in bits per vertex and includes the overhead induced by the random accessible memory layout, which is about 7 – 19%. Numbers between parentheses are entropies. Geometry is quantized to 12 bits. The Neptune Model has very irregular connectivity and smooth geometry.

<table>
<thead>
<tr>
<th>Model</th>
<th>#V</th>
<th>cut wire length</th>
<th>indices</th>
<th>geom. (bpv)</th>
<th>total (bpv)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Igea (T)</td>
<td>61k</td>
<td>1.74 (1.05)</td>
<td>1.15 (0.50)</td>
<td>16.09 (16.00)</td>
<td>28.5</td>
</tr>
<tr>
<td>Ramses (Q)</td>
<td>140k</td>
<td>1.85 (1.78)</td>
<td>1.39 (1.16)</td>
<td>13.41 (13.33)</td>
<td>18.4</td>
</tr>
<tr>
<td>Armadillo (T)</td>
<td>170k</td>
<td>1.72 (1.61)</td>
<td>1.04 (0.20)</td>
<td>11.84 (11.74)</td>
<td>24.7</td>
</tr>
<tr>
<td>Buste (T/Q)</td>
<td>241k</td>
<td>1.85 (1.80)</td>
<td>1.70 (1.51)</td>
<td>11.69 (11.51)</td>
<td>20.0</td>
</tr>
<tr>
<td>Eros (T)</td>
<td>416k</td>
<td>1.77 (1.68)</td>
<td>1.14 (0.47)</td>
<td>11.60 (11.53)</td>
<td>22.9</td>
</tr>
<tr>
<td>Neptune (Q)</td>
<td>3.7M</td>
<td>1.93 (1.87)</td>
<td>2.16 (2.10)</td>
<td>5.80 (5.53)</td>
<td>14.22</td>
</tr>
</tbody>
</table>

4.2. View-dependant Rendering

To illustrate the significance of our approach, we implemented as an example a view-dependant rendering framework. This approach is useful when the model is so large that rendering the whole dataset is too long, or even impossible because the model does not fit into main memory. Using the random accessibility provided by our compression method, we can render only the portion of the model which is of interest to the user without decompressing the whole model. Thus, we decrease the time between request and actual display.

View frustum culling [Cla76] is often used to enable view-dependant rendering. The method consists of building a hierarchy of bounding volumes. If a bounding volume does not intersect the view frustum, then all its children will not lie inside the frustum. Else, the hierarchy is searched one level deeper. We can take advantage of the hierarchical representation provided by our method, by slightly modifying the compression process. For each submesh in our hierarchical representation, a bounding sphere is computed. Its center is the center of mass of the boundary, because this is the only information available to the decoder. The radius of this sphere is stored along with the cut wire. It is aggressively quantized to 8 bits. As the radius gets smaller in lower and more populated levels, its entropy is very low, in the order of 1 bit per polygon. The overhead induced by the method is thus very low. At the time of decoding, only a quick frustum/sphere intersection test need to be carried out to decide whether refinement is necessary or if the whole submesh can be discarded. Figure 7 illustrates this technique.

Figure 7: View-dependant rendering using a cone-shaped frustum. The framed image shows what the user sees, while the other shows what is actually decoded.

View-dependant rendering performance: The random access capability of our method enables interactive visualization of large models. As only the portion of the model that falls into the viewport is fully decompressed, the parts of interest can be decompressed and displayed very quickly (typically 3µs per polygon), resulting in interactive frame rates as long as the part of interest is not too big. Table 2 gives the timings and memory footprints for rendering typical images like those on figure 7.
Table 2: Performance results for view-dependant rendering of various classical models. For each model, a part of the mesh was rendered, as in figure 7. The table summarises the compression and random-access decompression times, and the memory footprint used. The decompression time is given in microseconds per polygon, including the overhead of decoding the wires that lie outside the viewport, but were needed for decompression. The memory footprint is the maximum number of uncompressed vertices that may be needed at the same time to render any selected part of a model.

<table>
<thead>
<tr>
<th>Model</th>
<th>comp. time</th>
<th># faces visible</th>
<th>time per polygon</th>
<th>memory footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Igea</td>
<td>4.4 s</td>
<td>30k (12%)</td>
<td>2.63 µs</td>
<td>935</td>
</tr>
<tr>
<td>Ramses</td>
<td>4.3 s</td>
<td>12k (8.5%)</td>
<td>2.93 µs</td>
<td>1508</td>
</tr>
<tr>
<td>Armadillo</td>
<td>7.9 s</td>
<td>50k (15%)</td>
<td>3.27 µs</td>
<td>1075</td>
</tr>
<tr>
<td>Buste</td>
<td>7.5 s</td>
<td>51k (19%)</td>
<td>2.75 µs</td>
<td>1499</td>
</tr>
<tr>
<td>Eros</td>
<td>15 s</td>
<td>125k (15%)</td>
<td>2.19 µs</td>
<td>1377</td>
</tr>
<tr>
<td>Neptune</td>
<td>7 h</td>
<td>43k (1.2%)</td>
<td>2.36 µs</td>
<td>5394</td>
</tr>
</tbody>
</table>

4.3. Comparison with previous approaches:

In [CKLS09], the authors use their algorithm for view-frustum culling. To enable this, they must entirely decode the wire-net mesh to test whether each face intersects the viewport, and decode the corresponding chart if this is the case. As they use the algorithm of [KADS02] to encode the wire-net mesh, the memory footprint needed for their view-dependant rendering method is

\[
m_c(n, K) = O(K + T) = O(K + \frac{n}{K})
\]

where \( n = K \times T \) is the number of vertices in the global mesh, \( K \) the number of charts and \( T \) the number of vertices per chart. The associated best case time complexity is

\[
t_c(n, K) = O(K + \frac{qn}{T}) = O(K + \frac{\alpha n}{K})
\]

where \( \alpha \) is the proportion of vertices that fall into the viewport.

On the other hand, in the same case, we use

\[
m_o(n) = O\left(\sum_{i=0}^{log_2(n)} \sqrt{\frac{n}{2^i}}\right) = O(\sqrt{n})
\]

because we offer random access with polygon granularity.

The corresponding time complexity is

\[
t_o(n) = \frac{\log_2(n) - \log_2(\alpha n)}{2} \sqrt{\frac{n}{2^i}} + \alpha n = O(\sqrt{n} \times (1 - \sqrt{\alpha}) + \alpha n)
\]

When main memory availability becomes a problem, the equation 1 suggests that the best choice for \( K \) is \( \sqrt{n} \), in which case the memory complexity of both algorithms are roughly equivalent. In that case, our algorithm has a small time complexity advantage. The results given in [CKLS09] favor high compression ratios and speed of decompression over memory usage by using small values for \( K \) (100 \( \ll \sqrt{n} \)). Therefore, we cannot fairly compare compression ratios with their scheme because they increase with \( K \). However, by extrapolating on their results, we can suppose that their scheme is better for triangular meshes.

Our method seems better than [CKLS09] on quad or higher degree polygonal meshes. As they use the single-rate compression scheme of [KADS02] to compress the charts, we can expect their rates to be roughly the same for higher degree polygons as for triangles. However, our scheme is more efficient on higher degree polygons, because the vertices/faces ratio is lower. This is confirmed experimentally as quad meshes are compressed to 20 bpv instead of 27 for triangular meshes.

Once again, it is difficult to compare our view-dependant rendering times with the ones in [CKLS09] fairly, as the overhead for their method depends on \( K \). [CKLS09] renders a vertex in 1.35 microseconds, not including the overhead of decoding the vertices that lie outside the viewport but inside the partially visible charts. For reference, if we do not include this overhead, our algorithm decodes a vertex in 1 microsecond approximately.

We do not compare our results with [YL07], because their method is very different and provides features not addressed by our algorithm. It is targeted towards triangle mesh compression and imposes no traversal order for the mesh, enabling cache-oblivious approaches and competitive speeds at the cost of reduced compression efficiency.

5. Conclusion and Future Work

We have presented a novel algorithm for random accessible mesh compression. Our method is useful when certain interesting parts of a mesh need to be decoded without decompressing the whole mesh. We have shown that this approach is useful for view-dependant rendering, thus enabling interactive visualization of large meshes directly from their compressed form. We achieve seemingly better compression rates than similar approaches for quad and higher degree polygonal mesh compression. While the compression rates remain smaller than with single-rate compression, we think that random accessibility is worth the trade.
A very important works that is yet to be done is the generalization of our algorithm to meshes with handles. This could be done by allowing the binary tree of charts to have some genus removal nodes with only one child, where the cut wire would remove one genus instead of cutting the mesh in two. It remains to be shown how this can be done.

In addition to this, we believe that there is still much room for improvement. We want to determine better heuristics for determining cut wires, to enable quicker compression, better geometry compression, or more balanced subgraphs for faster random access. It would also be interesting to find memory layouts which enable the use of more efficient coding schemes than Huffman coding.

References


[IA02] Isenburg, Alliez: Compressing polygon mesh geometry with parallelogram prediction. In VIS ’02: Proceedings of the conference on Visualization ’02 (2002).


