Dependency Free Parallel Progressive Meshes

Evgenij Derzapf and Michael Guthe
Graphics and Multimedia Group, FB12, Philipps-Universität Marburg, Germany

Technical Report
August 23, 2011

Abstract

The constantly increasing complexity of polygonal models in interactive applications poses two major problems. First, the number of primitives that can be rendered at real-time frame rates is currently limited to a few million. Second, less than 45 million triangles – with vertices and normal – can be stored per gigabyte. While the rendering time can be reduced using level-of-detail (LOD) algorithms, representing a model at different complexity levels, these often even increase memory consumption. Out-of-core algorithms solve this problem by transferring the data currently required for rendering from external devices. Compression techniques are commonly used because of the limited bandwidth. The main problem of compression and decompression algorithms is the only coarse grained random access. A similar problem occurs in view-dependent LOD techniques. Due to the inter-dependency of split operations, the adaption rate is reduced leading to visible popping artifacts during fast movements. In this paper, we propose a novel algorithm for real-time view-dependent rendering of gigabyte-sized models. It is based on a neighborhood dependency free progressive mesh data structure. Using a per operation compression method, it is suitable for parallel random-access decompression and out-of-core memory management without storing decompressed data.

1 Introduction

The desire for high quality polygonal models in interactive applications is constantly increasing. Despite the enormous processing power of graphics processors (GPUs), highly detailed models cannot be rendered in real-time. Often they even do not fit into graphics memory since only models with up to 44.7 million triangles can be stored within a gigabyte. The standard solution to reduce rendering time are static or dynamic levels-of-detail (LODs). While static LODs are simply a set of polygon meshes, dynamic LODs store a coarse base mesh and a sequence of refinement operations. Dynamic LODs have the advantages that view-dependent adaption is possible and transitions between LODs, so-called popping artifacts, are much less visible. The most common data structure used in this context are progressive meshes. Sequential algorithms can however not process enough data to fully feed the GPU and the problem of all previous parallel approach are the local vertex dependencies. While this is unproblematic for serial algorithms, the dependencies drastically reduce the number of parallel operations. Thus they do not only increase the number of triangles but also reduce the adaption speed. A high adaption speed is on the other hand the only way to prevent popping artifacts since even prefetching algorithms cannot compensate slow adaption for more than a few frames.

Out-of-core techniques were developed as LOD techniques normally increase the total memory consumption. For static LODs, the model is typically partitioned using a spatial hierarchy. Then a single LOD is generated for each node. This results in a hierarchical LOD (HLOD) structure where only the currently required nodes need to be kept in memory. The approach can also be extended using dynamic LODs for each node for fully view-dependent adaption. In any case, special care must be taken at the boundaries between nodes to prevent visible holes in the model. Out-of-core techniques only shift the problem of limited fast memory to limited bandwidth of slower external devices. Compression techniques are widely used to reduce the bandwidth. Unfortunately, efficient compression approaches provide only coarse grained random access. For HLODs, the problem can be circumvented using node-wise compression. The contents of each node are compressed separately and decompressed during loading. The compression however cannot be used to reduce the graphics memory consumption.

We solve these problems with our fully random-accessible dependence-free progressive mesh data structure. It is not necessary to decompress the operations. By using an optional bounding volume hierarchy, it is suitable for in-core and out-of-core rendering. Our main contributions are:

- A view-dependent in-core and out-of-core full random-access compressed progressive mesh data
structure.
- No inter-dependencies between adjacent vertices to prevent waiting for dependent operations.
- A massively parallel adaption algorithm with stable, real-time frame rates.
- A bounding volume hierarchy for out-of-core rendering and occlusion culling.

2 Related Work

View-dependent simplification has been an active field of research over the last two decades. Hoppe [Hop96] introduced progressive meshes (PMs) that smoothly interpolate between different levels-of-detail. A sequence of split- or collapse operations can be performed for each vertex to generate a view-dependent simplification [XYV96, Hop97]. Hoppe later optimized the data structures and improved the performance of the refinement algorithm [Hop98]. Pajarola et al. [PR00] introduced compressed progressive meshes, that allow for a very compact coding, but view-dependent adaption was impossible. Pajarola and DeCoro [Faj01, PD04] developed an optimized sequential view-dependent refinement algorithm. Their FastMesh is based on the half-edge data structure and manages split-dependencies by storing a collapse-operation for each half-edge. Diaz-Gutierrez et al. [DGGP05] proposed a hierarchyless simplification algorithm that can also be used for stripification and compression. While they completely remove any interdependency of split operations, an efficient view-dependent adaption is not possible since that requires a split-hierarchy. Hu et al. [HSH09] proposed a parallel adaption algorithm for progressive meshes. They introduced a relatively compact explicit dependency structure that allows to group vertex splits and half-edge collapses into parallel steps. The drawbacks of this technique are the explicit dependencies that need additional memory and that only half-edge collapses are supported. A more compact progressive meshes data structure for parallel adaption was proposed by Derzapf et al. [DMG10a, DMG10b]. The problem of both approaches are however the local vertex dependencies that reduce the adaption performance.

The first HLOD approach was proposed by Erikson et al. [EMB01]. The problem of this technique is that no simplification along cuts between hierarchy nodes is possible without introducing visible gaps. Guthe et al. [GBK03] solved this problem by first using an unconstrained simplification of the nodes. The gaps are then filled during rendering using line strips. Cignoni et al. [CGG*04] proposed a different solution by creating alternating diamond shaped hierarchies. This way the triangles along a node boundary can be simplified at coarser levels. Finally, Borgeat et al. [BGB*05] proposed to use geomorphing to simplify the triangles along node boundaries during rendering. Unfortunately, the transform performance is approximately halved this way such that the previous two approaches are faster. Another approach are the Far-Voxels [GM05], which replace pixel sized triangles by a point and use an octree for point clustering. Sander et al. [SM06] proposed an algorithm that performs geomorphing on the GPU to render a given mesh. This approach extends the idea of Borgeat et al. and applies geomorphing on all triangles. The clustered hierarchy of progressive meshes (CHPM) approach [YSGM04] was the first to combine HLOD and progressive meshes. A progressive mesh is stored for each node to allow for smoother LOD transitions. Nevertheless, fully view-dependent adaption is still not possible due to the use of view-independent adaption inside each node.

Mesh compression approaches have good compression rates [TG99, AAR05], but random access is not possible. The first approach allowing random access was introduced by Choe et al. [CKL*04]. Kim et al. [KCL06] provide a more effective approach for random access compression, based on their multi-resolution data structure [KL01]. Yoon et al. [YL07] use streaming mesh compression to improve the compression rate over previous approaches. The data are divided into blocks and each block is compressed separately. The approach of Choe et al. [CKL09] is similar to [CKL*04] but contains some improvements. The performance of this approach was slightly improved by Du et al. [DJCM09] using a k-d tree. The approach of Courbet et al. [CH09] has a slightly better performance but uses a single-rate compression scheme. The CHuMI Viewer [JGA09] introduces a primary hierarchical structure (nSP-tree) in which a kd-tree is embedded to improve the performance. Although all previous compression approaches have good compression rate, only coarse grained random access is supported and interactive rendering is not possible without severe popping artifacts.

3 Progressive Mesh Data Structure

The proposed compressed progressive mesh data structure is based on Hoppe’s original progressive mesh algorithm [Hop96]. The progressive mesh is generated by simplifying the original mesh with a sequence of collapse operations until no faces are left. The original mesh can then be reconstructed by applying the corresponding split operations in reverse order. Figure 1 shows an edge collapse operation \( \text{coll}_v \), which removes the vertex \( v_q \) and modifies \( v_i \) to \( v \). The adjacent faces \( f_i \) and \( f_r \) of \( v_i \) and \( v_q \) degenerate and are removed from the mesh. The corresponding vertex split \( \text{spl}_v \) inverts this operation. Accordingly the faces \( f_i \) and \( f_r \) are generated when the vertex \( v \) is split into \( v_i \) and \( v_q \). In addition, some of the faces adjacent to \( v \) become adjacent to the new vertex \( v_q \), the others remain connected to \( v_i \).

After building a progressive mesh, a view-dependent re-
construction can be generated by performing only those split operations necessary for the current view. Originally, Hoppe [Hop96] explicitly encoded the vertex indices of v1 and v5, and the indices of the faces adjacent to v5. Xia et al. [XV96] optimized the data structures by encoding the vertex and face indices relative to the neighborhood of the split operation. The memory consumption can be drastically reduced this way since v1 and v5 can be encoded in a few bits. This encoding was previously used by Derzapf et al. [DMG10a, DMG10b] for parallel view-dependent refinement. While view-dependent adaption is possible, the local dependencies require that v2, r, and v5 exist when performing a split operation of v (see Figure 1). Hoppe [Hop97] proposed a slightly different approach that does not require v1 and v5 to be present, but nevertheless forces splitting of their ancestors afterwards to prevent foldovers. These of course cannot be encoded as compactly as in the previous approach. The faces adjacent to v2 are then found by traversing the edges in clockwise order from v1 to v5. Hu et al. [HSH09] later used a modification of this technique for parallel refinement. In both cases, the simplification constructs a forest of binary trees (Figure 2). The neighborhood dependencies (dotted lines) are either encoded explicitly, or implicitly by using special numbering of the vertices.

![Figure 1: Edge collapse and vertex split operation.](image)

While previous approaches require at least v1 and v5 to exist, we remove this constraint and only require v to exist. This allows to perform splits of all currently active vertices completely in parallel. In the example above (Figure 3), all four vertices can be split within a single adaption pass. We achieve this by encoding all possible topology modifications within the vertex indices of the faces. The faces are then stored along with the split where they are generated. To support non-manifold meshes, we first store the number of generated faces and then the faces themselves. The final indices FVID0,2 after applying all split operations are stored for each face. Now the vertices need to be numbered such that we can efficiently find the currently active vertex into which the vertex with FVID is collapsed. Figure 4 shows this numbering of the vertices. The leaf nodes of the binary tree forest are simply numbered from left to right. Then the collapsed vertex v receives the FVID of its left child v1 which is the smaller one. The resulting encoding of the faces is shown on the right side of the figure for a simple model. If a face is now decoded, when the split operation is performed, the current vertices are either those with the same FVID, or the one with the greatest FVID smaller FVID). In the example, we need to find the active vertex for the FVID 1 when performing split 1. The currently active vertex with the greatest FVID smaller than 1 is vertex 0 which is the collapse target of vertex 1.

In theory this can lead to foldovers as the generated triangles can be flipped if v1 or v5 change their position. This can however not happen if the simplification errors of collapsing v1 and v5 are at least that of collapsing v. Note, since the edge collapses are generated with increasing error, this is automatically handled during simplification. The local monotonicity is enforced by tracking the simplification error of the adjacent vertices during simplification and using the maximum of all neighbor’s errors as the actual error.

![Figure 2: Vertex hierarchy represented as a forest of binary trees with full (green) and reduced (red) neighborhood dependencies.](image)

![Figure 3: Dependent split operations.](image)
Now we only need to make sure that a vertex is split if any adjacent triangle is visible and the simplification error exceeds the screen space threshold to prevent visible foldovers. This would be the case if the model was adapted to a constant error, leading to the same sequence of operations as generated during simplification. Although we do not explicitly force splitting of \( v_t \) and \( v_u \), we did not notice any visible foldovers in our experiments as the error is smoothly changing over the mesh anyways.

### 3.1 Split Operation

In addition, each split operation needs to encode the refinement criteria for LOD selection, the vertex attributes of \( v_t \) and \( v_u \), and the references to the splits of \( v_t \) and \( v_u \). Table 1 gives an overview of the complete split operation data structure.

<table>
<thead>
<tr>
<th>group</th>
<th>element</th>
<th>memory (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>connectivity</td>
<td>( v_t ) (FVID)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>faces ( {P', ID} )</td>
<td>12</td>
</tr>
<tr>
<td>refinement criteria</td>
<td>( \Delta v ) (geometric error)</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>( \varepsilon_a ) (attribute error)</td>
<td>4</td>
</tr>
<tr>
<td>attributes</td>
<td>( v_t )</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>( v_u )</td>
<td>36</td>
</tr>
<tr>
<td>binary tree forest</td>
<td>address pointer</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>child pointer</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 1: Elements and size of the uncompressed split operation, where \( f \) is the number of generated faces and \( k \) the number of vertex attributes.

We use several view-dependent refinement criteria to determine whether a vertex needs to be split or can be collapsed. It can be collapsed if it is either outside of the view frustum or all adjacent triangles are back-facing. At runtime only the normal of the adapted vertex is available. We thus encode the maximum angular deviation \( \alpha \) from the normal of the simplified vertex forming a normal cone [Hop97]. Since each vertex of the adapted mesh can be adjacent to triangles on different levels-of-detail we need to consider the normals of all possibly adjacent faces. To prevent the computation of trigonometric functions at runtime, \( \sin \alpha \) is stored. A vertex needs to be split if it is visible and the simplification error exceeds some pre-defined limit in screen space. Instead of directly using the quadric error for the LOD selection we compute the geometric attribute error [GBBK04] after simplification to improve the visual quality. The simplification error is comprised of a geometric error \( \varepsilon_g \) and an attribute error \( \varepsilon_a \). While the attribute error is independent of the view direction \( \mathbf{d} \), the geometric error originates from a displacement in normal direction \( \mathbf{n} \).

As in most previous approaches, we do not directly store the attributes of \( v_t \) and \( v_u \), but only the differences \( \Delta v_t/u \) to the attributes of \( v \). This has the advantage that we can reconstruct the attributes of \( v \) from those of \( v_t \) and the data stored in the split operation.

For the binary tree forest we first encode whether \( v_t \) and \( v_u \) are further split. Then pointers to their operations are stored as address offset to the end of the current operation. We do not need to store an offset for \( v \), since that operation starts directly after the current one. We also do not need to store an offset for the next operation of \( v_t \) if only a single child is present.

#### 3.2 Compression

We use arithmetic compression [Sai04] to store the operations in graphics memory. Due to this optimal entropy coding, the key to achieve a high compression rate, is to reduce the entropy. Therefore, the rest of this section describes how we encode the data with low entropy. As each operation needs to be decoded independently, we use common probability tables but encode each operation in a separate byte stream. This way we only need the starting address to decode an operation. As the compression changes the length of the data and thus the starting address offsets, we need to perform a bottom up compression of the operations. With the sequential ordering of operations, this leads to compressing them in reverse order. In addition, we can only determine the symbol probabilities after compression. We therefore start with probability tables that are constructed with zero offsets and the re-compress the data with the correct probabilities afterwards. The overall compression thus works as follows:

1. Compute uncompressed operations with zero address offsets.
2. Compute the probability tables.
3. Compress the operations in reverse order, computing the correct address offsets.
4. Re-compute the probability tables and re-compress.

Most of the data form zero centered normal distributions that can be compressed quite well. As some contain only absolute values the others are remapped to positive numbers. We use the following mapping to maintain a normal distribution:

\[
\mu = \begin{cases} 
2v & v \geq 0 \\
2|v| - 1 & v < 0
\end{cases}
\]

where \( v \) is a variable from a signed distribution and the \( \mu \)'s form a positive distribution.
Arithmetic coding independently processes single bits or bytes to restrict the probability table to a reasonable size. Unfortunately the progressive mesh data do not fit into single bytes but often require 32 or even 48 bits in the out-of-core case. We therefore use a context based arithmetic compression. A separate probability table is used depending on the byte significance. If a preceding byte of the currently encoded value is non-zero, the probabilities drastically change. In this case an additional table is used to encode the successive bytes.

3.2.1 Successive Operations

The possibilities of split operations for $v_t$ and $v_u$ are sorted by descending probability and we store: 0 if none are present, 1 for both, and 2 and 3 for $v_t$ and $v_u$ only. As mentioned above, we only need to store the address of $v_u$ since the operation of $v_t$ directly starts after the one of $v_u$. The address offset $offset$ for $v_u$ can be estimated as $s_{avg}(v_u - v - 1)$ if we know the average operation size $s_{avg}$. Then only the difference to the estimation is stored.

3.2.2 Connectivity Coding

In theory we do not need to store the FVID of $v_u$, as the difference to $v$ simply one plus the number of split operations in the subtree of $v_t$. If no split for $v_t$ exists, we thus know that the offset is one and do not need to store it. Otherwise it is at least two and we store the offset minus two to prevent traversing the subtree. The triangle indices (FVIDs $2, 3$) are sorted such that the first two are children of $v_t$ and $v_u$ respectively. Then these two are encoded as differences to the FVIDs of $v_t$ and $v_u$. While we could also encode the topology modifications of $v_t$ and $v_u$ bit-wise, as proposed by Kim and Lee [KL01]. This would however not be optimal as the tree is not balanced. The third vertex cannot be a child vertex of $v$. Its FVID is either less than $v$ or at least the next higher FVID $v_u$ of the currently active vertices. In the first case, a negative offset to $v$ is stored and in the second case, a positive offset to $v_u$. The offset $\nu_3$ is then again mapped to a positive value as described above. If more than one triangle is generated by the operation we can exploit an additional degree of freedom. First the triangle with the smallest third vertex offset $\nu_3$ is compressed. The successive triangles are sorted by increasing $\nu_3$ and only the difference to the previous offset is stored. In addition to the indices, we also store the number of faces to support non-manifold meshes and improve the compression rate for boundary edges.

3.2.3 Refinement Criteria

As no high accuracy is required for back-face culling, $\sin \alpha$ is quantized to eight bits. Due to the shrinking neighborhood, $\sin \alpha$ becomes smaller for the successive splits. We thus encode the difference to the parent operation to exploit this fact. A separate probability table is used as the distribution is nevertheless not centered at zero but at a model dependent value.

We encode $\epsilon_g$ and $\epsilon_n$ together as the screen space error $\epsilon_s$ is a combination of these two: 

$$\epsilon_s = \max(\epsilon_g, \epsilon_n(d \cdot n)),$$

where $d$ is the view direction and $n$ the normal. The geometric error is only relevant if it is greater than the attribute error and we thus encode the maximum error $\epsilon$ and the ratio $\mu$ of $\epsilon_a$ to $\epsilon_g$. Then we can then simply clamp $\mu$ to be at most one. Similar to the normal cone angle, the probabilities significantly deviate from a normal distribution. Thus we also encode $\mu$ quantized in a single byte with its own probability table. The maximum error can vary over a huge range. The upper bound depends on the model size and its lower bound is the quantization step $q$. To compactly encode this range, we exploit the fact that only a rather low accuracy is required. In our implementation we use a relative accuracy of 2% which is expressed by the following coding function:

$$\epsilon_{enc} = \left[ \ln(\epsilon - \ln q) \right] / \ln 1.02,$$

where $\epsilon_{enc}$ is the encoded simplification error. Similar to $\sin \alpha$, the simplification error is monotonously decreasing and we also encode the difference to the parent operation. In contrast to $\sin \alpha$ and $\mu$, the difference exhibits a normal distribution.

Note that the refinement criteria are stored as the first four bytes of the operation since they need to be decoded first to evaluate the necessary operations. The ID of $v_u$ and the number of faces are stored directly after the refinement criteria as they are also required when checking for and preparing the required operations.

3.2.4 Attributes

To prepare the attributes for compression, we first quantize each coordinate. The quantization step is chosen depending on the progressive mesh to be encoded. We calculate the root mean square attribute difference $\bar{\sigma}_i$ for each attribute $i$ over all operations and then use $q = a_{scale} \sigma_i$ as quantization step. In our implementation we chose $a_{scale} = \frac{1}{16}$ which is a reasonable trade off between accuracy and compression rate. Assuming the vertices were collapsed to their midpoint, we use second order prediction for $\Delta v_u$ (i.e. $-\Delta v_t$) and store the difference. More sophisticated estimations using subdivision schemes are not possible as this would again require at least the adjacent vertices to be present.

Note, that $v_t$ and $v_u$ of each split operation can be swapped without altering the encoded progressive mesh. We exploit this to improve the compression rate. All operations are checked and if the total size of the encoded model is reduced when swapping the vertices, the swap is
3.3 Out-of-Core Hierarchy

We additionally build a bounding volume (BV) hierarchy over the split/collapse operations for out-of-core rendering. The hierarchy serves two purposes: first, the operations should be grouped such that those which are likely to be performed simultaneously or successively are stored together. And second, it should be used for occlusion culling in order to coarsen invisible parts of the model. During hierarchy construction it thus needs to be optimized for both purposes. Meißner et al. [MBH01] proposed a simple heuristic to construct efficient kd-tree hierarchies of triangle meshes for occlusion culling using a greedy algorithm. We adapt this approach to a bounding volume hierarchy of variable size split operations. Note that storing operations only differs from the hierarchy used in Quick VDR, where each node contains a complete progressive mesh. We do not only store operations at leaf level but also at inner volumes to reconstruct coarse approximations of the model.

When processing a BV, we first need to determine the operations that are stored in it. Then the operation subtrees are partitioned into the child nodes. Finding the directly stored operations is straightforward as those with the highest simplification error are required first. After storing the operations in the current node their operation subtrees are partitioned into the child nodes. This way a complete operation subtree is stored in a single BV hierarchy subtree. Due to storing operations not only at the leaf volumes, the estimated subtree area is slightly modified:

\[ A \approx A_l \log_2 \left( \frac{s_l}{s_{\text{max}} + 1} \right) + A_r \log_2 \left( \frac{s_r}{s_{\text{max}} + 1} \right), \]

where \( A_l \) and \( A_r \) are the bounding box areas of left and right child node and \( s_{l/r} \) the size of the operations in bytes.

The operation address is also split into a BV index \( i_n \) and the local address depending on the index \( addrl \) inside the hierarchy node due to the partitioning of operations. Using a node size of up to \( 2^{16} \) bytes, the combined offset \( o_{\text{ooc}} \) is stored as:

\[ o_{\text{ooc}} = \begin{cases} i_n = n & : o_{\text{addr}} \\ i_n \neq n & : 2^{16} (i_n - n) + addr_l, \end{cases} \]

where \( o_{\text{addr}} \) is the offset inside the node and \( n \) the current BV node.

In contrast to the in-core case, \( o_{\text{ooc}} \) of \( V_l \) can be non-zero and needs to be stored as well. Additionally, the offsets cannot be estimated any more since we do not know how many subtree operations are contained in the current node.

4 Runtime Algorithm

The adaption algorithm is subdivided into several consecutive steps to implement the refinements on massively parallel hardware. The partitioning is required for thread synchronisation while each step can be processed completely in parallel. First, each vertex is classified to be split, kept, or collapsed. Then the necessary operations are performed on the adapted mesh. This mesh is then used as input for the next frame to exploit temporal coherence.

The dynamic data structures required for rendering are the vertex buffer containing the position and attributes and the index buffer storing the connectivity of the adapted mesh. Both are stored as vertex buffer objects (VBOs) and are separated from all other data. Table 2 gives an overview of all dynamic data structures that are discussed in the following.

![Figure 5: Split/collapse operation hierarchy represented as a forest of binary trees.](image)

**Figure 5** compares the operation ordering for the in-core and out-of-core case.

<table>
<thead>
<tr>
<th>Buffers</th>
<th>Elements</th>
<th>Memory (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex</td>
<td>VBOs</td>
<td>540</td>
</tr>
<tr>
<td>vertex ID</td>
<td>(1/2)</td>
<td>360</td>
</tr>
<tr>
<td>vertex vertex split &amp; collapse cache</td>
<td>(1/2)</td>
<td>360</td>
</tr>
<tr>
<td>vertex collapse target</td>
<td>(1/2)</td>
<td>360</td>
</tr>
<tr>
<td>vertex split &amp; collapse</td>
<td>(1/2)</td>
<td>360</td>
</tr>
<tr>
<td>temporary</td>
<td>face count</td>
<td>480</td>
</tr>
<tr>
<td>vertex prefix sum</td>
<td>480</td>
<td></td>
</tr>
<tr>
<td>face prefix sum</td>
<td>480</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td></td>
<td>1112</td>
</tr>
</tbody>
</table>

Table 2: Elements of the dynamic data structure. \( k \) and \( m \) are the number attributes and adapted mesh vertices. Next split and collapse are stored with 32 bits in the in-core (a) and 48 bits in the out-of-core case (b).
4.1 Vertex State Update

In the first step we determine the necessary operations. If the vertex \( v \) needs to be split according to its refinement criteria, we set its number of child vertices in the next iteration to two. Additionally the number of faces that are created by this operation is decoded. Otherwise the number of vertices is set to zero if the refinement criteria allow a collapse, or to one if not. The collapse of a vertex is only possible if its corresponding and its target vertex \( v_t \) have not performed further splits. This can efficiently be checked by keeping the vertices sorted based on their FVID. In this case the previous vertex of \( v \) needs to be its target \( v_t \) and the target of the next vertex must not be \( v \).

Two or three refinement criteria are checked for each active vertex for in-core and out-of-core rendering respectively. The most simple one is view frustum culling as a vertex can be collapsed if it lies outside the view frustum regardless of the simplification error. To prevent foldovers, we do not simply collapse all vertices that are outside of the view frustum but modify the distance \( d \) of these vertices for the following LOD selection:

\[
\tilde{d} = \left( c_{LOD} \left( \frac{\max(|x|,|y|,|z|)}{w} - 1 \right) + 1 \right) d,
\]

where \( x, y, z \) and \( w \) are the homogeneous coordinates of the vertex after projective transformation. In our experiments \( c_{LOD} = 100 \) is used for a smooth LOD falloff outside the view frustum which prevents foldovers and popping artifacts when rotating or panning.

The next test is back-face culling. The vertex is culled if \( \mathbf{n} \cdot \mathbf{d} > \sin \alpha \), where \( \mathbf{d} \) is the direction, \( \mathbf{p} \) the vertex position, \( \mathbf{n} \) the normal, and \( \alpha \) the normal cone angle as discussed in Section 3.1. In addition to these tests we use occlusion culling for out-of-core rendering. The occlusion culling is based on the visibility of the bounding volumes. The vertex can be collapsed if its split or collapse is stored in an occluded bounding volume. We use hardware occlusion queries [CCG+01] to determine which bounding volumes are visible. The queries are performed after rendering the complete scene and the results are fetched before the next adaptation. This way the visibility is lagging behind by one frame but this is unproblematic as it is not used for rendering but for LOD selection only. Regardless of the query results we always render the complete adapted mesh. This approach is very efficient because the number of bounding volumes is small compared to the number of triangles, we only use asynchronous queries, and only require a single switch between rendering and occlusion queries per frame. The simplification error is evaluated after culling. Each active vertex has an eight byte cache storing the split and collapse refinement criteria to reduce decoding time and unaligned global memory access. The split cache only needs to be updated if the vertex was modified or created by a split operation in the previous frame. The complete vertex update is shown in Algorithm 1.

\[
\text{Algorithm 1: Parallel vertex state update algorithm.}
\]

4.2 Parallel Edge Collapses

To perform a collapse, we need to check whether the target \( v_t \) of the current vertex \( v_u \) was not marked for splitting in the first stage. The collapse operation then simply moves vertex \( v_t \) to its old position \( v \) and copies the collapse cache of \( v_u \) to the split cache of \( v \). Removal of vertex \( v_u \) and the degenerate faces are handled in later stages. Algorithm 2 shows the parallel processing of the edge collapse operations to prepare removal of the collapsed vertices and faces.

\[
\text{Algorithm 2: Parallel edge collapse algorithm.}
\]

4.3 Memory Management

Memory must be reserved for the additional vertices and faces before the split operations can be applied. While the ordering of faces is irrelevant for the algorithm, the vertices must be sorted by their FVID as discussed above. If the index and FVID buffers are large enough, the faces can be directly appended. To determine the position of the faces in the index buffer, the parallel prefix sum [SHZ07] of the number of generated faces is computed. After calculating the prefix sum, we can determine the total number of faces after all split operations. If the size of the face buffers is too small or significantly too large, new buffers are allocated and the content of the old ones is copied into them. When a reallocation is performed, the buffer size is set to the number of faces \( n_f \) plus a user-defined threshold \( n_{alloc} \). If the buffer is larger than \( n_f + 2n_{alloc} \), it is reduced to \( n_f + n_{alloc} \).

While the buffer resizing is applied to the vertices as well, new vertices cannot be simply appended to the vertex buffer. They need to be sorted by their ID as discussed above. This means that they have to be inserted after the corresponding split vertex. We accomplish this by copying the old vertices into a new buffer. During this process, the collapsed vertices are also removed by calculating the parallel prefix sum of the vertex count to determine the positions in the new buffer. The vertex IDs, caches and next split/collapse buffers need to be processed this way as well. While the memory of the old buffers could be freed after
this step, the repeating allocation would drastically reduce the performance. The complete parallel reorganisation and compaction of the vertex buffer is shown in Algorithm 3.

4.4 Parallel Vertex Splits

The split operations are performed after memory allocation and reorganisation of the vertex buffer. To improve thread utilization we first compact the splits [SHZO07] such that each thread performs an operation. Every operation generates a new vertex \( v_a \) and moves \( v \) to its new position \( v_r \). Additionally, the new faces are added to the index and the FVID buffers. For this we need to determine the current indices for each of the new faces. Fortunately, the first two vertices of each new face are known as they are \( v_r \) and \( v_a \). The third vertex needs to be located in the vertex buffer. By construction it is the vertex with the greatest FVID less or equal to the one stored in the face. We use the Binary Search to find this vertex in the vertex buffer. Note that while this does not fully utilize memory bandwidth it keeps the threads within every warp running in parallel which is important for performance. Algorithm 4 shows the parallel vertex split.

Algorithm 4: Parallel vertex split algorithm.

4.5 Index Update

The indices of the faces adjacent to split and collapse vertices need to be updated (e.g. \( f_{n-1} \), \( f_n \), \( f_{l_0} \), \( f_l \) and \( f_r \) in Figure 1) after performing all operations. This is necessary as the vertices of adjacent faces can perform their operations in parallel. The correct vertex can either be the previous one, the current one, or the next vertex in the sorted array. The first case occurs when the vertex was collapsed, while the last one occurs when the vertex was split. The second case can either happen when no operation was performed or the operation did not change the connectivity of that face. Algorithm 5 shows the parallel index update.

Algorithm 5: Parallel index update.

4.6 Buffer Compaction

The final step of the adaption is the compaction of the index buffers to delete degenerate faces. Note that as the index VBO is used for rendering it needs to be compact anyways. We use a specialized in-place compaction algorithm [DMG10a] since the ordering does not need to be preserved. The main advantage besides a minor speedup is that we do not need to duplicate these buffers.

4.7 Out-of-Core Memory Management

An additional memory management of the static data structures is performed for out-of-core rendering. Only the currently necessary bounding volumes are kept in graphics memory based on a priority scheduling. All relevant data is stored in graphics memory and the main memory consumption is minimal as the complete algorithm runs on the GPU. In addition to temporary memory for loading, only the mapping of bounding volumes to memory positions, the file offsets required for out-of-core management, and the bounding boxes are stored in main memory. For loaded bounding volumes we also store pointers to their data in device memory. A bounding volume is required when at least one of the active vertices has a reference to it. A split operation can create vertices that reference volumes not available in graphics memory. In this case the data needs to be loaded from disk. Each bounding volume contains operations with different simplification errors. Based on the maximum error of the operations stored in a volume, we can derive a distance \( d_v \) beyond which no split is ever necessary. Then we calculate a priority \( p = \frac{d_v}{\Delta t} \), where \( d_v \) is the distance between the viewer and the bounding box. The data of the bounding volume is only required if its priority is at least one. The bounding volumes with higher priority are loaded first if the transfer from disk to graphics memory is not fast enough. This has the advantage that the model is uniformly adapted and no LOD starvation can occur. To limit the memory consumption, a maximum number of nodes \( n_{max} \) kept in graphics memory is specified by the user. When rendering several progressive meshes, the node memory is shared among all models. When the user moves through the scene, the visibility and the required LOD of the object change. This results in a continuous change of the bounding volumes currently required in graphics memory. As discussed above, loading data results in a visible delay of the adaption. We solve this problem by not only loading the currently required nodes, but also the nodes with lower priority, as long as enough space is available. Before up-
loading the currently required bounding volumes to graphics memory we first remove unnecessary ones until enough space is available.

Since accessing the hard disk causes high delays, loading operations into main memory is performed in a second thread. As soon as the data is available in main memory, the rendering thread can copy it into graphics memory after scheduling the occlusion queries.

5 Results

Our test system consists of a 3.333 GHz Intel Core i7-980X CPU with 6 GB DDR3-1333 main memory, 16 lanes PCIe 2.0 slot, and an NVIDIA GTX580 (841/4204MHz) graphics card. OpenGL is used for rendering and CUDA for the adaption algorithm. The out-of-core data is stored on a SATAII hard disk (8.5ms/64MB/7200rpm) with approximately 100 MB/s read speed. The bounding volume data size is set to 64 kB, as host to device copy of blocks with up to this size is asynchronous. We use a resolution of 1920 × 1080 with a screen space error of 0.5 pixel. We used at most \( n_{\text{max}} = 4096 \) (256 MB cache) for all out-of-core models. Table 3 gives an overview of the progressive meshes we tested in our experiments. All models use position and normal as vertex attributes \((k = 6)\). The original meshes contain \(v_{\text{max}}\) vertices and \(f_{\text{max}}\) faces. The number of base mesh faces is zero and that of base mesh vertices \(v_0\) is very low. The number of operations is \(v_{\text{max}} - v_0\). The resulting file sizes and compression rates for the in-core (ic) and out-of-core (ooc) case are listed in Table 3 together with the number of BV nodes and bytes per vertex (bpv). Compared to in-core, the out-of-core static data requires approximately 1 additional byte for each operation. On average 1.2 bpv are consumed by the refinement information and 1.0/2.0 bpv by the tree structure in the in-core and out-of-core case. The connectivity and geometry need \(2.4\) and \(5.8\) bpv and \(0.6\) bpv are wasted due to the per operation compression. Additionally, 35 bytes main memory and 1.0/2.0 bpv by the tree structure in the in-core case. The connectivity and geometry need

![Figure 6: Renderings of view-dependently refined meshes. The external views show the view frustum (yellow), LOD (red: low; green: high), and the nodes used for occlusion culling (red: occluded; green: visible).](image)

Table 3: Progressive meshes examined in our experiments.

Table 4 lists the number of rendered faces, the total rendering time, and the memory consumption for the views shown in Figure 6, and the scene in the accompanying video, where the numbers are taken from the most complex frame. The ratio compared to an indexed face set (IFS) of the original model is shown in parenthesis. During rendering, the dynamic data structures consume additional memory. The total amount of graphics memory nevertheless stays below that of the original models. The out-of-core algorithm facilitates occlusion culling and out-of-core memory management, therefore the frame time is approximately 10% higher compared to in-core rendering although the number of faces is approximately 10% lower. We can process up to 244/226 million triangles per second (MTPS) for static views in the in-core and out-of-core case respectively. The frame time linearly increases with the number of faces. This face count converges to a constant value with increasing model size which is a typical behavior of all LOD algorithms. Therefore, the frame time converges

Table 4: Memory consumption, rendering time, and triangle rate (MTPS) of the different models for static views.

![Table 4](image)
to a constant value as well. The same holds for the memory consumption of the out-of-core algorithm. As our GPU can render an indexed face set with 600 MTPS, the performance of our method is faster than rendering the original model as soon as 60% of the faces are removed. Figure 6 also shows the coarsening of culled faces.

The adaption and rendering time together with the memory consumption and number of faces for a pre-recorded movement through the scene are shown in Figure 7. The consumed memory is always below 796 MB, the frame rate is constantly above 30 frames per second (fps) with an average of 50 fps and 140 MTPS. Our approach quickly reacts to changes of the view with fast adaption of the scene complexity. Due to the high adaption performance no popping artifacts are visible in the video despite fast movements and the screen space error of 0.5 pixel is always achieved.

![Figure 7: Timings, memory consumption and triangle rate for the scene using a pre-recorded camera path.](image)

In Table 5 we compare our algorithm with three different types of approaches. We can render 180/140 MTPS (incore/out-of-core) for dynamic views and up to 23/13 MTPS can be generated. Note that the number of generated triangles of our out-of-core algorithm is limited by the HDD speed. Due to the slightly reduced number of triangles for identical views, the relative rendering performance is 155 MTPS with up to 14.44 MTPS generated. The progressive mesh requires 11-14 bytes per vertex (bpv).

While compression approaches of course achieve better compression ratios, they have significant shortcomings regarding rendering. First of all, most of them do not support extracting a level-of-detail and thus only support view frustum culling (VFC) [CKLL09, CH09]. Others only allow simple level-of-detail schemes based on regular vertex clustering [DJCM09, JGA09]. This is however known to require at least an order of magnitude more primitives to achieve the same quality. Note that high compression ratios are also achieved by not encoding vertex normals which also reduces the visual quality as the normals computed from a simplified mesh can drastically differ from correctly simplified ones. Another problem is the complex connectivity coding that prevents parallel decompression. The fastest compression approach only achieves decoding of up to 0.3 MTPS. Considering the increased number of triangles compared to our approach, the relative adaption performance is less than 30,000 triangles per second so it at least 500 times slower. Once generated, the resulting mesh can be rendered at full performance. Again we need to consider the tenfold increase in model complexity which translates to a relative performance of less than 60 MTPS or less than 40% compared to our algorithm. So our approach can render the same view more than twice as fast and can adapt the LOD more than two orders of magnitude faster.

Hierarchical level-of-detail (HLOD) algorithms also generally achieve high rendering performance unless special shaders are used [GBK03, GM05]. Compared to view-dependent progressive meshes, the number of primitives is however drastically increased. This is due to three reasons: First, the LOD is only evaluated per node of the hierarchy. This already doubles the number of primitives considering that there is usually a resolution factor of two between successive level. The second reason is that the LOD is only distance instead of fully view-dependent which prevents coarsening of back-facing and non-silhouette triangles. This also approximately doubles the number of primitives. Finally, special care must be taken at the node boundaries which also slightly increases the primitive count. In total, the number of primitives is 5 to 7 times higher for the same view [GBK03, CGG*04]. This factor can exceed 10 if vertex clustering is used [GM05]. The number of generated triangles either depends on the hard disk speed or the mesh decompression. On our system, between 3.4 and 4.6 MTPS can be generated. Due to the higher primitive count, these are reduced to a relative performance of at most 1.47 MTPS [GBK03] (10% of our adaption performance). The relative rendering performance is also reduced to at most 76 MTPS [CCGG*04] or 49% of our approach. In summary, our approach renders approximately twice as fast and can react ten times quicker to view changes.

![Table 5: Comparison of triangle rate and memory consumption with previous approaches. The relative performance is shown in parenthesis.](image)
tives. The actual factor depends on whether view- or distance-dependent adaption is used. It also depends on the degree of neighborhood dependencies and lies between 1.05 [HSH09] and 1.2 [DMG10a] with view-dependent adaption and 2.5 [YSGM04] otherwise. The rendering performance is significantly lower than for HLODs due to the continuous geometry changes and interleaving adaption with rendering. The relative rendering performance lies between 28 MTPS [HSH09] and 66 MTPS [DMG10a] which is 16-37% of our approach. The adaption performance of the GPU algorithms (up to 4.4 MTPS) is significantly higher than CPU algorithms [YSGM04] with only 15,000 triangles per second. The relative adaption rate is at least 2.3 times faster and adapt the LOD four times faster.

Finally, we analyze the runtime of each step of the adaption and rendering algorithm in Figure 8. As the rendering performance is identical to rendering a static model with the same number of triangles, our method needs approximately four times as long as rendering a static mesh. Considering that we already cut down the vertices significantly due to the simplification of culled faces, our method will almost always be faster than rendering the original model. The most expensive steps of our algorithm are the state update and reorganisation. The state update is expensive because each active vertex needs to perform this step and the reorganisation as we need to maintain a sorted vertex buffer. The split/collapse cache reduced the state update time by 40%. Map and unmap are required for the mapping and unmapping of the index and vertex buffer for access from CUDA and can hardly be reduced or prevented.

6 Conclusion and Limitations

We have proposed an in-core and out-of-core dependence free progressive mesh representation that is specifically designed for parallel view-dependent adaption. It is based on an implicit coding of topology modification inside the faces. In contrast to previous approaches no splits need to be postponed as they are waiting for others to be applied before them, which is otherwise very problematic for fast movements. Compared to progressive meshes and HLOD approaches we reduce the rendering time, popping artefacts and the memory consumption significantly. This allows rendering of large models with fast movement nearly without popping artefacts. Compared to compression approaches we require more disk space, but can keep compressed data in graphics memory. Further drawbacks of compression approaches are coarse grained random access and slow decompression, resulting in severe popping artefacts. Moreover, the refinement criteria and normals are not encoded. This improves the compression rate, but reduces the quality and significantly increases the number of faces.

The main limitation of our algorithm is the lossy compression of vertex attributes. While the accuracy is high enough for rendering purposes, it might be problematic in other areas, like surface analysis and collision detection. Another limitation is that the reorganisation of the vertices is rather expensive. An acceleration or prevention of this step would significantly increase the performance. Of course, the efficiency of the method solely depends on the underlying mesh simplification. If a model cannot be reduced using geometric simplification, other approaches (e.g. Fax Voxels [GM05]) are better suited.

References


[DGGP05] DIAZ-GUTIERREZ P., GOPIL M., PAJAROLA R.: Hierarchyless simplification, stripification and com-


