Eurographics Symposium on Parallel Graphics and Visualization (2010)
J. Ahrens, K. Debattista, and R. Pajarola (Editors)

Parallel View-Dependent Refinement of Compact Progressive Meshes

E. Derzapf[1], N. Menzel[1] and M. Guthe[1]

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

Abstract

The complexity of polygonal models still grows faster than the ability of the graphics hardware to render them in real-time. A common way to deal with such models is to use multiple levels of detail (LODs). These can be static with the advantage that the simplification can be performed without regarding real-time constraints and the rendering algorithm simply chooses which LODs to render at runtime. Static LODs however suffer from sudden mesh transitions (popping artifacts) when the levels are too different. Dynamic or view-dependent LODs solve this problem by allowing for a continuous and smooth refinement. Unfortunately, they become computationally too expensive when the number of vertices is high, because refinement operations have to be computed for every vertex. In this paper, we address this problem by introducing a compact data structure for progressive meshes optimized for parallel processing and low memory consumption on the GPU. We also present an efficient LOD adaption algorithm resulting in an adaption time almost equal to the rendering time of the adapted mesh.

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

1. Introduction

To satisfy the ever growing demand for realistic renderings, the complexity of polygonal models is constantly increasing. Despite the enormous processing power of the GPU, such models can not be rendered in real-time. To reduce the number of triangles, existing techniques either use static or dynamic levels of detail (LODs). Static LODs are easy to use since simplification and rendering are decoupled, but suffer from popping artifacts if the simplification levels are not similar enough. In addition, the memory overhead compared to an ordinary mesh is typically about 50%. Dynamic LODs depart from this approach by encoding the simplification operations into a continuous sequence. As a result, popping artifacts are often nearly invisible. As the LOD is view-dependent, it uses no more polygons than necessary. Yet another advantage is that back-facing polygons don't have to be refined, resulting in an overall reduction by a factor of three to four compared to static LODs. Two problems are however induced by dynamic LODs: First, the memory consumption is significantly higher than for static LOD if the operations and displacement vectors are stored in a straightforward way. For this reason, compressed representations using variable length encoding were proposed. These however do not support efficient view-dependent LOD adaption. The second problem is that the number of split- and collapse operations which have to be processed for every view-point change increases with the number of triangles. Existing approaches tried to compensate this problem by distributing LOD adaption over several frames. This solution is unsatisfying, since the rendering performance is increasing much faster than the CPU processing power, resulting in a growing adaption delay. Existing parallel GPU approaches only partially solve this problem as they are either too restrictive or the rendering is still significantly faster than the adaption.

In this paper we solve both problems discussed above by introducing a compact representation of progressive meshes that is specifically designed for real-time parallel adaption on the GPU. Our two main contributions are:

- A novel random access data structure for progressive meshes that requires less than 50% of an ordinary mesh.
- A massively parallel adaption algorithm running on the GPU that is almost as fast as rendering the adapted mesh. Our approach outperforms previous techniques by almost an order of magnitude.

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

2. Related Work

View-dependent simplification has been an active field of research over the last decades. In addition to the existing static LOD techniques [GH97], Hoppe introduced progressive meshes that smoothly interpolate between different LODs [Hop96]. Depending on the view position and distance, a sequence of split- or collapse operations is performed for each vertex. 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]. El-Sana et al. [ESV99] prevented fold-overs of triangles by introducing a view-dependent tree containing modified rules for the operations. The dependency also requires no additional memory, but a suitable neighborhood data structure for the adapted mesh. In addition, the split operations cannot be stored as compactly as with the approach of Hoppe. 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. This however limits the algorithm to two-manifold triangle meshes and requires 24 additional bytes per vertex of the adapted mesh.

Recently methods for interactive visualization of large multiresolution geometric models have been modified for parallel processing. Sander et al. [SM06] proposed an algorithm that performs geomorphing on the GPU to render a given mesh. This approach requires building hierarchical static LOD structures, a trade off between static and view-dependent LODs. The GPU-based approach by Ji et al. [JWLL06] generates an LOD texture atlas by resampling the original model onto a regular remesh over a polycube map. They use a vertex shader to displace invisible vertices to infinity. This however has a significant impact on performance since no vertex transformations are saved. DeCoro and Tatarchuk [DT07] propose an octree-based vertex clustering for real-time simplification on the GPU. Adaptive simplification is supported by warping the input mesh. While the algorithm is fast enough to generate LODs at runtime, the visual quality is suboptimal due to the primtive vertex-clustering. A more general approach that is directly based on progressive meshes was developed by Hu et al. [HSH09]. They introduce 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 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.

3. Compact View-Dependent Progressive Mesh

Our view-dependent refinement algorithm is based on the vertex hierarchy of progressive meshes [Hop97]. The construction of the split hierarchy is unmodified, but instead of directly using the quadric error for the LOD selection we utilize the appearance error of Guthe et al. [GBBK04] to support arbitrary vertex attributes, because it significantly improves the visual quality at the cost of a slightly higher primitive count. The progressive mesh is generated by simplifying the original mesh to the base mesh with a series of collapse operations. The original mesh can then be reconstructed by applying the corresponding split operations in reverse order. A view-dependent reconstruction can be generated by performing only those splits that are necessary for the current view point. 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 where the root nodes are the vertices of the base mesh. Figure 1 shows an edge collapse operation col_v which removes the vertex v_u and modifies v. The adjacent faces f_l and f_r of v 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 and v_u. In addition, some of the faces adjacent to v become adjacent to the new vertex v_u.

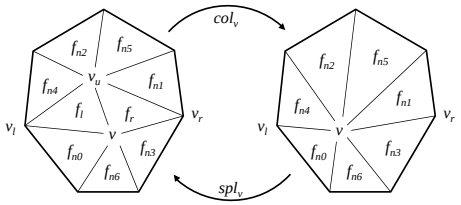

Figure 1: *Edge collapse and vertex split operation.*

3.1. Tree Structure and Dependency Coding

The operations are stored in a one-dimensional array. The tree structure could now be encoded by storing the indices i of the left and right child for every node. This would however require 8 bytes per operation, whereas a binary tree can be encoded using only two bits per node with a succinct coding. We only store if left and right child are present using a single bit for each of them in a *tree structure byte*. We can then calculate the indices i_l and i_r from the current index i:

$$\begin{aligned} i_l &= 2i + n_r - skip_i \\ i_r &= 2i + n_r - skip_i + 1, \end{aligned}$$

where $skip_i$ is the number of empty nodes up to the left child and n_r is the number of root nodes (i.e. base mesh vertices). An example of the operation tree is shown in Figure 2. Unfortunately, calculating the current $skip_i$ requires parsing the complete data structure up to the current node and counting the number of zero bits. Instead of this, we could store the skip count with every operation. This would again require 4.25 bytes per operation. A far more compact encoding with equal computational complexity can be achieved by only storing the skip for every n-th node. Then we could count the number of zero bits from the last *base skip*. In our approach, we completely avoid counting by encoding the difference between $skip_i$ and the base skip. This difference is stored the six remaining bits of the tree structure byte. In these six bits we can encode numbers up to 63. Considering the fact that $skip_i$ can be at most two larger than $skip_{i-1}$, we need to store the base skip for every 32nd node. This sums up to a total of 1.125 bytes per node to encode the tree structure.

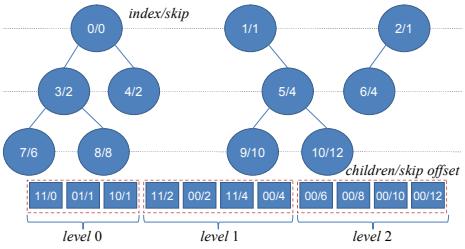

Figure 2: *Compactly encoded forest of binary trees.*

Unfortunately we cannot use the operation indices to preserve the local ordering since the tree encoding has changed their sequence. This necessitates to explicitly encode the neighborhood dependency for each operation. A compact encoding of this dependency can however be derived rather simple: we start with the base mesh M_0 and collect all operations that can be applied directly. After applying these, we get the maximally refined next mesh M_1. Repeating this procedure, we generate a series of meshes $\{M_0, M_1, \ldots, M_n\}$, where M_n is the original mesh. A split that refines M_i to M_{i+1} is then defined as having a split level of i. The local ordering is preserved if only splits with a lower split level than those of the neighboring vertices are applied. The corresponding condition for an edge collapse is simply that the split level is less or equal for all vertices adjacent to v and v_u. As the level n of the original mesh is proportional to the logarithm of its vertices, a single byte is sufficient to store the split level even for models with several millions of triangles.

3.2. Topology Encoding

To efficiently encode which of the neighbor vertices are adjacent to the new faces, we impose an ordering on the vertex neighborhood. Then we can simply encode the vertices v_l and v_r by their rank in this ordered sequence. The same applies to the partitioning of the neighbor faces into those that are adjacent to v_u after the split and those that remain adjacent to v. Here we can simply use a bit vector where an entry is set to one of the v is replaced by v_u for that face. Previous approaches use a cyclic ordering that requires a neighborhood graph. Instead, we assign a unique ID to every vertex and face which remains constant over any modification of the mesh. This allows for an efficient handling of vertex and face orderings and enables the algorithm to support non-manifold meshes. While the unique ID's of base mesh vertices and faces are simply their indices, we define the ID's of vertices and faces created by a split operation as follows:

$$\begin{aligned} ID_{v_u} &= v_0 + i_s \\ ID_{f_l} &= f_0 + 2i_s \\ ID_{f_r} &= f_0 + 2i_s + 1, \end{aligned}$$

where v_0 and f_0 are the number of vertices and faces in the base mesh and i_s is the index of the split operation.

Since a split creates zero to two faces, we also need to encode the case that f_l or f_r are not present. If both ranks are encoded using four bits each, up to 15 neighboring vertices can be handled within a single byte. When storing this partitioning in two bytes, vertices with up to 16 neighboring faces are supported. The limitations to 15 neighboring vertices and 16 faces do not impose significant restrictions since the average valence in a triangle mesh is 6 and valences above 15 are extremely rare. Nevertheless, the simplification algorithm needs to adhere to these restrictions. If the original mesh contains vertices with higher valence, they can not be collapsed until enough adjacent vertices are removed.

3.3. Attribute Encoding

In addition to the connectivity, the new attributes (position, normal, texture coordinates, etc.) of v and v_u must also be stored. Both of them are encoded as difference to the attributes of v before the split. In contrast to previous approaches we do not use a linear quantization of the whole input data but an individual one for each operation. This allows to reduce the quantization error by inserting a dummy split that does not change the connectivity. As the same quantization is used for all attribute differences of the operation, they must be scaled relatively to each other. Since the overall quantization error needs to be minimized, the relative scale factors should be as close as possible to the absolute difference values of all compression operations. This is achieved by computing the mean absolute difference of each attribute difference over all split operations. Then these norms are used as scale factors. As the scaling is relative to the other attributes, we can multiply all of them with an arbitrary factor. This factor is chosen such that the maximum absolute difference is scaled to MAX_HALF in order to utilize the complete range of values for the scaling of each operation. This per operation scaling is a division by a factor s such that the difference values are mapped into the interval $[-1\dots1]$. The optimal scaling (the minimal s) is then stored as half precision floating point. Despite the adaptive scaling one difference is most of the time significantly larger than the others. To accurately represent small and large values we apply a cubic function before the final quantization to n bits. Given a discrete value $d \in [-1\dots1]$, the quantized value q is:

$$q = \sqrt[3]{d}(2^{n-1}-1),$$

where n is the number of bits. The dequantization is:

$$d = \frac{q^3}{2^{n-1}-1}$$

3.4. Refinement Criteria

Three view-dependent criteria determine whether a vertex needs to be split or can be collapsed. A vertex can be collapsed if it is either outside of the view frustum or its normal if facing away from the viewer. Since each vertex of the adapted mesh represents several original vertices, there exists no single normal. Instead, we encode the maximum angular deviation α from the normal of the simplified vertex. Given the view direction $\mathbf{d} = \frac{\mathbf{p}-\mathbf{e}}{\|\mathbf{p}-\mathbf{e}\|}$, where $\mathbf{p}$ and $\mathbf{e}$ are the vertex and eye position, and the normal $\mathbf{n}$, the vertex is back-facing if:

$$\mathbf{n}\cdot\mathbf{d} > \sin\alpha$$

To save memory we do not store $\sin\alpha$ as floating point value but only a quantized value using four bits. During quantization the ceiling is used such that the inaccuracy only leads to back-facing vertices being classified as front-facing. Note that even the quantization to four bits, as we use it in our implementation, only marginally increases the number of rendered triangles. The vertex can possibly be collapsed when it is back-facing or outside the view frustum. Otherwise we need to check the simplification error of the associated split and collapse operations. The projected simplification error ε_s of the vertices' split operation s consists of the geometric distance γ_s and the attribute difference μ_s. The maximum simplification error δ_s is then simply $\max(\gamma_s,\mu_s)$. While the attribute difference is equal for all view directions, the projected geometric error depends on the angle between $\mathbf{n}$ and $\mathbf{d}$. The squared projected error can now be written as:

$$\varepsilon_s^2 = \frac{((\mathbf{n}\cdot\mathbf{d})\mu_s)^2 + (\|\mathbf{n}\times\mathbf{d}\|\delta_s)^2}{D^2},$$

where $D = \|\mathbf{p}-\mathbf{e}\|$ is the view distance. To efficiently store the two simplification errors, we exploit the fact that δ_s is more important as it dominates the projected error. So instead of two floating point values, we only store δ_s as half float and quantize the ratio $\lambda_s = \frac{\mu_s}{\delta_s}$ using four bits. The ratio and $\sin\alpha$ are stored into a common byte. We then write the projected error as follows:

$$\begin{aligned}\varepsilon_s^2 &= \frac{((\mathbf{n}\cdot\mathbf{d})\lambda_s)^2 + \|\mathbf{n}\times\mathbf{d}\|^2}{D^2}\delta_s^2 \\ &= \frac{(\mathbf{n}\cdot\mathbf{d})^2(\lambda_s^2-1)+1}{D^2}\delta_s^2\end{aligned}$$

If ε_s exceeds a given threshold the split operation has to be applied. On the other hand, the vertex can be collapsed, if the simplification error ε_c of its collapse operation c is below the threshold. In the current implementation, we use $\tan\frac{1}{60}^\circ$ as threshold to guarantee that the difference is not perceived if the projection matches the real-world view condition. Other thresholds, e.g. $\frac{1}{2}$ pixel screen space error, are also possible.

3.5. Dynamic Data Structures

The adaption algorithm maintains a static *split tree* storing the split hierarchy as well as a dynamically updated *VertexBuffer*, *IndexBuffer*, and some other temporary data. The SplitTree contains both the topologic and the geometric information of the progressive mesh. The base mesh is only used to initialize the dynamic data and thus not kept in memory. By using these data structures, the selectively refined mesh can be rendered in real-time. Table 1 shows the static and dynamic data structures in detail, which are required to maintain the relevant buffers. Since the complete algorithm runs on the GPU, the all data is stored in graphics memory.

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. The neighborhood information for the currently applied split operations is stored in the neighborhood array. It contains the number of adjacent triangles and their indices for each vertex that will be split. Since up to 16 neighbor triangles exist,

buffers	elements	memory (bytes)
static structures		
operations	tree structure	$1\frac{1}{8}n$
	dependency	$1n$
	ref. criteria	$3n$
	topology	$3n$
delta vectors	quant. delta	$2kn$
	delta scale	$2n$
dynamic structures		
active faces	index VBO	$24m$
	triangle ID	$8m$
active vertices	vertex VBO	$4km$
	vertex ID	$4m$
	next split	$4m$
	next collapse	$4m$
	state v_{state}	$1m$
collapse tree	split index i_c	$4m$
	prev collapse	$4m$
	vertex v_u	$4m$
temporary	prefix sum	$24m$
neighborhood	size	$1m$
	triangle index	$16m$
total		$(10\frac{1}{8}+2k)n+(98+4k)m$

Table 1: *Elements of the data structure. k, n, and m are the number of attributes, original, and base mesh vertices.*

we need 68 bytes per split. As only a quarter of the vertices can be split in parallel, this translates to 17 bytes per active vertex. For each active vertex v, the algorithm additionally stores its state, the unique ID and the next split and collapse operation. As the tree structure of the progressive mesh is only stored from root towards the leaves, the upwards references are kept in the dynamically updated *collapse tree*. Its elements consist of the index of the corresponding split operation i_s, a reference to the previous collapse, and a reference to the vertex v_u that is removed by this operation. In addition, each active vertex v holds a reference to the corresponding entry in the collapse tree. The complete structure is shown in Figure 3. In addition, three temporary buffers are required for the compactions: one for the scan input, one for the output and one temporary buffer [SHZO07]. Each buffer

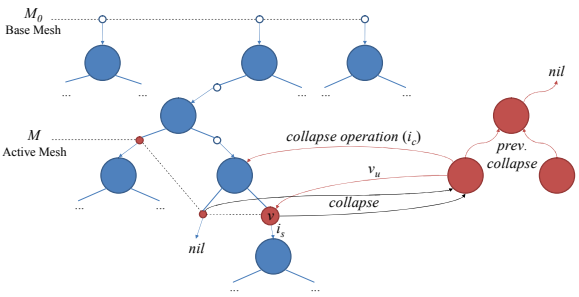

Figure 3: *Linking between active vertices, the SplitTree, and the CollapseTree.*

contains four bytes per entry and the maximum number of entries is the number of triangles in the current mesh which is twice the number of vertices.

Most other algorithms support meshes with $k = 8$ attributes only consisting of position, normal, and 2d texture coordinates. With $k = 8$ the complete hierarchy and adapted mesh use a total of $26\frac{1}{8}n+130m$ bytes, where n and m are the numbers of vertices in the original and adapted meshes respectively. Table 2 shows a comparison with previous view-dependent LOD schemes. As for highly detailed models, it is generally impossible to view the whole surface at high resolution within a single frame, one can assume that $m << n$. Therefore, the requirements for the static data structures that are the only depending on n are more important than the ones of the dynamic structures. With only $26\frac{1}{8}n$ our proposed data structure favorably compares to all other view-dependent methods with at least a reduction of 62%, so roughly a third of the memory.

View-Dependent LOD scheme	*Memory size (bytes)*
VDPM [Hop97]	$216n$
SVDLOD [Hop98]	$88n+100m$
MT [DFMP98]	$75n$
VDT [ESV99]	$90n$
FastMesh [PD04]	$88n+6m$
PVDPM [HSH09]	$69n+56m$
Our scheme	$26\frac{1}{8}n+130m$

Table 2: *Comparison of memory size with previous schemes for $k = 8$ attributes.*

Compared to the 129 bits per vertex (bpv) we need, the compressed progressive meshes of Hoppe [Hop96] and Pajarola and Rossignac [PR00] require 31–50 and 21–28 bpv only. The drawback of those is that both use variable length coding which is not suitable for efficient random access decompression. Therefore, their approaches cannot be used for selective parallel LOD adaption. In addition, [PR00] does not support view-dependent refinement at all.

4. Runtime Algorithm

In order to efficiently exploit the parallel architecture of the Compute Unified Device Architecture (CUDA), the algorithm is subdivided into several steps, that are performed for each parallel adaption. The partitioning is chosen such that each step can be processed completely in parallel. To classify which operation can or should be applied to a vertex, we track those options in a status byte v_{state}. Two bit flags, *potential_split* and *potential_collapse*, are used to mark those vertices that can possibly be split and/or collapsed. If their neighborhood prevents any of these operations, the according bit is cleared. The operation that should be applied to the vertex is stored in two other bit flags, *want_split* and *want_collapse*. A combination with the corresponding potential flags marks a vertex for a split or collapse operation.

In addition, a vertex can have one of the following two special states after a collapse operation: the vertex that was removed is labeled with the *removed* state and the other one is marked as *collapsed*. The state is stored in a separate array to facilitate coalesced reading and writing since it is accessed very often. Based on these states, split and collapse operations are applied.

4.1. Vertex State Update

If the refinement criteria determine that vertex v needs to be split, we set the *want_split* flag in its state. Otherwise, we set the *want_collapse* flag if the refinement criteria allow a collapse and the vertex is marked as *potential_collapse*. In all other cases no operation is required for v, unless it was already marked as *want_split* in a previous iteration. In both of these cases the state remains unchanged. After we have determined the desired operation for each vertex, we need to check if both vertices of a collapse operation are marked for coarsening. If only one of them is marked, the *want_collapse* flag is removed again. Note that since we check for impossible operations later, this step is not strictly necessary but leads to a considerable speedup.

Due to the dependency of split operations we may have marked vertices for split or collapse operations that cannot be performed. In case of a collapse this is not problematic since the operation is not necessary to achieve a desired quality. For a split operation however, we need to find those neighboring vertices that must be split before the current vertex v. For this purpose, each face f is checked whether one of its vertices is marked as *want_split* but another vertex of the same face has a lower split level. If this vertex is not already marked for splitting as well, its *want_split* flag also needs to be set. The procedure is performed twice since only the neighboring dependent splits are marked each time. This way, every dependent split with a topological distance of d_t is marked after at most $\frac{d_t}{2}$ adaption iterations. To remove splits and collapses that cannot be performed yet, the algorithm traverses all triangles and again checks the vertex states. For the split operations, only the vertex with the lowest split level can be split in each face f. In addition, if any vertex is marked for splitting, no vertex of f can be collapsed. If no vertex of the face needs to be split, we finally check the collapse operations. Here only the collapse operation with the highest level can be performed in each triangle. Note that we do not change the corresponding *want* flags, but the *potential* flags to prevent repeated checking of the same vertices. Algorithm 1 shows the complete vertex update partitioned into the four stages described above.

When processing large amounts of data on the GPU the aligned memory access of each thread group (warp) is crucial for performance. This access pattern is called coalesced reading and writing. When looping over all faces, we need the three vertex indices of each face but directly loading them from global memory would violate coalescing. To prevent this, we read all vertex indices of a thread block into its shared local memory and then fetch the indices of the current triangle from there.

```
foreach vertex v in parallel do
    if marked(v, collapsed)
        mark(v, potential_split, potential_collapse)
    if need_split(v)
        mark(v, want_split)
    elif may_collapse(v) && marked(v, potential_collapse)
        mark(v, want_collapse)
foreach vertex v in parallel do
    v_u = get_other(v)
    if v != v_u
        if !marked(v_u, want_collapse)
            unmark(v, want_collapse)
        if !marked(v, want_collapse)
            unmark(v_u, want_collapse)
repeat twice
    foreach face f in parallel do
        if any_vertex_marked(f, want_split)
            level_max = get_max_active_split_level(f)
            mark_dependent_splits(f, level_max, want_split)
foreach face f in parallel do
    if any_vertex_marked(f, split)
        level_min = get_min_active_split_level(f)
        unmark_dependent_splits(f, level_min)
    if any_vertex_marked(f, want_split)
        unmark_all_collapses(f)
    elif any_vertex_marked(f, collapse)
        level_max = get_max_active_collapse_level(f)
        unmark_illegal_collapses(f, level_max)
```

Algorithm 1: The four parallel stages to update the vertex states. The third stage is performed twice to speed up the propagation of dependent splits through the mesh.

4.2. Parallel Vertex Splits

After updating the state of all active vertices and removing illegal splits and collapses, the operations can be applied. Before the splits can be performed, the neighborhood of each split vertex needs to be known. The neighborhood information is collected by traversing all faces and if a face f is adjacent to a split s_v, the face index is added to the neighborhood of v. As we only want to collect the neighborhood for the vertices that are currently split, we have to first perform a so-called compaction operation on the split indices. For this purpose, we use the parallel compaction algorithm of Sengupta et al. [SHZO07] to compute an array of split indices. After generating this array and collecting the face neighbors for each split vertex v, the following operations are performed:

1. The new vertex v_u is generated and v is moved to it's new position.
2. The two faces f_l and f_r are added to the index buffer and the triangle ID array.
3. The other faces in the neighborhood of v are relinked according to the encoded topology changes.

4. The adjacent vertices of v and v_u are marked as *potential_split* as their split could be waiting for the current one. As the adjacent vertices cannot be collapsed, the *potential_collapse* flag is cleared.
5. v and v_u are marked as *potential_split* and *potential_collapse* since both operations could be possible in the next frame.

Algorithm 2 gives an overview of the complete parallel vertex split procedure.

```
compact(splits)
foreach face f in parallel do
    if adjacent_to_split(f)
        append_to_neighborhoods(f)
foreach split s_v in parallel do
    split_vertex(v)
    add_faces(v)
    relink_neighbor_faces(v)
    mark_neighbor_vertices(v, potential_split)
```

Algorithm 2: Parallel vertex split algorithm.

As the neighborhood information is parsed identically for every split vertex, we achieve coalesced reading with the following layout: First, we store the number of adjacent faces, then the indices of the first neighbor triangle, than that of the second and so on. In addition to this layout, we must assure that each new block of indices begins at an address that is a multiple of 128. Therefore, we round up the number of splits to the next multiple of 32 for addressing in the neighborhood array.

4.3. Parallel Edge Collapses

To perform the collapse c_v, the corresponding vertex v_u is required. Since v_u is stored with the collapse operation, we simply need to check whether the current vertex is different. The collapse is only applied if both are marked as *collapse*. The operation c_v marks vertex v_u as *removed*, moves v to the target position and marks it as *collapsed*. In addition, the target vertex for v_u is stored in the next split since this is not required any more after removing v_u. Then all faces are relinked by checking if a vertex of face f was removed. In this case, the target vertex is fetched and the vertex of the face is set accordingly. If the face becomes degenerated it is removed as well. When a collapse was applied to one vertex of the face, all other vertices are candidates for a possible collapse and are marked as such. Algorithm 3 shows the parallel processing of the edge collapse operations.

```
foreach vertex v in parallel do
    if marked(v, collapse)
        v_u = get_other(v)
        if v! = v_u && marked(v_u, collapse)
            collapse_vertices(v, v_u)
foreach face f in parallel do
    relink_vertices(f)
    if degenerate(f)
        remove_face(f)
    else if changed(f)
        mark_vertices(f, potential_collapse)
```

Algorithm 3: Parallel edge collapse algorithm.

4.4. Buffer Compaction

The final 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. Note that when compacting the vertices or collapses, the references to them must be updated accordingly. While the compaction of the faces and thus the indices is mandatory since the index VBO is used for rendering, the compaction of the vertices and collapse operations is not. The latter two only need to be compacted every few frames to prevent bloating of the buffers. As only a few elements are removed each time and the ordering does not need to be preserved, we have developed a specialized in-place compaction algorithm. In contrast to previous approaches, it has the advantage that we do not need to duplicate the array that is compacted. Otherwise we would need copies of all dynamic data structures except the temporary buffers and the neighborhood information which would drastically increase the memory consumption.

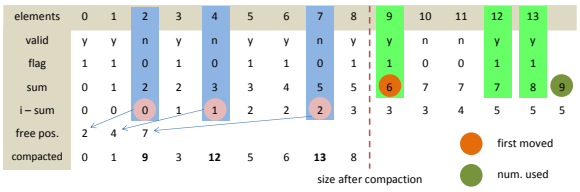

Figure 4: *Basic principle of our in-place compaction algorithm.*

The main idea of the compaction is to first calculate the number n_c of elements after the compaction. Then all gaps before n_c are filled with elements after n_c (see Figure 4). First the valid elements are marked with one and the invalid ones with zero. Then we compute the prefix sum using the parallel algorithm of Sengupta et al. [SHZO07]. This gives us the number of valid elements n_c as well as the first valid element that needs to be moved. We then gather the positions of the empty elements in the final array. Their position in the *free position* array can be computed by subtracting the prefix sum from their index. Finally we can compute the target position of the elements that need to be moved by subtracting the *first moved* from the prefix sum of the current element. We then look up the position in the *free position* array and can copy the element into the compacted buffer. The algorithm does not require additional temporary memory except that used to compute the prefix sum. The free positions can

overwrite the flag array, since the flags are not required anymore after computing the sums.

4.5. Memory Management

During adaption the memory requirements of the dynamic data structures can grow or shrink. To alleviate the cost for memory allocation and copy when the size of an array is modified, we always reserve more memory than currently required. In addition, the array size is restricted to multiples of 4096 elements. Figure 5 shows an example of growing and shrinking a data structure. If the currently required amount of memory exceeds the array size, we allocate one additional block to prevent re-allocation in the next frame. For shrinking a similar strategy is employed by only allocating a smaller array if more than two blocks are empty. Despite reducing the memory consumption, we still keep one free block to prevent quick re-allocation when the array is growing again.

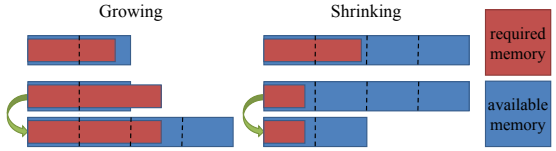

Figure 5: *Growing (left) and shrinking (right) of an allocated array during adaption.*

To improve the rendering performance, the triangles are sorted such that the transform cache can be utilized. As the ordering is gradually destroyed when inserting new faces at the end of the buffer, we need to restore it every few frames. Since the memory consumption also changes when many operations are applied, we simply integrate the sorting into the memory management. When a new buffer is allocated for the index VBO and the face IDs, we sort the triangles by their minimum index during the copy operation. This results in a mesh that is mainly composed of triangle fans and reduces the transform cost by a factor between of two to three. In total, the rendering time is reduced by 20% to 50% depending on the complexity of the fragment shader. If no allocation occured for more than two seconds, we force a re-allocation and thus a sorting of the faces. This is necessary as gradual movements do not quickly enough lead to changes in memory consumption but nevertheless perturb the ordering.

5. Results

Our test system is built of a 3 GHz Intel Core2 Duo CPU with 2 GByte of main memory and a GeForce GTX 285 where we use the OpenGL API for rendering. We first evaluate the memory requirements of the static progressive mesh data structure. Table 3 gives an overview of the progressive meshes we used as input and the number of added dummy split operations.

model	v_0	f_0	k	*# ops.*	*# dummy ops.*	lvl.
Phl. Dragon	41	44	14	240,016	6,099 (2.54%)	142
St. Dragon	815	536	6	436,830	3,215 (0.74%)	207
Buddha	727	1866	6	542,925	3,529 (0.65%)	135
Manuscript	42	17	10	2,155,575	3,369 (0.16%)	191
Asian Dragon	35	16	6	3,609,565	7,801 (0.22%)	253

Table 3: *Progressive meshes used as input, number of added dummy split operations, and maximum split level.*

With the exception of the Phlegmatic Dragon, the number of dummy splits is significantly below 1% of the original number of operations. But fortunately, the compression ratio of this model will increase again as it has the highest number of attributes k as tangents and texture coordinates are required to map a BTF on the model. All other models except the manuscript, which additionally stores per vertex color, only use position and normal as vertex attributes. The maximum split level is proportional to the logarithm of the ratio of original to base mesh vertices. Therefore, larger models can be handled by increasing the complexity of the base mesh. The resulting compressed sizes, compared to an indexed face set that is traditionally stored on the GPU for rendering, are listed in Table 4. As expected, the models with higher number of attributes are slightly less compressed by our method. Nevertheless, the memory reduction is relatively similar for all models. The memory consumption lies between 46% and 50% compared to an indexed face set.

model	v_{max}	f_{max}	*mem. IFS*	*mem. PM*
Phl. Dragon	240,057	480,076	18.3MB	9.0MB (49.2%)
St. Dragon	437,645	871,414	19.9MB	9.3MB (46.5%)
Buddha	543,652	1,087,716	24.9MB	11.5MB (46.2%)
Manuscript	2,152,840	4,305,679	123.3MB	57.9MB (47.0%)
Asian Dragon	3,609,455	7,218,906	165.2MB	76.1MB (46.1%)

Table 4: *Comparison of the static data that resides in graphics memory compared to an indexed face set.*

During rendering, the dynamic data structures consume additional memory. For all models except the phlegmatic dragon, the total amount of graphics memory nevertheless stays below that of an indexed face set. Table 5 shows the number of rendered faces, the total rendering time, and the memory consumption for the views shown in Figure 6. For almost all models the required memory and total frame time

model	*rendered # faces*	*memory (MB)*	*total frame time (ms)*
Phl. Dragon	224,090 (46.7%)	24.1 (129.1%)	10.4 (131.4%)
St. Dragon	190,236 (21.8%)	19.1 (93.3%)	3.2 (95.8%)
Buddha	152,716 (14.0%)	19.5 (78.1%)	3.3 (68.7%)
Manuscript	274,678 (6.4%)	73.5 (59.6%)	4.5 (39.8%)
Asian Dragon	646,844 (9.0%)	108.9 (65.9%)	10.5 (41.2%)

Table 5: *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.*

are always less than that of the original mesh. The coarsening on the back faces and outside the view frustum can be clearly seen in the external views of the adapted models. The reduced level-of-detail further away from the camera can also be noticed at the example of the Asian Dragon.

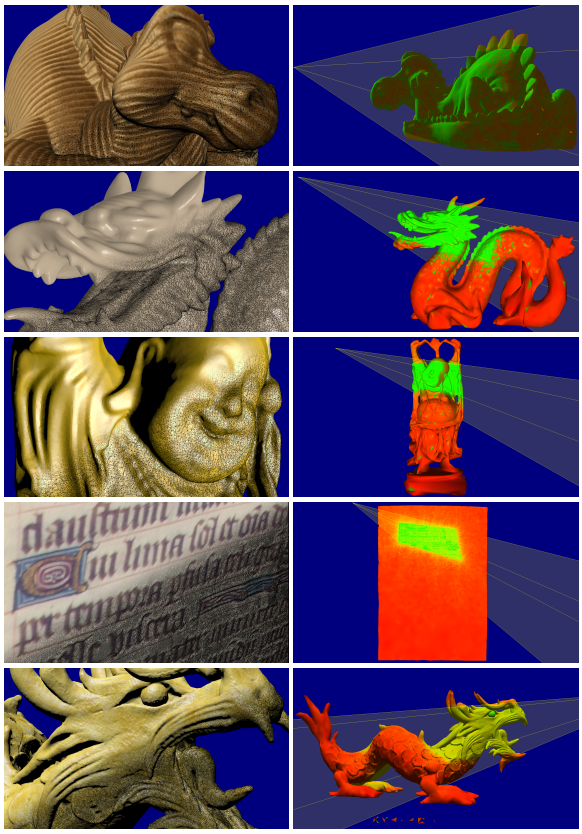

Figure 6: *Renderings of view-dependently refined meshes. The images on the right show external views with the view frustum in yellow. The color coding depicts the level of detail, where red is low LOD and green high.*

Figure 7 shows the adaption and rendering time together with the memory consumption for a pre-recorded movement around the Asian Dragon. The consumed graphics memory is always less than required by the original model. The frame rate seldomly drops below the 60 Hz of the display even when a high number of triangles is required. As the time required for each frame is often significantly below 16 ms, we could even perform more than one adaption iteration and only render once. Compared to static hierarchical LODs (HLODs) using the same error measure [GBBK04], 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. A special problem of HLODs is that rendering is split into several independent render calls which reduces the number of primitives per second compared to a single mesh.

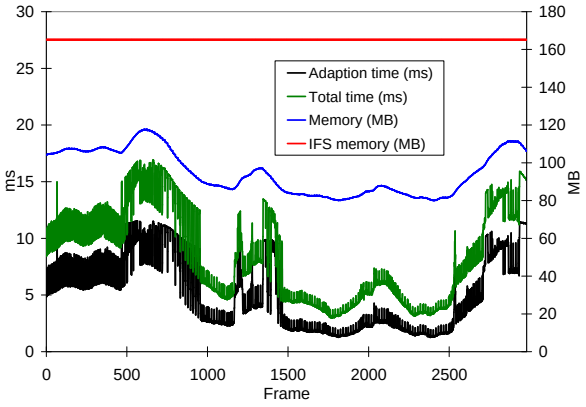

Figure 7: *Timings and memory consumption for the Asian dragon with a pre-recorded camera path.*

Figure 8 shows a detailed analysis of the runtime of each step of the adaption and rendering. Note that the rendering performance is identical to rendering a static model with the same number of triangles and thus our method needs approximately 2.6 times as long as rendering a static mesh. Considering that we already cut down the vertices by half due to the simplification of back 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. On average we can process 120 million triangles per second (M△/sec). This is a speedup of factor 9.2 compared to the approach of Hoppe et al. [HSH09] that only achieves 13 M△/sec on our test system. The main source of speedup is probably due to the fact that we use a single CUDA compaction instead of several geometry shaders to construct a compact indexed buffer for rendering. In addition, rendering the adapted mesh is also approximately twice as fast using our approach due to the

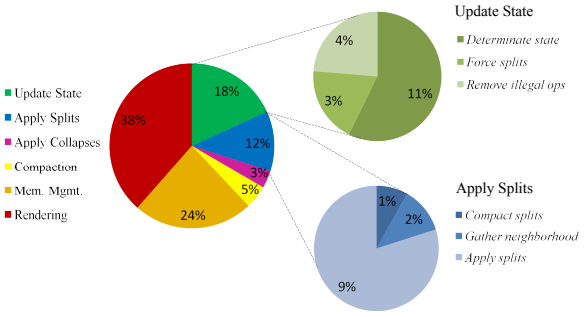

Figure 8: *Time per frame partition for the serveral steps of our algorithm and for map-/unmap as well as rendering.*

compact vertex buffer. Due to ther stronger neighborhood contraints our method however requires thrice the number of iterations to reach the specified LOD than the method of Hoppe et al. [HSH09]. Since on the other hand each iteration is more than nine times faster, our method still converges in a third of the time. Nevertheless, popping artifacts are still visible during a fast pan over the model which can be observed in the accompanying video. Note that the relatively large amount of time required for memory management could be reduced by using larger blocks during allocation. This would on the other hand increase the amount of memory used up by the dynamic data structures.

Even with most recent graphics drivers approximately 25% of additional rendering time is required for the mapping and unmapping of the index and vertex buffer for access from CUDA. According to the documentation the time for mapping should be insignificant if the device is set to OpenGL interoperability. As we have observed no difference between activating OpenGL interoperability or not, we consider this to be a driver problem and did not include this time in our results. This is also one of the reasons for the rather large share of the memory managemant as allocating a new VBO requires unmapping the old and mapping the new one.

6. Conclusion and Limitations

We have presented a compressed progressive mesh representation that was specifically developed for parallel refinement on modern graphics hardware. By performing all currently possible adaptions in parallel, we only need 1.6 times as long as for rendering of the adapted mesh. In total we need 2.6 times as long as for rendering alone. This means that the performance is increased as soon as 62% of the vertices are removed by simplification. Due to the view-dependent adaption, this reduction is almost achieved by coarsening the back faces alone. Compared to prevoius parallel view-dependent refinement algorithms we achieve an almost ten-fold performance improvement. Our algorithm even outperforms hierarchical LODs that were considered near-optimal for current graphics hardware by a factor of more than two.

In addition to the improved performance, our method also requires even less graphics memory than the original model stored as indexed face set. For larger models, approximately 30% to 40% are saved on average while other algorithms need more memory than the original model.

One limitation of our algorithm is that despite sorting the triangle into fans to utilize the vertex cache, 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

[DFMP98] DE FLORIANI L., MAGILLO P., PUPPO E.: Efficient implementation of multi-triangulations. In *VIS '98: Proceedings of the conference on Visualization '98* (Los Alamitos, CA, USA, 1998), IEEE Computer Society Press, pp. 43–50.

[DT07] DECORO C., TATARCHUK N.: Real-time mesh simplification using the gpu. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2007), ACM, pp. 161–166.

[ESV99] EL-SANA J., VARSHNEY A.: Generalized view-dependent simplification. *Computer Graphics Forum 18*, 3 (1999), 83–94.

[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)* (June 2004), Keller A., Jensen H. W., (Eds.), Eurographics Association, pp. 69–79 + 409.

[GH97] GARLAND M., HECKBERT P. S.: Surface simplification using quadric error metrics. 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. 209–216.

[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.

[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.

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

[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.

[JWLL06] JI J., WU E., LI S., LIU X.: View-dependent refinement of multiresolution meshes using programmable graphics hardware. *Vis. Comput. 22*, 6 (2006), 424–433.

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

[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.

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

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

[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.

[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.