Vision, Modeling, and Visualization (2010)

Parallel View-Dependent Out-of-Core Progressive Meshes

Evgenij Derzapf[1], Nicolas Menzel[1], and Michael Guthe[1]

[1]Graphics and Multimedia Group, FB12, Philipps-Universität Marburg, Germany

Abstract

The complexity of polygonal models is growing faster than the ability of graphics hardware to render them in real-time. If a scene contains many models and textures, it is often also not possible to store the entire geometry in the graphics memory. A common way to deal with such models is to use multiple levels of detail (LODs), which represent a model at different complexity levels. With view-dependent progressive meshes it is possible to render complex models in real time, but the whole progressive model must fit into graphics memory. To solve this problem out-of-core algorithms have to be used to load mesh data from external data devices. Hierarchical level of detail (HLOD) algorithms are a common solution for this problem, but they have numerous disadvantages. In this paper, we combine the advantages of view-dependent progressive meshes and HLODs by proposing a new algorithm for real-time view-dependent rendering of huge models. Using a spatial hierarchy we extend parallel view-dependent progressive meshes to support out-of-core rendering. In addition we present a compact data structure for progressive meshes, optimized for parallel GPU-processing and out-of-core memory management.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.3]: Picture/Image GenerationDisplay algorithms Computer Graphics [I.3.5]: Computational Geometry and Object ModelingCurve, surface, solid, and object representations Computer Graphics [I.3.1]: Hardware architectureParallel processing

1. Introduction

The need for high quality polygonal models in interactive applications is constantly increasing. Despite the enormous processing power of the graphics processor (GPU), highly detailed models cannot be rendered in real-time. Often they even do not fit into graphics memory since a scene may contain several of them. To solve this problem the number of triangles must be reduced.

One possibility is to use static or dynamic levels of detail (LODs). Static LODs are easy to use, but suffer from visible transitions, so called popping artifacts, when switching between levels. Additionally, the memory overhead compared to an ordinary mesh is typically about 50%. Dynamic LODs use a continuous sequence of the simplification operations. Therefore popping artifacts are less visible. Another advantage is that the dynamic LODs can be extended to view-dependent LOD and thus do not use more triangles than necessary. Recently proposed parallel adaption algorithms running on the GPU make it possible to render large progressive meshes in real-time. One limitation of these algorithms is however, that the whole model must fit into graphics memory. This constraint does not allow for the rendering of huge models exceeding the amount of available graphics memory.

Hierarchical levels of detail (HLODs) solve the memory problem by partitioning the models using a spatial hierarchy and generating a level of detail for each node. If static LODs are used, a fully view-dependent adaption is not possible. The frame rate fluctuates and can drop severely when many nodes need to be transferred between subsequent frames. Using dynamic LODs for each node improves the adaption efficiency, but the frame time still significantly increases when different nodes are required than in the previous frame. In addition, the cuts between the spatial hierarchy nodes complicate the simplification which also slightly increases the number of rendered triangles.

In this paper we try to solve both problems of HLODs by expanding the approach of [DMG10] for out-of-core progressive meshes. Our main contributions are:

- A view-dependent out-of-core progressive mesh data structure which can also be used for occlusion culling.

DOI: 10.2312/PE/VMV/VMV10/025-032

- No simplification constraints between nodes of the spatial hierarchy.
- A massively parallel adaption algorithm with stable, real-time frame rates.

The remainder of this paper is structured as follows: in Section 2 we give an overview of existing techniques. In Section 3 the proposed out-of-core data structure is explained in detail. In Section 4 we then introduce the out-of-core memory management and the modifications of the parallel adaption algorithm for real-time rendering. Finally we evaluate our approach in Section 5.

2. Related Work

View-dependent simplification has been an active field of research over the last two decades. Hoppe introduced progressive meshes (PM) that smoothly interpolate between different levels of detail [Hop96]. Depending on the view position and distance, a sequence of split- or collapse operations can be performed for each vertex to generate a view-dependent simplification. The inter-dependency of split operations can either be encoded explicitly [XV96] or implicitly [Hop97]. Hoppe later optimized the data structures for the operations and improved the performance of the refinement algorithm [Hop98]. Pajarola and Rossignac [PR00] introduced compressed progressive meshes, where the input mesh is simplified in *batches*. A batch is created by selecting the first 11% of non-adjacent edges from the priority queue. All of these edge-collapses have to be performed in parallel. This allows for a very compact coding, but view-dependent adaption is not possible. 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. In total this requires 24 additional bytes per vertex of the adapted mesh. Hu et al. [HSH09] proposed a parallel adaption algorithm for progressive meshes. They introduced a 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 dependency that needs additional memory and that only half-edge collapses are supported. In recent progressive mesh representations however the edge collapse operation with optimized vertex position and attributes is used. The advantage over the half-edge collapse is that it produces simplifications with significantly higher quality and thus less triangles for a given accuracy. In addition to that, the rendering performance is also degraded by using a single vertex array containing all vertices and attributes of the original mesh. A more compact progressive meshes data structure for parallel adaption was proposed by Derzapf et al. [DMG10]. They introduced a random access compact data structure based on Hoppe's original refiment algorithm [Hop98] with a massively parallel adaption algorithm for real-time rendering of large models.

For huge models none of these approaches is suitable since they all require the complete progressive mesh to fit into main and/or graphics memory. For models exceeding the available amount of memory, hierarchical levels of detail (HLODs) were proposed which only require the currently needed LODs to be kept in memory. In addition, they allow a LOD selection that is performed per node. The first approach has been introduced by Erikson et al. [EMB01]. The problem of this technique is that no simplification along the cuts between hierarchy nodes is possible without introducing visible gaps. Constraining the simplification however leads to a significant increase of the number of primitives and thus low frame rates. Guthe et al. [GBK04] solved this problem by using an unconstrained simplification of the parts. The gaps are then filled during rendering using line strips. Cignoni et al. proposed a different solution by creating alternating diamond shaped hierarchies [CGG*04]. This way the triangles along a node boundary can be simplified at coarse levels. Finally, Borgeat et al. proposed to use geomorphing to simplify the triangles along node boundaries during rendering [BGB*05]. Unfortunately, the transform performance is approximately halfed this way such that the previous two approaches are faster in almost any cases. On the other hand, popping artifacts are drastically reduced due to smoother LOD transitions. 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 of Yoon et al. [YSGM04] was the first to combine HLOD and progressive meshes. For each node, a progressive mesh is stored to allow for smoother LOD transition, but still fully view-dependent adaption is not possible, due to the use of view-independent adaption.

3. Out-of-Core View-Dependent Progressive Mesh

The proposed view-dependent out-of-core refinement algorithm is based on the in-core algorithm of Derzapf et al. [DMG10] which we briefly describe before discussing our modifications. After building a progressive mesh, a view-dependent reconstruction is generated by performing only those split operations necessary for the current view. During this process the local ordering of operations needs to be preserved. This leads to the dependency rules formulated by Hoppe [Hop97]:

- The ordering of operations applied to a single vertex must be preserved.
- A split can only be applied if the next split operation of each neighboring vertex was generated earlier during simplification.
- Edge collapse operations are only legal if the next collapse of each neighboring vertex was created later.

The first dependency rule can be efficiently encoded in a forest of binary trees as shown in Figure 1. For each vertex of the base mesh, a binary tree is constructed.

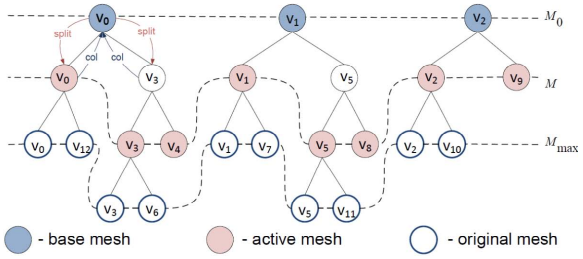

Figure 1: *Split/collapse operation hierarchy represented as a forest of binary trees.*

The operation indices cannot be used to preserve the local ordering because their sequence is not preserved. Therefore the neighborhood dependency is encoded explicitly for each operation. For compact encoding the main idea is to construct consecutive independent sets:

1. Start with base mesh $M_i = M_0$.
2. Store all currently possible operations in level i.
3. Perform all operations of level i on M_i to construct M_{i+1}.
4. Increment i and continue with the second step until no operations are left.

The split level is then stored for each operation and the local ordering is preserved if only the vertex with lowest level in each face is split. Accordingly, only the vertex with the highest level in each face can be collapsed.

A compact data structure is used to store the topology and attribute modifications of each operation. The up to two new faces created by a split operation are defined by two neighbor vertices v_l and v_r. These are stored by first defining an ordering on the vertex neighborhood. For this purpose an operation index i is used and the index of each vertex is either its base mesh index or the operation index plus the number of base mesh vertices. Then, v_l and v_r are encoded by their rank in the ordered sequence of neighbor vertices. In the example shown in Figure 2, the ranks are 0 and 4. In addition, the connectivity modifications are encoded using an ordering of the neighbor triangles. The triangle index is calculated analogously to the vertex index. A bit flag is then set to one for each triangle adjacent to v_u after the split. In Figure 2 the modified faces and the resulting bit vector are shown.

Both, the ranks of v_l and v_r, and the bit flag are encoded with fixed length variables. The valence of the vertices is restricted to 15 and the number of neighbor faces to 16. Thus 8 bits are sufficient for the new faces and 16 bits for the connectivity modifications. Vertices with higher valence are not collapsed during simplification. As the collapse of two neighbor vertices reduces their valence they will be collapsed at a later stage if suitable.

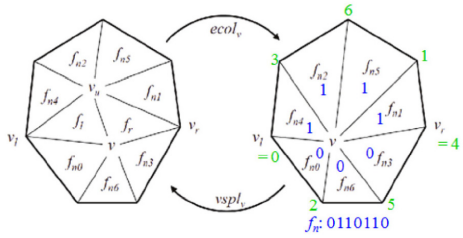

Figure 2: *Example of topology encoding.*

In addition to the topology, the attribute differences (position, normal, texture coordinates etc.) of v and v_u are stored. First, each attribute is scaled such that the variances are confined to the same range. Then the differences are scaled to the range $[-1,1]$ per operation and quantized to n bits using a cubic function. To reduce the quantization error, dummy splits are introduced if necessary. For the refinement criteria, the simplification distance, normal cone angle, and the ratio of geometric to attribut error are stored.

The adaption algorithm additionally maintains a few dynamic data structures. They store the split hierarchy as well as a dynamically updated vertex buffer and index buffer. The main data structures required for rendering are the vertex buffer, which contains the position and attributes of the adapted vertex and the index buffer that contains the connectivity. Both are stored as vertex buffer objects (VBOs) and are therefore separated from all other data. Together they form the indexed face set that is used for rendering. To store the prefix sum, neighborhood and collapse information additional memory is required.

3.1. Spatial Operation Hierarchy

For out-of-core rendering we now build a spatial hierarchy of split/collapse operations. The advantage over previous approaches is that this way no special boundary constraints between adjacent hierarchy nodes are introduced. The hierarchy serves two purposes: first of all, the operations should be grouped such that those which are likely to be performed simultaneously or successively are stored together. In addition, it is also to 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.

For occlusion culling, Meißner et al. [MBH*01] proposed a simple heuristic to construct efficient kd-tree hierarchies for triangle meshes using a greedy algorithm. It is based on the idea to minimize the total area of all bounding boxes in a hierarchy with a fixed maximum number of triangles n_{max} per leaf node. Starting from the root node, an optimal partition with respect to an estimated total bounding box area

is performed. The area A is estimated as:

$$A \approx A_l \left(1+\log_2 \left\lceil \frac{n_l}{n_{max}} \right\rceil\right) + A_r \left(1+\log_2 \left\lceil \frac{n_r}{n_{max}} \right\rceil\right),$$

where A_l and A_r are the bounding box areas of left and right child node and n_l and n_r the respective number of triangles. Finding the optimal split is performed by sorting the triangles in x-, y-, and z-direction and then calculating the estimated area for all ordered partitions.

In contrast to a hierarchy for triangles, we do not only store operations at leaf level but also at inner nodes to reconstruct coarse approximations of the model. When processing a node, we first need to determine the operations that are stored in the current node. Then their subtrees are partitioned into the child nodes. Finding the directly stored operations is rather straightforward as those with the highest simplification error are required first. Since we do not want to skip a node for two successive refinements of a vertex, we first add all operations which are referenced from the previous node. For the root node, these are the operations of the base mesh. When the operations are stored in the current node their next operations are partitioned into the child nodes. This way a complete operation subtree is stored in a single subtree. Due to storing operations not only at the leaf nodes, the estimated area is slightly different than in the approach of Meißner et al.:

$$A \approx A_l \log_2 \left\lceil \frac{n_l}{n_{max}+1} \right\rceil + A_r \log_2 \left\lceil \frac{n_r}{n_{max}+1} \right\rceil,$$

where n_* now is the number of operations.

3.2. Tree Structure

The child relations of the operation tree are encoded by storing the operations relative to the current node. The relative indices are stored as c_l for left and c_r for right child operation. Each operation is accessed using two values: the node index i_n and the local index of the operation in this node i_l. A unique global index i which is required later can be computed as:

$$i = n_{max} i_n + i_l$$

The node and local index of the child nodes are calculated as follows:

$$i_{n,l/r} = \begin{cases} c_{l,r} < n_{max} & : \; i_n \\ n_{max} \leq c_{l/r} < 2n_{max} & : \; left\_child(i_n) \\ c_{l,r} \geq 2n_{max} & : \; right\_child(i_n) \\ \text{else} & : \; \text{no operation} \end{cases}$$

$$i_{l,l/r} = c_{l/r} \mod n_{max},$$

where i_n is the node of the current operation.

To lessen the constraint of 256 split levels as in the in-core case, we use 10 bits instead of only a single byte for the level. Together with 11 bits for each of the child indices, the ordering and dependency are encoded in four bytes per operation. Using the index coding described above, up to 682 operations can be stored per node.

3.3. Data Structures

In contrast to the in-core algorithm we only store the currently required parts of the static data in graphics memory. By using these data structures, the view-dependent mesh can be refined and rendered in real-time with minimal memory requirements. Table 1 shows the static and dynamic data structures required to maintain and update the vertex and index buffers in detail. Differences to the in-core algorithm are marked in blue. Since the complete algorithm runs on the GPU, all the relevant data is stored in graphics memory and the main memory consumption is minimal. Besides temporary memory for loading, only the mapping of nodes to memory positions and file offsets required for out-of-core management are stored in main memory.

buffers	elements	memory (bytes)
static structures		
nodes	nodes	$8o$
	offset	$4o$
	visibility	$1o$
operations	tree structure	$3n$
	dependency	$1n$
	ref. criteria	$3n$
	topology	$3n$
	quant. delta	$2kn$
	delta scale	$2n$
dynamic structures		
active faces	index VBO	$24m$
	triangle ID	$8m$
active vertices	vertex VBO	$4km$
	vertex ID	$4m$
	node ID	$4m$
	local ID	$2m$
	next collapse	$4m$
	v_{state}	$1m$
collapse tree	node ID	$4m$
	local ID	$2m$
	prev. collapse	$4m$
	v_u	$4m$
temporary	prefix sum	$24m$
	neighb. size	$1m$
	neighb. index	$16m$
total		$13o+(12+2k)n$ $+(102+4k)m$

Table 1: *Elements of the data structure. o, k, n, and m are the number of operation nodes, attributes, operations in graphics memory, and vertices of the adapted mesh.*

As the operations are referenced using a node and a local ID, we store these instead of the next split or collapse operation index. To apply the operations we also need to store the operation hierarchy. To determine which nodes are loaded, we additionally store pointers of the operation nodes in device memory. For occlusion culling we additionally store

the visibility of the nodes. In summary, the out-of-core data management requires 1.875 additional bytes for each operation, 4 bytes for each active vertex and 13 bytes for each node. As each node stores many operations, their number is rather low. In total we require $1.875n + 4m + 13o$ bytes additional memory for the out-of-core data management compared to [DMG10]. On the other hand, the memory requirements for the static data are drastically reduced as only a subset of the operations is kept in graphics memory.

4. Runtime Algorithm

For the dynamic data structures we use the same block based memory management as Derzapf at al. [DMG10], but additionally introduce an out-of-core management for the static data structures.

4.1. Out-of-Core Memory Management

The operations and attributes are subdivided into the nodes described above and stored in a large file. Then only the currently necessary nodes are loaded into graphics memory. A node is required when at least one of the active vertices or collapses has a reference to it. A split operation can create vertices that reference nodes not available in graphics memory. In this case, we load this node from disk into graphics memory. Before loading the currently required nodes we first remove all inactive ones.

Since accessing the hard disk is a severe bottleneck, loading nodes 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. To prevent strong frame rate fluctuations we only load a maximum number of l_{max} nodes per frame, because the memory transfer to the graphics card is relatively expensive. This approach slightly slows down the adaption, but guarantees stable frame rates. If the number of required nodes exceeds l_{max} we load the ones that were requested the highest number of consecutive frames. This has the advantage that the model is uniformly adapted and no LOD starvation can occur.

The operation memory is shared for all progressive models of the scene and a maximum number of nodes o_{max} kept in graphics memory is specified. If the number of totally required nodes exceeds o_{max}, a global level of detail scaling factor is used to coarsen all models until enough memory is available. If later more free node are available the factor is gradually reduced again.

4.2. Parallel Adaption Algorithm

In order to optimally exploit the parallel architecture of the GPU, the adaption algorithm is subdivided into several consecutive steps. Each step is then performed in parallel. The algorithm is based on 4 bit-states to encode possible and necessary operations and 2 temporary states for collapsed and removed vertices. It is composed of the five following main stages, where the first three steps are similar to the in-core variant.

In the first step we update the state of the vertices, where the refinement criteria determine which vertices need to be split and which can be collapsed. In addition to the original refinement criteria – i.e. view-frustum culling, back-face culling and projected error – we also use occlusion culling to further reduce the number of active vertices. The occlusion culling is based on the operation nodes whose visibility is determined after rendering. If the next split or collapse of a vertex is in an occluded node, the vertex itself is marked as invisible. After checking if a vertex is occluded, we test the original refinement criteria to determine which operations need to be performed. When the necessary operations are determined, we need to force splits of neighbor vertices due to the face dependencies and remove of all impossible operations. Since not all static data is available in graphics memory, we have to check whether the required operation and the operations of its neighbor vertices are available. If this is not the case, we need to delay the operation.

In a second step we apply all remaining operations. For this purpose we first collect the neighborhood of the split vertices and then perform the split operations. Applying a split operation includes generating the new vertex v_u, moving the vertex v and creating two new faces f_l and f_r. After the split operations the collapse operations are performed. For each collapse vertex v the corresponding vertex v_u is removed, the faces are updated and degenerate ones are removed.

The third step of the adaption is the compaction of buffers where elements have been removed. These buffers are the active vertices (including the vertex VBO), active faces (with the index VBO), and collapse operations. To improve rendering performance, the index VBO is periodically sorted to exploit vertex caching.

Finally we determine which of the nodes are occluded to use this information for the next frame. For this purpose we use hardware occlusion queries [CCG*01] and render the bounding boxes of the nodes. The query is performed after rendering the complete scene and the result is fetched before the next parallel adaption. This way the visibility is lagging one frame behind which is not problematic as it is only used for LOD selection, but not for rendering. Regardless of the query results we always render the complete adapted mesh. This approach is very efficient because the number of the active nodes 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.

4.3. Prefetching

When the user moves through the screen, the visible portions and the required LOD of the object change. This results in a

continuous change of the nodes currently required in device memory. Due to the high latency of the hard disk, loading out-of-core data results in a visible delay of the mesh refinement. We solve this problem with a two-step prefetching algorithm: First, we do not only load the currently required nodes, but also their direct children if enough space is available. In addition, we extrapolate the current camera movement and enlarge the view frustum to contain all frusta of the subsequent 10 frames. This results in a dynamic adaption of the view frustum, which means that we pre-refine parts of the model before they become visible.

Note that the prefetching algorithm is memory sensitive: it only works if enough memory is available on the device. This is achieved by assigning a low priority to all prefetching candidates, so these nodes are only processed after all other ones are loaded.

5. Results

Our test system is built of a 2.4 GHz Intel Core2 Duo CPU with 2 GB of DDR2 main memory, 16 lanes PCIe slot, and a GeForce GTX 480 where we use the OpenGL API for rendering. The out-of-core progressive meshes are stored on a Seagate SATAII hard disk with 7200 rpm and 32 MB cache. The access time is 8.5 ms and approximately 100 MB can be read per second. We use a resolution of 1920×1080 in all experiments. Table 2 gives an overview of the progressive meshes we used in our experiments.

model	v_0	f_0	*# ops.*	*# dummy ops.*	lvl.	nodes
Asian Dragon	40	19	3,612,383	2968 (0.08%)	213	5507
David	659	1416	3,615,968	2529 (0.07%)	276	5536
Statuette	16	40	5,014,234	14254 (0.28%)	212	7678
Lucy	15	29	14,040,204	24681 (0.17%)	268	21421
Sponza scene	730	1504	26,282,789	44432 (1.69%)	276	40142

Table 2: *Progressive meshes used in our experiments, number of base mesh vertices, base mesh faces, operations, added dummy split operations, maximum split level and nodes.*

All models use position and normal as vertex attributes only (i.e. $k = 6$ attributes). The resulting file sizes – compared to an indexed face set – are listed in Table 3. The file size reduction is almost identical for all models and is approximately 50% due to identical number of attributes. Note that HLODs and Quick-VDR both need 50% to 80% more disk storage than an indexed face set.

model	v_{max}	f_{max}	*mem. IFS*	*mem. PM*
Asian Dragon	3,609,455	7,218,906	165.2MB	82.9MB (50.2%)
David	3,614,098	7,227,031	165.4MB	83.0MB (50.2%)
Statuette	4,999,996	10,000,000	228.8MB	115.0MB (50.2%)
Lucy	14,027,872	28,055,742	642.2MB	322.1MB (50.2%)
Sponza scene	26,251,421	52,501,679	1201.6MB	603.0MB (50.2%)

Table 3: *Number of original mesh vertices and faces and comparison of the static data (PM) to an indexed face set (IFS).*

Table 4 shows the number of rendered faces, the total rendering time, and the memory consumption for the views shown in Figure 3 and the Sponza scene in the accompanied video, where the numbers are taken from the most complex frame. For the Asian Dragon, David and Statuette we used 4096 nodes (64.0 MB). For the Lucy and Sponza scene we used 6144 nodes (95.9 MB), whereas each node contains 682 operations. During rendering, the dynamic data structures consume additional memory. For all models, the total amount of graphics memory nevertheless stays below that of an indexed face set. The savings to an IFS are about 30% for the medium sized and up to 82% for larger models. This

model	*rendered # faces*	*memory (MB)*	*total frame time (ms)*
Asian Dragon	853,667 (11.8%)	112.07 (67.9%)	11.0 (89.7%)
David	916,044 (12.6%)	115.48 (69.8%)	11.8 (87.0%)
Statuette	1,450,698 (14.5%)	144.05 (62.9%)	18.7 (88.0%)
Lucy	2,125,849 (7.6%)	224.59 (34.9%)	24.9 (20.3%)
Sponza scene	2,116,626 (4.0%)	220.65 (18.3%)	33.2 (19.8%)

Table 4: *Memory consumption and total rendering time of the different models. The ratio compared to rendering an indexed face set of the original model is shown in parenthesis.*

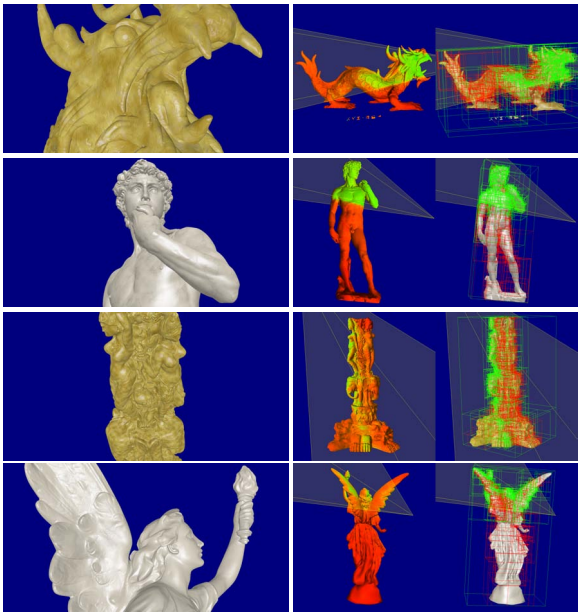

Figure 3: *Renderings of view-dependently refined meshes. The images on the left show the models as rendered from the point of view. In the middle external views with the view frustum (yellow) are shown. The color coding depicts the level of detail, where red is low LOD and green high. The image on the right shows the nodes used for occlusion culling. Occluded nodes are shown in red and visible ones in green.*

allows to safe up to 1 GB of graphics memory for the sponza scene and over 80% of the rendering time. Figure 3 also shows the coarsening of the back faces, faces outside the view frustum and occluded ones.

Table 5 shows the number of rendered faces, the total rendering time and the memory consumption of the in-core algorithm for identical views. Comparisons to David, Lucy and Sponza scene are not possible, because the split level is too high. With the occlusion-culling we reduce the number of active faces significantly (over 50%). Additionally, we need about 44% less memory and up to 29% less rendering time for identical views.

model	*rendered # faces*	*memory (MB)*	*total frame time (ms)*
Asian Dragon	1,740,953 (+103.9%)	174.02 (+55.3%)	15.4 (+40.0%)
Statuette	2,536,238 (+74.8%)	246.25 (+70.9%)	21.1 (+12.9%)

Table 5: *Memory consumption and total rendering time of the approach of Derzapf et al. [DMG10]. The ratio compared to the values in the Table 4 for identical views.*

Compared to static hierarchical LODs (HLODs) [GBBK04] using the same error measure, the number of primitives is reduced by a factor of 3 to 5 and the frame rate improves by a factor between two and three. On our test system, for the David model and identical view, the HLOD rendering requires approximately 4.1 M faces, 175 MB graphics memory and 56 ms (17.8 fps) for rendering. When the LOD switches, the frame rate can even drop below 10 fps. This is due to the fact that the LODs can only be selected based on the viewing distance as only a single mesh is stored per node. The performance of Quick-VDR [YSGM04] is only slightly better than static HLODs while the number of triangles is approximately halved. On our test system, for the David model and identical view, Quick-VDR requires approximately 2.1 M faces, 200 MB graphics memory, 600 MB main memory and achieves 25-30 fps. Additionally, the adaption of the Quick-VBR is very slow. It achieves approximately 60 k operations per second on our test system whereas our approach achieves over 1.2 M operations per second.

Figure 4 shows the adaption and rendering time, the memory consumption, and the number of active faces for a pre-recorded movement for the Asian Dragon. The frame time and consumed graphics memory is always significantly lower than required by the in-core algorithm, because we reduce the number of active faces with the occlusion-culling significantly (up to 60%). In addition to the reduced frame time we require up to 50% less memory.

Figure 5 shows the adaption and rendering time together with the memory consumption and the number of active faces for a pre-recorded movement through the Sponza scene. The consumed graphics memory is always below 220 MB. The frame rate is constantly about 50 frames per second, with some drops down to 40 fps. The number of faces

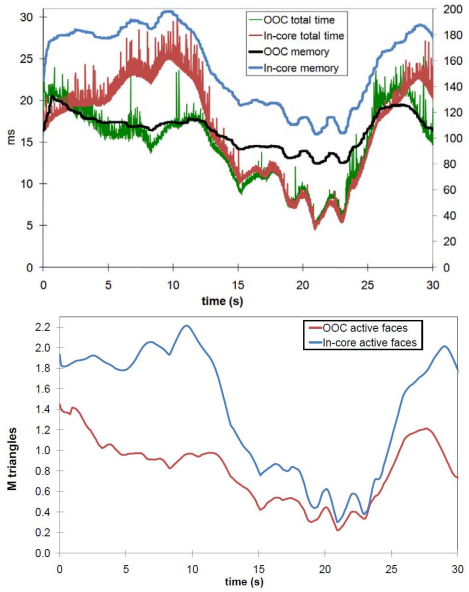

Figure 4: *Comparison of timings, memory consumption and number of active faces of our approach (OOC) with in-core algorithm for the Asian Dragon with a pre-recorded camera path.*

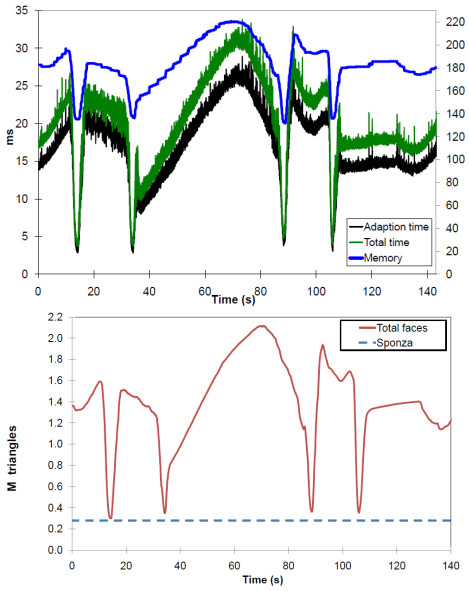

Figure 5: *Timings, memory consumption and number of faces for the Sponza scene with a pre-recorded camera path.*

during the walk through slightly exceeds 2 millions in some situations. Our approach quickly reacts to changes of the view direction with fast adaption of the scene complexity.

Since the rendering performance is identical to render-

ing a static model with the same number of triangles, our method needs approximately five times as long as rendering a static mesh. Considering that we already cut down the vertices significantly due to the simplification of back-faces, faces outside the view frustum and occluded faces. We can conclude that our method will almost always be faster than rendering an indexed face set of the original model. While this even holds for rather coarse models, the performance gain increases with the complexity of the original mesh. Due to the time required for the pixel shaders, the speedup is of course not linear with the reduction.

6. Conclusion and Limitations

We have proposed an out-of-core progressive mesh representation that was specifically developed for parallel refinement on modern graphics hardware. Our algorithm extends the approach of Derzapf et al. [DMG10] with out-of-core data management and is the first out-of-core approach that is completely based on view-dependent progressive meshes. The dependency and operation coding are modified for large models and the dynamic data structures, static data structures, and first three steps of the algorithm are extended for out-of-core data management. In addition we propose a spatial hierarchy for the operations that is also used for occlusion culling.

We remove more than two thirds of the vertices by view-dependent simplification compared to HLODs. Due to the much finer granularity, this reduction is practically always surpassed. Compared to HLODs we thus more than double the frame rate. In addition, the frame rate drop when changing the LOD is insignificant whereas for HLODs the frame rate can drop by a factor of more than two.

One limitation of our algorithm is that an additional memory reduction would be possible by using a generalized triangle strip. Another, probably more severe limitation is that some splits are postponed several frames as they are waiting for others to be applied before them. Although this is only problematic for fast panning over the model, a less restrictive dependency scheme would be desirable.

References

[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. http://oss.sgi.com/projects/ogl-sample/registry/ARB/occlusion_query.txt (23.03.2010). 5

[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

[DMG10] DERZAPF E., MENZEL N., GUTHE M.: Parallel view-dependent refinement of compact progressive meshes. In *Eurographics Symposium on Parallel Graphics and Visualization* (2010). 1, 2, 5, 7, 8

[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* (New York, NY, USA, 2001), ACM, 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), Keller A., Jensen H. W., (Eds.), Eurographics Association, pp. 69–79 + 409. 7

[GBK04] GUTHE M., BORODIN P., KLEIN R.: Efficient view-dependent out-of-core visualization. vol. 5444, pp. 428–438. 2

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

[Hop97] HOPPE H.: View-dependent refinement of progressive meshes. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), ACM Press/Addison-Wesley Publishing Co., pp. 189–198. 2

[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* (New York, NY, USA, 2009), ACM, pp. 169–176. 2

[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. 3

[Paj01] PAJAROLA R.: Fastmesh: Efficient view-dependent meshing. *Computer Graphics and Applications, Pacific Conference on 0* (2001), 0022. 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

[SM06] SANDER P. V., MITCHELL J. L.: Progressive buffers: view-dependent geometry and texture lod rendering. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses* (New York, NY, USA, 2006), ACM, pp. 1–18. 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* (Los Alamitos, CA, USA, 1996), IEEE Computer Society Press, pp. 327–ff. 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* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 131–138. 2, 7