Dependency Free Parallel Progressive Meshes

E. Derzapf and M. Guthe

Graphics and Multimedia Group, FB12, Philipps-Universität Marburg, Germany
{derzapf, guthe}@mathematik.uni-marburg.de

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.

Keywords: Level of Detail Algorithms, Real-Time Rendering, Data Compression

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms I.3.1 [Computer Graphics]: Hardware architecture—Parallel processing

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 using (32 bit floats) 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

© 2012 The Author(s)
Journal compilation © 2012 The Eurographics Association and Blackwell Publishing Ltd.
Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

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 before storing them in graphics memory. 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, realtime 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 [XV96, 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 [Paj01, 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 inter-dependency 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 FarVoxels [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 geomorping 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 [TR99, 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. [CKLL09] 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. Overview

The proposed compressed progressive mesh data structure is based on Hoppe's original progressive mesh algorithm [Hop96]. The progressive mesh is generated by sim-

© 2012 The Author(s)
Journal compilation © 2012 The Eurographics Association and Blackwell Publishing Ltd.

plifying 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 col_v which removes the vertex v_u and modifies v_t to v. The adjacent faces f_l and f_r of v_t and v_u degenerate and are removed from the mesh. The corresponding vertex split spl_v inverts this operation. Accordingly the faces f_l and f_r are generated when the vertex v is split into v_t and v_u. In addition, some of the faces adjacent to v become adjacent to the new vertex v_u, the others remain connected to v_t.

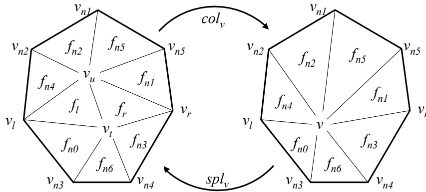

Figure 1: *Edge collapse and vertex split operation.*

After building a progressive mesh, a view-dependent reconstruction can be generated by performing only those split operations necessary for the current view. Performing a local adaption requires a random access data structure that allows to locally perform the operations. While the operations are already local by definition, the method of encoding the connectivity strongly influences the degree of locality. The main idea of our data structure is to store the connectivity changes in the triangles instead of storing it within the operation. This way, the connectivity of each face can be updated without considering its current neighborhood.

3.1. Neighborhood Dependencies

Originally, Hoppe [Hop96] explicitly encoded the vertex indices of v_l and v_r, and the indices of the faces adjacent to v_u. 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 v_l and v_r 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 v_l, v_r, and $v_{n1} - v_{n5}$ exist when performing a split operation of v (see Figure 1). Hoppe [Hop97] proposed a slightly different approach that does not require v_l and v_r 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 v_u are then found by traversing the edges in clockwise order from v_l to v_r. 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 a special numbering of the vertices.

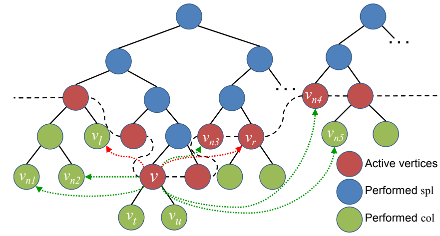

Figure 2: *Vertex hierarchy represented as a forest of binary trees with full (green) and reduced (red) neighborhood dependencies.*

The drawback of these approaches is that when vertex v needs to be split, it often needs to wait for neighboring splits. Figure 3 shows such a case. The vertices v_1 and v_3 need to split before v_4. In addition, v_2 also needs to split if all neighboring vertices are required. As splitting v_3 requires v_{2u} and v_2 requires v_{1u}, the splits can only be performed sequentially. While this is unproblematic in a sequential refinement algorithm, a parallel algorithm needs four adaption passes. By removing all neighborhood dependencies, our method can split all four vertices in parallel.

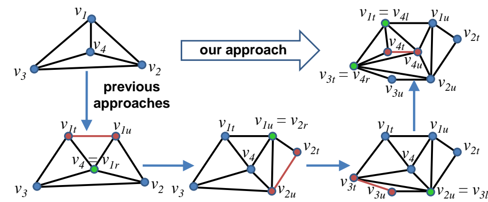

Figure 3: *Dependent split operations. Each arrow denotes a parallel adaption step.*

3.2. Split Operations

While previous approaches require at least v_l and v_r 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 faces are encoded by storing their vertex indices $FVID_{0..2}$ for the finest resolution. The current indices can then be found by searching those vertices into which the final vertices are collapsed. Thus the vertices need to be numbered such that we can efficiently find the currently active vertex into which the vertex

© 2012 The Author(s)
Journal compilation © 2012 The Eurographics Association and Blackwell Publishing Ltd.

with FVID_i 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 v_t 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_i. 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.

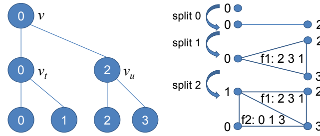

Figure 4: *Computation of the final vertex IDs and encoding of the generated faces.*

In theory this can lead to foldovers as the generated triangles can be flipped if v_l or v_r change their position. This can however not happen if the simplification errors of collapsing v_l and v_r 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. 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_l and v_r, we did not notice any visible foldovers in our experiments as the error is smoothly changing over the mesh anyways. 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. Finally, the operations are compressed using arithmetic coding to reduce the memory consumption. In contrast to previous approaches the compression is performed independently for each operation to retain random access.

3.3. GPU Adaption

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.

4. Data Structure

Table 1 gives an overview of the complete split operation data structure. 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 α 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 ε_g and an attribute error ε_a. While the attribute error is independent of the view direction $\mathbf{d}$, the geometric error originates from a displacement in normal direction $\mathbf{n}$.

group	element	memory (bytes)
connectivity	v_u (FVID)	4
	number of faces	1
	faces (FVID)	$12f$
refinement criteria	α (normal cone angle)	4
	ε_g (geometric error)	4
	ε_a (attribute error)	4
attributes	Δv_t	$4k$
	Δv_u	$4k$
binary tree forest	children present	1
	child pointer	4

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.*

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_t since that operation starts directly after the current one. We also do not need to store an offset for the next operation of v_u if only a single child is present.

4.1. 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 are 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

© 2012 The Author(s)
Journal compilation © 2012 The Eurographics Association and Blackwell Publishing Ltd.

coarsen invisible parts of the model. During hierarchy construction it thus needs to be optimized for both purposes. Meißner et al. [MBH*01] 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. Storing operations only is different than the hierarchy used in Quick VDR, where each node contains a complete progressive meshes. 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:

$$A \approx A_l \log_2 \left\lceil \frac{s_l}{s_{max}+1} \right\rceil + A_r \log_2 \left\lceil \frac{s_r}{s_{max}+1} \right\rceil,$$

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.

4.2. Operation Encoding

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:

$$u = \begin{cases} v \geq 0 & : \quad 2v \\ v < 0 & : \quad 2|v| - 1, \end{cases}$$

where v is a variable from a signed distribution and the u'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. We perform a bottom up coding of the operation tree compressing the operations from end to start. As the relative symbol frequencies have changed, we also rebuild the probability table and restart compression. The spatial hierarchy is also rebuilt after each compression run as the operations might exceed the maximum node size. We iteratively reduce the maximum node size when building the hierarchy and store the model as soon as all nodes are small enough after compression.

4.2.1. 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. The address offset o_{addr} 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. In the out-of-core case the operation address is split into a BV index i_n and the local address depending on the index $addr_l$ inside the hierarchy node due to the partitioning of operations. Using a node size of up to 2^{16} bytes, the combined offset o_{ooc} is stored as:

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

where o_{addr} is the offset inside the node and n the current BV node. In contrast to the in-core case, o_{ooc} of v_t can be non-zero and needs to be stored as well. Additionally, the offsets

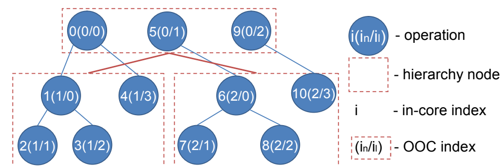

Figure 5: *Split/collapse operation hierarchy represented as a forest of binary trees.*

© 2012 The Author(s)
Journal compilation © 2012 The Eurographics Association and Blackwell Publishing Ltd.

cannot be estimated any more since we do not know how many subtree operations are contained in the current node. Figure 5 compares the operation ordering of both cases.

4.2.2. Connectivity Coding

In theory we do not need to store the FVID of v_u as the difference to v is 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. In practice, not storing the FVID of v_u in other cases would require to traverse the whole subtree. As it is then at least two, we store the offset minus two to prevent the traversal. The triangle indices ($FVID_{0..2}$) 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 FVID's 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_n 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_n. The offset vo_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 vo_3 is compressed. The successive triangles are sorted by increasing vo_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.

4.2.3. 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 ε_g and ε_a together as the screen space error ε_s is a combination of these two:

$$\varepsilon_s = max(\varepsilon_a, \varepsilon_g(\mathbf{d}\cdot\mathbf{n})),$$

where $\mathbf{d}$ is the normalized view direction and $\mathbf{n}$ the normal. The geometric error is only relevant if it is greater than the attribute error and we thus encode the maximum error ε and the ratio μ of ε_a to ε_g. Then we can then simply clamp μ to be at most one. Similar to the normal cone angle, the probabilities significantly deviate from a normal distribution. Thus we also encode μ 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:

$$\varepsilon_{enc} = \left\lceil \frac{\ln\varepsilon - \ln q}{\ln 1.02} \right\rceil,$$

where ε_{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 μ, 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.

4.2.4. 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 $\overline{\sigma}_i$ for each attribute i over all operations and then use $q = a_{scale}\overline{\sigma}_i$ 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 Δv_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 performed. The size reduction can be due to the different vertex attributes (especially for half-edge collapses), the FVID of the generated vertex, and – especially for operations close to the base vertices – due to smaller FVID offsets. To exactly calculate the compression improvement we would need to compress the complete progressive mesh twice for each operation, leading to a total complexity of $O(n^2)$. Swapping an operation however only influences the geometry offsets of the operation itself and the FVID offsets of all operations on neighbor vertices. Using an fixed approximate probability table, we can estimate the first one in $O(n)$ time for all operations. The neighborhood for the FVID offsets grows by a factor of $\sqrt{2}$ with each level of the hierarchy. Thus the total number of influenced operations throughout the whole hierarchy is: $\sum_{i=0}^{\log_2 n} 2^i\sqrt{\frac{n}{2^i}} = O(n)$. Starting from leaf level, we collect all operations that generate neighboring triangles and propagate them up to the root operations. During this process, triangles that lie completely between the two collapsed vertices are stored at that operation. After searching the influenced operations, we can estimate the effect of the swap on the compressed size of the FVIDs in total linear time as well.

© 2012 The Author(s)
Journal compilation © 2012 The Eurographics Association and Blackwell Publishing Ltd.

5. Runtime Algorithm

Besides classifying the vertices before applying the operations, the adaption is split into first applying all collapse operations. Then the memory for the new vertices and faces is allocated while the data of collapsed ones is freed. Finally, the split operations can be performed by updating the vertices and then the current vertex indices. 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.

buffers	elements	memory (bytes)
active faces	index VBO	$24m$
	FVIDs	$24m$
active vertices	vertex VBO ($\times 2$)	$8km$
	vertex ID ($\times 2$)	$8m$
	split & collapse cache ($\times 2$)	$16m$
	collapse target ($\times 2$)	$8m$
	next split & collapse ($\times 2$)	$16m$ [a] / $24m$ [b]
temporary	vertex count	$4m$
	face count	$4m$
	vertex prefix sum	$4m$
	face prefix sum	$4m$
total	in-core / out-of-core	$(112+8k)m$ / $(120+8k)m$

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).*

5.1. 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 however 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 normalized view direction, $\mathbf{p}$ the vertex position, $\mathbf{n}$ the normal, and α the normal cone angle as discussed in Section 4.2.3. In addition we use occlusion culling for out-of-core rendering 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 adaption. 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. 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.

```
foreach vertex v in parallel do
    update_split_cache(v)
    set_vertex_cnt(v, 1)
    set_face_cnt(v, 0)
    if need_split(v)
        decode_number_of_faces(v)
        set_vertex_cnt(v, 2)
    elif need_collapse(v)
        target = get_target(v)
        target_next = get_target(next(v))
        if prev(v) == target && v != target_next
            set_vertex_cnt(v, 0)
```

Algorithm 1: Parallel vertex state update algorithm.

5.2. 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.

5.3. 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

© 2012 The Author(s)
Journal compilation © 2012 The Eurographics Association and Blackwell Publishing Ltd.

```
foreach vertex v_u in parallel do
    v_t = get_target(v_u)
    if !marked(v_t, split)
        collapse_vertices(v_t, v_u)
    else
        set_vertex_cnt(v_u, 1)
```

Algorithm 2: Parallel edge collapse algorithm.

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 [SHZO07] 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 reorganisation and compaction of the vertex buffer are shown in Algorithm 3.

```
face_sum = prefix_sum(face_cnt)
if need_face_buffer_resize()
    resize_face_buffers()
vertex_sum = prefix_sum(vertex_cnt)
if need_vertex_buffer_resize()
    resize_vertex_buffers()
foreach vertex v in parallel do
    new_pos = vertex_sum[v]
    next_pos = vertex_sum[v+1]
    if new_pos != next_pos
        copy_vertex(v, new_pos)
```

Algorithm 3: Memory management algorithm.

5.4. 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_u and moves v to its new position v_t. 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_t and v_u. 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 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 also important for performance. Algorithm 4 shows the parallel vertex split.

```
compact(splits)
foreach split vertex v in parallel do
    v_u = v + 1
    split_vertex(v, v_u)
    append_faces(v)
```

Algorithm 4: Parallel vertex split algorithm.

5.5. Index Update

The indices of the faces adjacent to split and collapse vertices need to be updated (e.g. $f_{n1} - f_{n6}, f_l$, 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.

```
foreach indices i in parallel do
    ID = get_vertex_id(i)
    FVID = get_final_id(i)
    if ID < FVID
        set_vertex_id(i, ID + 1)
    elif ID > FVID
        set_vertex_id(i, ID - 1)
```

Algorithm 5: Parallel index update.

5.6. 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.

5.7. 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. In addition to temporary memory for

© 2012 The Author(s)
Journal compilation © 2012 The Eurographics Association and Blackwell Publishing Ltd.

model	v_{max}	f_{max}	*orig. (MB)*		*PM (MB)*	BV nodes	bpv	compr. time	*rendered # faces*	*memory (MB)*	*frame time (ms)*	*MTPS*
Dragon	3,609,455	7,218,906	165.2	ic:	43.0 (26.1%)		12.5	9m	1,301,757 (18.0%)	152.3 (92.2%)	6.5	200.3
				ooc:	48.2 (29.1%)	843	14.0	14m	1,023,843 (14.2%)	142.2 (86.1%)	5.9	173.5
Statuette	4,999,996	10,000,000	228.8	ic:	58.7 (25.7%)		12.3	12m	1,479,282 (14.8%)	180.9 (79.1%)	7.2	205.4
				ooc:	64.9 (28.3%)	1136	13.6	19m	1,338,246 (13.4%)	182.4 (79.7%)	7.3	183.3
Lucy	14,027,872	28,055,742	642.2	ic:	152.9 (23.8%)		11.4	37m	2,804,876 (10.0%)	379.9 (59.1%)	11.6	241.8
				ooc:	168.6 (26.3%)	2965	12.6	56m	2,672,566 (9.5%)	397.3 (61.9%)	13.5	198.0
David	28,184,526	56,230,343	1288.6	ic:	303.3 (23.5%)		11.3	1h 10m	2,954,923 (5.2%)	541.8 (42.0%)	12.1	244.2
				ooc:	335.2 (26.3%)	5945	12.5	1h 49m	2,859,350 (5.1%)	498.0 (38.6%)	14.3	200.0
St.Matthew	186,810,938	372,422,615	8537.8	ooc:	2038.0 (23.9%)	36034	11.4	11h 54m	3,412,032 (0.9%)	541.7 (6.3%)	15.3	223.0
Atlas	254,837,027	509,674,062	11665.5	ooc:	3039.7 (26.0%)	53159	12.5	16h 27m	4,025,064 (0.8%)	605,7 (5.2%)	17.8	226.1
Scene	492,469,814	983,601,668	22528.1	ooc:	5694.6 (25.3%)	100082	12.1		5,514,913 (0.6%)	795.8 (3.5%)	39.7	138.9

Table 3: *Progressive meshes examined in our experiments, compression results and rendering statistics.*

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 and the data needs to be loaded from disk. Each BV contains operations with different simplification errors. Based on the maximum error of the operations stored in a volume, we can derive a distance d_n beyond which no split is ever necessary. Then we calculate a priority $p = \frac{d_n}{d_v}$, 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 uploading 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 preformed 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.

6. 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 $n_{max} = 4096$ (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_{max} vertices and f_{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 $v_{max} - v_0$. 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 (bounding box, visibility, offset, and file position) and 5 bytes graphics memory (visibility and offset) are required per BV node. The compression performance is approximately 7 kop/s (in-core) and 4.3 kop/s (out-of-core). In the first case, approximately half of the time is required for the optimization and the other half for the two arithmetic coding runs. In the second, three additional arithmetic coding runs were required until convergence for all models.

Table 3 also 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 approx-

© 2012 The Author(s)
Journal compilation © 2012 The Eurographics Association and Blackwell Publishing Ltd.

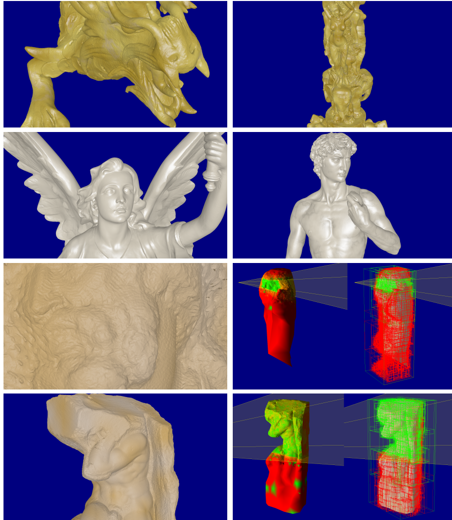

Figure 6: *Renderings of view-dependently refined meshes. The external views show the view frustum (yellow), LOD (red: low; green: high), and the nodes used for occlusion culling (red: occluded; green: visible).*

imately 10% lower. The culling overhead is low 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. 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 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.

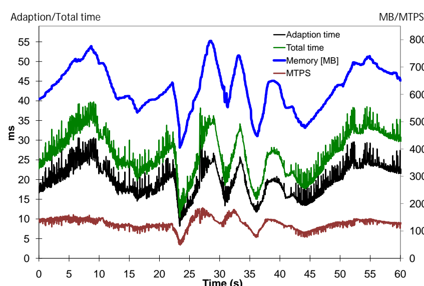

Figure 7: *Timings, memory consumption and triangle rate for the scene using a pre-recorded camera path.*

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.

6.1. Discussion

In Table 4 we compare our algorithm with three different types of approaches. We can render 180/140 MTPS (in-core/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).

Algorithm	relative #triangles	rendered MTPS	generated MTPS	bpv
Our ic	1	180	23	11-13
Our ooc	0.9	140 (155)	13 (14.44)	11-14
Compression approaches				
[CKLL09]* ooc	VFC only	600 (n.a.)	0.07 (n.a.)	1-3
[CH09]* ic	VFC only	600 (n.a.)	0.4 (n.a.)	2-4
[DJCM09]* ooc	>10	600 (<60)	0.04 (<0.004)	1-3
[JGA09]* ooc	>10	600 (<60)	0.3 (<0.03)	2-4
HLOD approaches				
[GBK03] ooc	∼5.5	400 (73)	4.6 (0.84)	26
[CGG*04] ooc	∼6.5	500 (76)	4.6 (0.71)	33
[GM05] ooc	>10	190 (<19)	3.4 (<0.34)	61
Progressive Meshes				
[YSGM04] ooc	∼2.5	90 (36)	0.015 (0.006)	79
[HSH09] ic	∼1.05	30 (28)	0.8 (0.76)	69
[DMG10a] ic	∼1.2	80 (66)	4.4 (3.66)	22
[DMG10b] ooc	∼1.1	60 (55)	4.0 (3.63)	24

Table 4: *Comparison of triangle rate and memory consumption with previous approaches. The relative performance is shown in parenthesis. Results marked with * are results of the original authors scaled to the performance of our system, while all other were measured.*

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

© 2012 The Author(s)
Journal compilation © 2012 The Eurographics Association and Blackwell Publishing Ltd.

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 that 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 [CGG*04] or 49% of our approach. In summary, our approach renders approximately twice as fast and can react ten times quicker to view changes.

View-dependent adaption algorithms can better compete with our approach regarding the number of primitives. 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 most 3.66 MTPS which is 25% of our method. Compared to these algorithms, our method can render the same views three times faster and adapt the LOD four times faster.

6.2. Analysis

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.

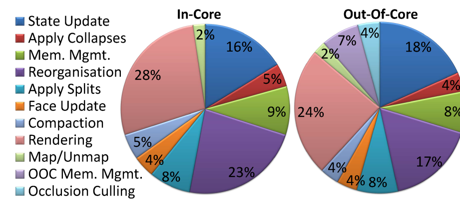

Figure 8: *Relative time of the adaption steps compared to rendering.*

7. 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 that the reorganisation of the vertices is rather expensive. An acceleration

© 2012 The Author(s)
Journal compilation © 2012 The Eurographics Association and Blackwell Publishing Ltd.

or prevention of this step would significantly increase the performance. Of course, the efficiency of the method soley 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

[AAR05] ALREGIB G., ALTUNBASAK Y., ROSSIGNAC J.: Error-resilient transmission of 3d models. *ACM Trans. Graph. 24*, 2 (2005), 182–208. 2

[BGB*05] BORGEAT L., GODIN G., BLAIS F., MASSICOTTE P., LAHANIER C.: Gold: interactive display of huge colored and textured models. *ACM Trans. Graph. 24*, 3 (2005), 869–877. 2

[CCG*01] CUNNIFF R., CRAIGHEAD M., GINSBURG D., LEFEBVRE K., LICEA-KANE B., TRIANTOS N.: *ARB occlusion query*. Tech. rep., NVIDIA and ATI, 2001. 7

[CGG*04] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.: Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Trans. Graph. 23*, 3 (2004), 796–803. 2, 10, 11

[CH09] COURBET C., HUDELOT C.: Random accessible hierarchical mesh compression for interactive visualization. In *Proceedings of the Symposium on Geometry Processing* (2009), pp. 1311–1318. 2, 10

[CKL*04] CHOE S., KIM J., LEE H., LEE S., SEIDEL H. P.: Mesh compression with random accessibility. In *Israel-Korea Bi-National Conf* (2004), pp. 81–86. 2

[CKLL09] CHOE S., KIM J., LEE H., LEE S.: Random accessible mesh compression using mesh chartification. *IEEE Transactions on Visualization and Computer Graphics 15*, 1 (2009), 160–173. 2, 10

[DGGP05] DIAZ-GUTIERREZ P., GOPI M., PAJAROLA R.: Hierarchyless simplification, stripification and compression of triangulated two-manifolds. *Computer Graphics Forum 24*, 3 (2005), 457–467. 2

[DJCM09] DU Z., JAROMERSKY P., CHIANG Y.-J., MEMON N.: Out-of-core progressive lossless compression and selective decompression of large triangle meshes. In *Proceedings of the 2009 Data Compression Conference* (2009), pp. 420–429. 2, 10

[DMG10a] DERZAPF E., MENZEL N., GUTHE M.: Parallel view-dependent refinement of compact progressive meshes. In *Eurographics Symposium on Parallel Graphics and Visualization* (2010), pp. 53–62. 2, 3, 8, 10, 11

[DMG10b] DERZAPF E., MENZEL N., GUTHE M.: Parallel view-dependent out-of-core progressive meshes. In *Vision, Modeling, and Visualization* (2010), pp. 53–62. 2, 3, 10

[EMB01] ERIKSON C., MANOCHA D., BAXTER III W. V.: Hlods for faster display of large static and dynamic environments. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics* (2001), pp. 111–120. 2

[GBBK04] GUTHE M., BORODIN P., BALÁZS Á., KLEIN R.: Real-time appearance preserving out-of-core rendering with shadows. In *Rendering Techniques 2004 (Proceedings of Eurographics Symposium on Rendering)* (2004), pp. 69–79. 4

[GBK03] GUTHE M., BORODIN P., KLEIN R.: Efficient view-dependent out-of-core visualization. In *Proceedings of the 4th International Conference on Virtual Reality and its Applications in Industry (VRAI '2003)* (2003), pp. 428–438. 2, 10, 11

[GM05] GOBBETTI E., MARTON F.: Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Trans. Graph. 24*, 3 (2005), 878–885. 2, 10, 11, 12

[Hop96] HOPPE H.: Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), pp. 99–108. 2, 3

[Hop97] HOPPE H.: View-dependent refinement of progressive meshes. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), pp. 189–198. 2, 3, 4

[Hop98] HOPPE H.: Efficient implementation of progressive meshes. *Computers & Graphics 22*, 1 (1998), 27–36. 2

[HSH09] HU L., SANDER P. V., HOPPE H.: Parallel view-dependent refinement of progressive meshes. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games* (2009), pp. 169–176. 2, 3, 10, 11

[JGA09] JAMIN C., GANDOIN P.-M., AKKOUCHE S.: Technical section: Chumi viewer: Compressive huge mesh interactive viewer. *Comput. Graph. 33*, 4 (2009), 542–553. 2, 10

[KCL06] KIM J., CHOE S., LEE S.: Multiresolution random accessible mesh compression. *Computer Graphics Forum 25*, 3 (2006), 323–332. 2

[KL01] KIM J., LEE S.: Truly selective refinement of progressive meshes. In *Graphics Interface 2001* (2001), pp. 101–110. 2, 6

[MBH*01] MEISSNER M., BARTZ D., HÜTTNER T., MÜLLER G., EINIGHAMMER J.: Generation of Decomposition Hierarchies for Efficient Occlusion Culling of Large Polygonal Models. In *Vision, Modeling, and Visualization* (2001), pp. 225–232. 5

[Paj01] PAJAROLA R.: Fastmesh: Efficient view-dependent meshing. In *9th Pacific Conference on Computer Graphics and Applications* (2001), pp. 20–30. 2

[PD04] PAJAROLA R., DECORO C.: Efficient implementation of real-time view-dependent multiresolution meshing. *IEEE Transactions on Visualization and Computer Graphics 10*, 3 (2004), 353–368. 2

[PR00] PAJAROLA R., ROSSIGNAC J.: Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics 6*, 1 (2000), 79–93. 2

[Sai04] SAID A.: *Introduction to Arithmetic Coding Theory and Practice*. Tech. Rep. HPL-2004-76, Hewlett-Packard Laboratories, 2004. 5

[SHZO07] SENGUPTA S., HARRIS M., ZHANG Y., OWENS J. D.: Scan primitives for gpu computing. In *Graphics Hardware 2007* (2007), pp. 97–106. 8

[SM06] SANDER P. V., MITCHELL J. L.: Progressive buffers: view-dependent geometry and texture lod rendering. In *ACM SIGGRAPH 2006 Courses* (2006), pp. 1–18. 2

[TR99] TAUBIN G., ROSSIGNAC J.: 3d geometry compression. In *SIGGRAPH '99: Course Notes*. ACM, Los Angeles, Aug. 1999. Course 22. 2

[XV96] XIA J. C., VARSHNEY A.: Dynamic view-dependent simplification for polygonal models. In *VIS '96: Proceedings of the 7th conference on Visualization '96* (1996), p. 327 ff. 2, 3

[YL07] YOON S.-E., LINDSTROM P.: Random-accessible compressed triangle meshes. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1536–1543. 2

[YSGM04] YOON S.-E., SALOMON B., GAYLE R., MANOCHA D.: Quick-vdr: Interactive view-dependent rendering of massive models. In *VIS '04: Proceedings of the conference on Visualization '04* (2004), pp. 131–138. 2, 10, 11

© 2012 The Author(s)
Journal compilation © 2012 The Eurographics Association and Blackwell Publishing Ltd.