Vision, Modeling, and Visualization (2011)
Peter Eisert, Konrad Polthier, and Joachim Hornegger (Eds.)

Instant Level-of-Detail

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

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

Figure 1: *A subset of the 10 detail levels for the Welsh Dragon generated with our algorithm.*

Abstract

Highly detailed models are commonly used in computer games and other interactive rendering applications. In this context, static levels-of-detail are frequently used to achieve real-time frame rates. While this is a simple solution to improve the rendering performance, the additional geometry needs to be stored and loaded into graphics memory. This is especially problematic in online applications, where the data needs to be transmitted over a possibly slow connection. On the other hand, consumer level computers are usually equipped with a graphics card that can be used for general purpose parallel computing. Based on this observation, we propose a high-quality parallel mesh simplification algorithm based on the quadric error metric. The simplification performance can compete with the time required to load additional meshes from a local hard disk.

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.3.1 [Computer Graphics]: Hardware architecture—Parallel processing

1. Introduction

Highly detailed geometric models are very popular in interactive applications such as computer games or internet shops. These models are usually represented as triangle meshes. To render several of these models at real-time frame rates, level-of-detail (LOD) techniques are commonly used. For multiple smaller objects, static LODs are usually the method of choice. Each model is represented at several resolutions. During rendering, a level is chosen per object based e.g. on the distance to the viewer (see Figure 2). Simplification algorithms can be used to automatically generate the different resolutions, so that designers only need to model the finest level.

Figure 2: *With increasing distance, coarser approximations of the model can be used.*

The main goal of simplification algorithms is to reduce the number of rendered triangles while introducing little or no visual difference. The maximum screen-space deviation

DOI: 10.2312/PE/VMV/VMV11/293-299

ε_s in pixels and the simplification error ε between simplified and original model induce a minimum view distance d. The distance also depends on the field of view α and the screen resolution r. It can be computed by $d = \varepsilon \frac{r}{2\varepsilon_s \tan\frac{\alpha}{2}}$.

As the distance depends on the maximum error, a simplification error that reduces the model up to a specified error bound is desirable in this context. An efficient way to reduce the model to a specified ε are the quadric error metrics proposed by Garland and Heckbert [GH97]. The distance between an simplified and original model is estimated by accumulating quadratic plane distances. While this is an overestimation of the real error, the specified error is still an upper bound.

Due to the processing time, simplification algorithms are normally used as a preprocessing step. Considering that the average performance of the quadric error metrics simplifier is about 50,000 operations per second, the LOD generation needs 20 seconds for a model with one million vertices. Therefore, the levels are stored on disk and loaded at program startup. As the LODs in total normally contain as many triangles as the original mesh, the loading times are doubled. While this is unproblematic when loading from a local disk, it might be inacceptible for online applications. Here the LODs need to be generated from the transferred original mesh. Although almost every customer level computer contains a graphics card that can be used for general purpose computations, edge-collapse simplification algorithms are still working sequentially. This is mainly due to the fact that a significant amount of neighborhood information is required to compute an optimal ordering of operations.

The main contribution of our paper is a high-quality parallel simplification algorithm using edge collapse operations. Based on the observation, that the ordering only needs to be preserved locally, we propose to determine and collapse all possible edges in parallel. The collapsed vertex locations and the simplification errors are computed using the quadric error metric. This leads to an exceedingly fast high-quality simplification algorithm. Using our implementation, we can generate a complete set of 10 detail levels from the welsh dragon model (2.2 million faces) within 0.73 seconds. Figure 1) shows a subset of the generated levels.

2. Related Work

Mesh simplification is one of the fundamental techniques for real-time rendering of polygonal models. There is an extensive amount of methods that mainly focus on accurate bounds of the simplification error. A detailed review of simplification algorithms is given by Luebke [Lue01]. As we focus on real-time simplification, we only discuss the methods that would be suitable candidates.

Rossignac and Borrel [RB93] proposed to use uniform vertex clustering. The bounding box of the model is subdivided using a regular grid and all vertices inside the same grid cell are collapsed to their mean. Low and Tan [LT97] proposed a weighted vertex clustering to preserve edge features that are not aligned with the grid. While uniform clustering is relatively fast and a precise upper bound for the simplification error can be given, a further reduction in flat regions would be possible.

Garland and Heckbert [GH97] as well as Popović and Hoppe [PH97] introduced the vertex pair contraction. This approach has become the most common technique for the simplification of triangle meshes. The contraction operation is combined with the introduced quadric error metric. It allows a flexible control over the geometric error and can be used to calculate optimal vertex positions. Later Garland and Heckbert extended their approach to an arbitrary number of vertex attributes [GH98]. While the generated approximations are superior to vertex clustering at the same number of triangles, the simplification performance is significantly lower. On the other hand, the required levels can be generated using a single simplification sequence from the original model to the coarsest level.

Lindstrom [Lin00] proposed a combination of vertex clustering with error quadrics to improve the placement of the clustered vertices. Nevertheless, a high number of triangles is used in flat regions. Shaffer and Garland [SG01] proposed to overcome this problem by using a BSP tree instead of a uniform grid. The runtime is increased by a factor of three compared to uniform clustering, but the method is still faster than edge collapse simplification. An adaptive vertex clustering using octrees was later proposed by Schaefer and Warren [SW03]. The runtime is even higher than using a BSP tree, but the quality of the simplified mesh can almost compete with edge collapse simplification. Garland and Shaffer developed a multiphase algorithm [GS02] which combines vertex clustering with a subsequent edge contraction to generate a drastic simplification. While this is faster than edge collapses alone, it can only be used to generate a single detail level. Additionally, DeCoro and Tatarchuk [DT07] proposed a parallel implementation of vertex clustering on the GPU. It is based on the octree clustering approach of Schaefer and Warren [SW03] by implementing an efficient octree data structure on the GPU. While the performance is extremely high, it still has the same quality problems as all vertex clustering algorithms.

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. A more compact progressive meshes data structure for parallel adaption was proposed by Derzapf et al. [DMG10].

3. Quadric Error Metric

Our simplification algorithm generates the simplified models by collapsing all non-conflict edges in parallel. Figure 3

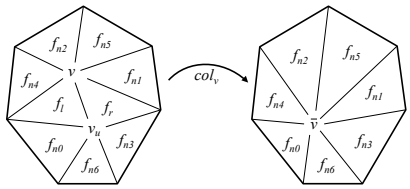

Figure 3: ***Edge collapse.*** *The edge defined by vertex v and v_u is collapsed into the vertex $\bar{v}$.*

shows the principle of an edge collapse operation col_v, which contracts an edge, connecting the vertices v and v_u, into a point. By applying col_v the adjacent faces f_l and f_r of the vertices v and v_u disappear and the position of the collapse vertex $\bar{v}$ is computed by minimizing the costs of the collapse operation col_v. To provide control over the simplification error and to evaluate the costs for col_v a suitable measure is required. Garland and Heckbert [GH97] proposed a quadric error metric that estimates the distance between simplified and original mesh. The approximation is based on the distances of the simplified vertex to the planes defined by the adjacent triangles in the original mesh. Let $P(v)$ be the set of planes adjacent to mesh vertex v, then the maximum error can be estimated with the sum of squared distances:

$$\Delta(\mathbf{v}) = \Delta[v_x\ v_y\ v_z\ 1]^T = \sum_{\mathbf{p}\in P(v)} (\mathbf{p}^T\mathbf{v})^2,$$

where $\mathbf{p} = [a\ b\ c\ d]^T$ is the implicit plane equation $ax+by+cz+d=0$ in normalized form . Note, that the coefficients a, b and c are the plane normal and d the signed distance between the origin and the plane. The sum of the squared distances can be transformed into a quadratic form:

$$\Delta(\mathbf{v}) \;=\; \mathbf{v}^T \left(\sum_{\mathbf{p}\in P(v)} \mathbf{Q}_p \right) \mathbf{v},$$

where $\mathbf{Q}_p$ is the convariance matrix of the planes $\mathbf{p}$ in $P(v)$.

To prevent a degeneration of the mesh boundary, we use the same approach as Garland and Heckbert. In addition to the vertex quadrics, a boundary quadric is calculated for each boundary edge. It is computed from a virtual plane that is orthogonal to the triangle plane. Let v_1 and v_2 be the vertices of the boundary edge and v_3 the third vertex of the only adjacent face. Then the normal equation of the virtual plane is: $t_x x + t_y y + t_z z - n \cdot v_1 = 0$, with

$$e_1 = \frac{v_2 - v_1}{\|v_2 - v_1\|}, e_2 = \frac{v_3 - v_1}{\|v_3 - v_1\|}, t = \frac{e_2 - (e_1 \cdot e_2)e_1}{\|e_2 - (e_1 \cdot e_2)e_1\|}.$$

The quadric error metrics can be generalized to arbitrary dimensions [GH98]. The general quadric $\mathbf{Q}_p$ can be written as:

$$\mathbf{Q}_p \;=\; \begin{pmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{b}^T & d \end{pmatrix},$$

with

$$\begin{aligned} \mathbf{A} &= \mathbf{Id} - e_1 e_1^T - t t^T \\ \mathbf{b} &= (v_1 \cdot e_1)e_1 + (v_1 \cdot t)t - v_1 \\ d &= v_1 \cdot v_1 - v_1 \cdot e_1 - v_1 \cdot t \end{aligned}$$

In order to compute the cost of collapsing a pair of vertices v and v_u, we can derive the associated error from the vertex quadrics $\mathbf{Q}_v$ and $\mathbf{Q}_{vu}$. The total sum of squared distances is $\overline{\mathbf{Q}} = \mathbf{Q}_v + \mathbf{Q}_{vu}$. In addition, $\overline{\mathbf{Q}}$ can be used to find the optimal position of the collapsed vertex $\bar{\mathbf{v}}$. The optimal vertex minimizes the sum of squared distances $\Delta(\bar{v})$. This translates into solving the following linear equation system:

$$\nabla\overline{\mathbf{Q}}\ (\bar{v}) = \begin{pmatrix} & \mathbf{A} & \mathbf{b} \\ 0\cdots 0 & & 1 \end{pmatrix} \begin{pmatrix} \bar{v} \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

4. Overview

Traditionally, edge collapse simplification works by sequentially collapsing the edge with the minimum induced error. When generating a specific level-of-detail, the simplification stops when collapsing the next edge would exceed the error threshold. This can also be seen as successively collapsing the vertex v_a that is adjacent to the edge with the minimum error with the other vertex of that edge v_b. With this, we can associate the minimal adjacent edge error with the vertex. In our example, v_a and v_b will have the same error which is also the global minimum. The quadric error of the collapsed vertex is then the minimum of the other edges adjacent to the collapsed one. By construction, the vertex quadrics are monotonically increasing and thus the quadric error of the adjacent edges will be at least as high as that of the collapsed edge. This means that we can not only collapse the edge with the minimal error, but also all other edges that have a local minimum of the error below the given threshold. Due to the monotonically increasing error of the other edges, these will also be collapsed in the sequential algorithm. The overall algorithm therefore works as follows:

1. Compute vertex quadrics.
2. Compute placement and error for each edge.
3. Find local error minima below threshold and collapse edges.
4. Continue with 2. until no collapses can be performed.

To perform the quadric computations and the edge collapses in parallel, we require an adequate data structure. In addition to vertices and faces, we also need a data structure for the edges of the mesh. To compute the target placement for an edge collapse, access to the vertices and the associated vertex quadrics from the edge is required. Theoretically, we also need access to all edges from a vertex in order to find the edge with minimal error. Since we compute this for all vertices, we can use atomic operations and process all edges

instead. The same holds for removing the degenerated faces. There we can simply update the indices and remove the faces using a compaction algorithm.

4.1. Connectivity Data Structure

As discussed above, we need to extract the edges of the mesh. In addition, we need to determine the boundary edges for the boundary quadrics. Initially, we load an indexed face set (IFS) and transfer it to the GPU. This means that we have attributes per vertex and three indices per triangle. Algorithm 1 shows how the edge information is build from this data.

```
foreach face f in parallel do
    i1,i2,i3 = get_face_indices(f)
    edge1 = create_edge(min(i1,i2),max(i1,i2),i3)
    edge2 = create_edge(min(i2,i3),max(i2,i3),i1)
    edge3 = create_edge(min(i3,i1),max(i3,i1),i2)
RADIX SORT edges in parallel by imax
RADIX SORT edges in parallel by imin
foreach edge e in parallel do
    ep = get_previous_edge(e)
    if ep==e
        set_edge_flag(e, 0)
    else
        set_edge_flag(e, 1)
        if get_previous_edge(e)==e
            set_single_flag(e, 0)
        else
            set_single_flag(e, 1)
COMPACT edges in parallel
```

Algorithm 1: Parallel generation of the edge data structure.

We first generate an edge for each of the three half-edges of a triangle. Then we sort the edges by their higher vertex i_{max} and then by the lower one i_{min} using radix sort. Since radix sort is stable, the edges are now sortened in lexicographic order and we can mark duplicates for removal. During this process, we also store the third vertex of the face and flag single edges.

The complete data structure required for the edge setup and the allocated memory (per entry) are shown in Table 1. The input data are the *vertex* and the *index buffer*. All generated data are stored per edge, where the opposite vertex and single edge flag are only required to compute the vertex quadric. The total memory required during this phase is $246+4k$ bytes per vertex, where k is the number of vertex attributes, or 222 bytes in addition to the IFS. After removing the duplicate edges and freeing the temporary arrays, this is reduced to $63+4k$ bytes per vertex, or 39 additional bytes.

5. Parallel Simplification

The algorithm is subdivided into several consecutive steps to implement the simplification on massively parallel hardware. The partitioning is required for thread synchronisation while each step can be processed completely in parallel. Figure 4 shows the steps of the algorithm. The first step is to compute the vertex quadrics. These are sums of all adjacent

buffers	elements	bytes per entry
vertices	vertex VBO	$4k$
faces	index VBO	12
edges	vertex index ($\times 2$)	8
	vertex index (opposite)	4
	single edge flag	1
temporary per edge	sorting edge ($\times 3$)	12
	sort order	4
	sort key	4
	sort prefix sum (scan)	4

Table 1: *Mesh data structure after generating the edge information, where k is the number of vertex attributes.*

face and boundary edge quadrics. After this step, the opposite vertex and single edge flag arrays can be deallocated. Then the parallel simplification loop starts. First, the optimal collapse position and cost are computed per edge. Then the local cost minima are found and the associated collapses are performed. After updating the face and edge connectivity, the collapsed edges are removed. If no further collapses are possible without exceeding the threshold, the current level can be copied to a vertex and index buffer. The simplification loop is then continued with an increased threshold, until all required LODs are generated.

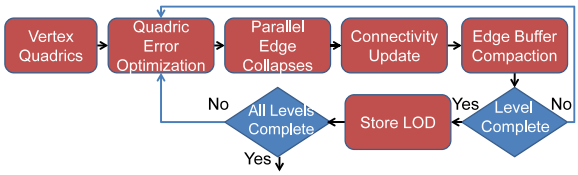

Figure 4: *The steps of the algorithm.*

The complete data structure maintained during simplification is shown in Table 2. For each vertex the quadric, the index of the adjacent edge with the minimal cost are required. Due to the symmetry of $\mathbf{Q}$ we only store the upper triangular matrix. Thus we only need $2k^2+6k+4$ bytes instead of $4k^2+8k+4$ bytes per quadric. Again, k is the number of vertex attributes. In addition, we need to flag inactive vertices to generate an IFS for each level-of-detail. To determine the minimal cost edge, we also need the minimal cost over all adjacent edges as temporary data. Later, when the collapse target of each vertex is required, we can reuse this array. For the faces, we only need the flag for degenerated triangles during construction of the indexed face set. Finally, we need to store the collapse cost and the optimal placement $\bar{\mathbf{v}}$ for each edge, together with the collapse state and a flag to mark degenerated edges. In total $2k^2+22k+120$ bytes per vertex are required, which is 204 bytes for $k=3$ and 324 bytes for $k=6$.

buffers	elements	per entry (bytes)
vertices	vertex VBO	$4k$
	vertex quadric	$2k^2+6k+4$
	min edge ID	4
	active flag	4
	min edge cost / target vertex	4
faces	index VBO	12
	active flag	4
edges	vertex indices	8
	edge cost	4
	optimal placement	$4k$
	collapse state	4
	active flag	4
temporary	perfix sum (scan)	4

Table 2: *Data structure used during simplification loop.*

5.1. Vertex Quadrics

The first step is the computation of the vertex quadrics which are the sum of all adjacent face quadrics. In addition, we need to compute and accumulate the boundary quadrics. As we have already flagged the boundary edges, we simply need to compute the virtual plane and add the corresponding quadric to the edge vertices. The complete parallel vertex quadric calcuation is shown in Algorithm 2. Note, that while we use atomic float addition in our implementation, parallel binning could also be used to accumulate the quadrics for GPU this operation.

```
foreach face f in parallel do
    i1,i2,i3 = get_face_indices(f)
    quad = compute_face_quadric(f)
    add_quadric(i1, quad)
    add_quadric(i2, quad)
    add_quadric(i3, quad)
foreach single_edge e in parallel do
    i3 = get_opposite_vertex(e)
    quad = compute_boundary_quadric(imin[e], imax[e], i3)
    add_quadric(imin[e], quad)
    add_quadric(imax[e], quad)
```

Algorithm 2: Parallel calculation of the vertex quadrics.

5.2. Quadric Error Optimization

The first step of the simplification loop is calculating the edge cost and to determine the possible collapses. The edge e can be collapse if its *cost* is at most ε^2, where ε is the error threshold. If the cost is below the threshold, the edge is marked as a collapse candidate. To compute the cost, we first need to find the optimal placement $\bar{\mathbf{v}}$ and then evaluate the quadric $\overline{\mathbf{Q}}$ for $\bar{\mathbf{v}}$. The edge quadric $\bar{\mathbf{v}}$ is the sum of the two vertex quadrics $\mathbf{Q}_{v_{min}}$ and $\mathbf{Q}_{v_{max}}$. Since $\mathbf{A}$ is a symmetric, positiv-semidefinite matrix we use the LDL decomposition (cholesky decomposition without square roots) to solve the linear equations. We also exploit the symmetry of $\overline{\mathbf{Q}}$ when calculating the quadric error. For each vertex we then store the minimal edge cost using the atomic min operation. Similar to the vertex quadric accumulation, we could use binning on GPU without atomic operations. Algorithm 3 shows the parallel quadric error minimization.

```
foreach edge e in parallel do
    quad = calc_edge_quadric(vertex_quadric[vmin],vertex_quadric[vmax])
    collapse_pos[e] = optimize_pos(quad)
    edge_cost[e] = calc_cost(quad, collapse_pos[e])
    if edge_cost[e] ≤ ε²
        collapse_state[e] = collapse
        atomic_min(min_edge_cost[vmin], edge_cost[e])
        atomic_min(min_edge_cost[vmax], edge_cost[e])
    else
        collapse_state[e] = no_operation
```

Algorithm 3: Parallel quadric error minimization algorithm.

5.3. Parallel Edge Collapses

The collapse of an edge is only possible if its cost is a local minimum. As we already stored the minimal cost per vertex, we now need to determine the edge with the associated cost. Then the edge can be collapsed if both vertices store a reference to it as minimal cost edge. For all other edges, the collapse flag is cleared. After the possible collapses are determined, they can be applied to the mesh. The collapse operation then simply moves vertex $v = v_{min}$ to its new position $\bar{\mathbf{v}}$, which is stored with the edge and marks $v_u = v_{max}$ as invalid and stores v as its target index. The new quadric $\mathbf{Q}_{\bar{v}}$ is the sum of the vertex quadrics $\mathbf{Q}_{v_{min}}$ and $\mathbf{Q}_{v_{max}}$. Algorithm 4 shows the parallel processing of the edge collapse operations.

```
foreach edge e in parallel do
    if collapse_state[e] == collapse
        cost = get_edge_cost(e)
        if min_edge_cost[vmin] == cost: edge_ID[vmin] = e
        if min_edge_cost[vmax] == cost: edge_ID[vmax] = e
foreach edge e in parallel do
    if collapse_state[e] == collapse
        if edge_ID[vmin] == e and edge_ID[vmax] == e
            v̄ = collapse_pos[e]
            vertex_quadric[vmin] += vertex_quadric[vmax]
            collapse_target[vmax]=vmin
            vertex_active[vmax]=false
```

Algorithm 4: Parallel edge collapse algorithm.

5.4. Connectivity Update

After performing the collapses, the indices of the adjacent faces and edges need to be updated (i.e. $f_{n1} - f_{n6}, f_l$, and f_r in Figure 3). Algorithm 5 shows the parallel index update. Here we use the collapse targets set during the edge collapses. If a face or edge becomes degenerated it is marked as invalid and will be removed in a subsequent stage.

```
foreach face f in parallel do
    update_indices(f,collapse_target)
    if degenerate(f): face_vaid[f]=false
foreach edge e in parallel do
    update_indices(e,collapse_target)
    if degenerate(e): edge_vaild[e]=false
```

Algorithm 5: Parallel index update.

5.5. Edge Buffer Compaction

The final step of the adaption is the compaction of the edge buffer. The removal of invalid edges is not only necessary for performance reasons, but also tells us when the simplification has finished. Algorithm 6 shows the edge compaction. At the end we can free the storage for the old edge buffer and thus gradually reduce the memory consumption. If no duplicate or degenerated edge was found, we skip the compaction and start with the LOD creation. Otherwise, we continue with the simplification loop. Note, that we sort the edges after each fifth iteration only as the speedup from the removed duplicates is less than the time required for sorting.

```
RADIX SORT edges in parallel by i_max
RADIX SORT edges in parallel by i_min
foreach edge e in parallel do
    e_p = get_previous_edge(e)
    if e_p==e or degenerate(e)
        set_edge_flag(e, 0)
    else
        set_edge_flag(e, 1)
COMPACT edges in parallel
```

Algorithm 6: Edge compaction algorithm.

5.6. LOD Creation

If no collapses were performed, the generated level can be stored. To store the mesh, we first compact the vertex buffer according to the active flag of the vertices. The compacted vertices are directly stored in a vertex VBO. Then we compact the indices according to the active flag of the faces and store them in an index buffer. If the number of faces is above a user specified threshold, we double the error threshold and generate another level. Otherwise, we can free all data structures except the original and generated VBOs and can now render them as static LODs.

6. Results

Our test system consists of a 3.333 GHz Intel Core i7-980X CPU with 6 GB DDR3-1333 main memory and an NVIDIA GTX580 (841/4204MHz). We used CUDA to implement the parallel simplification and generate indexed face sets for OpenGL. For comparison with loading precomputed LODs, we use a SATAII hard disk (8.5ms/64MB/7200rpm) with approximately 100 MB/s read speed. Table 3 gives an overview of the models we simplified. All models use position and normal as vertex attributes ($k = 6$). Additionally, the original model size (IFS) and the size of the generated LODs are shown.

Figure 1, 2, and 5 show some of the generated LODs and Table 4 gives an overview of all generated levels with their number of faces. The first level was generated with an error threshold of $\varepsilon = 0.1\%$ of the bounding box diagonal. With each level, the threshold is doubled and we stop LOD generation as soon as a level contains less than 10k triangles.

model	# vertices	# faces	IFS	LODs
Apache	445,836	807,365	19.4 MB	16.7 MB
St. Dragon	437,645	871,414	19.9 MB	17.5 MB
Welsh Dragon	1,105,352	2,210,673	50.5 MB	45.9 MB
Youthful	1,728,305	3,411,563	78.6 MB	70.6 MB
Awakening	2,057,930	4,060,497	93.5 MB	81.4 MB

Table 3: *Models used for evaluation.*

level	Apache	St. Dragon	Welsh Dragon	Youthful	Awakening
original	807,365	871,414	2,210,673	3,411,563	4,060,497
level 1	321,072	328,733	871,236	1,255,238	1,514,414
level 2	187,542	190,002	469,588	750.257	893,595
level 3	112,622	110,765	278,126	455,584	517,640
level 4	66,433	63,105	163,848	267.815	284,144
level 5	39,359	35,218	95,372	152,008	149,370
level 6	22,701	19,096	53,074	83,185	75,750
level 7	12,908	10,058	30,768	42,530	37,273
level 8	7,249	5,132	17,552	19,368	15,203
level 9	-	-	11,500	8,982	5,639
level 10	-	-	7,476	-	-

Table 4: *Generated levels with number of faces.*

Depending on the complexity of the model, 8 to 10 levels are generated this way.

Table 5 shows a comparison of our approach to the reference QSlim implementation of Garland and Heckbert [GH97]. The runtime complexity of their approach is $\mathcal{O}(N \log N)$, due to the required priority queue. In contrast to that, the complexity of our algorithm is $\mathcal{O}(N)$ as we only use radix sorting with fixed key length. Compared to CPU simplification, we achieve a speedup of 30 to 40 and can perform up to 2 million collapses per second. The simplification time is similar to the transfer time of the levels from HDD to the GPU and is significantly faster than the transfer time over a network. The total amount of consumed graphics memory is approximately 7 to 8 times higher than that of the original models. This is equal slightly less than the main memory consumed by CPU quadric error metrics.

	QSlim		our approach			
model	time (s)	k Op/s	memory	time (s)	k Op/s	speedup
Apache	8.0	55.7	136.4 MB	0.29	1537	28
St. Dragon	8.0	54.7	139.9 MB	0.28	1564	29
Welsh Dragon	22.2	49.8	354.2 MB	0.73	1519	31
Youthful	35.8	48.3	550.7 MB	0.89	1941	40
Awakening	43.9	46.9	655.6 MB	1.03	2003	43

Table 5: *Comparison of processing time and the number of operations per second with QSlim tested of our system.*

Compared to the vertex clustering algorithm of Lindstrom [Lin00] we have an approximate speedup of 10. Consequently, our method is 20 times faster than using BSP trees [GS02] and 70 times faster than octree vertex clustering [SW03]. Compared to the GPU implementation of vertex clustering by DeCoro and Tatarchuk [DT07], our method is 3 to 4 times slower. They implemented the method however for vertex position only ($k = 3$). With increasing quadric dimension, the difference vanishes since most of the time is

Figure 5: *Each second generated LOD of the Apache model. The first level is the original model.*

spent to optimize the vertex placement. In addition, even octree vertex clustering requires slightly more triangles to achieve the same quality.

Finally, we analyze the runtime of each step of the adaption and rendering algorithm in Figure 6. Already for $k = 6$, the most time consuming part of our algorithm is the quadric error minimization. As the LDL decomposition has a time cor er nu

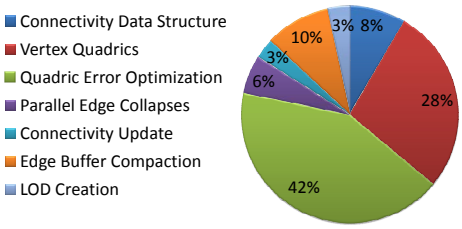

Figure 6: *Relative time of the adaption steps compared to rendering.*

7. Conclusion and Limitations

We have proposed a parallel implementation of the quadric error simplification developed by Garland and Heckbert [GH97]. By collapsing all edges with a local minimum of the collapse cost, the generated meshes are identical to those produced by the sequential algorithm for a given error bound. On a customer level graphics card, our method can generate a set of LODs for a model containing over 4 million faces in less than a second. This is comparable to loading the generated LODs from disk and significantly faster than network transfer.

The main limitation of our algorithm is that the computation of the target placement is rather expensive. With increasing number of attributes, this dominates the total runtime. Another limitation is, that we do not check for triangle flips during simplification. While this was unproblematic for the models we examined, it might produce visible artifacts for others.

A possible extension of our method would be the addition of vertex pair contractions. These could be integrated by adding an additional set of virtual edges before simplifying the mesh for a level. The maximum vertex distance would then be in the range of the error threshold.

References

[DMG10] 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.

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

[GH97] GARLAND M., HECKBERT P. S.: Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), SIGGRAPH '97, pp. 209–216.

[GH98] GARLAND M., HECKBERT P. S.: Simplifying surfaces with color and texture using quadric error metrics. In *Proceedings of the conference on Visualization '98* (1998).

[GS02] GARLAND M., SHAFFER E.: A multiphase approach to efficient surface simplification. In *Proceedings of the conference on Visualization '02* (2002), pp. 117–124.

[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.

[Lin00] LINDSTROM P.: Out-of-core simplification of large polygonal models. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), SIGGRAPH '00, pp. 259–262.

[LT97] LOW K.-L., TAN T.-S.: Model simplification using vertex-clustering. In *Proceedings of the 1997 symposium on Interactive 3D graphics* (1997), pp. 75–ff.

[Lue01] LUEBKE D. P.: A developer's survey of polygonal simplification algorithms. *IEEE Comp. Graph. Appl. 21* (2001).

[PH97] POPOVIĆ J., HOPPE H.: Progressive simplicial complexes. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), SIGGRAPH '97, pp. 217–224.

[RB93] ROSSIGNAC J., BOREL P.: Multi-resolution 3d approximations for rendering. *Geometric Modeling in Computer Graphics* (1993), 455–465.

[SG01] SHAFFER E., GARLAND M.: Efficient adaptive simplification of massive meshes. In *Proceedings of the conference on Visualization '01* (2001), pp. 127–134.

[SW03] SCHAEFER S., WARREN J.: Adaptive vertex clustering using octrees. In *SIAM Geometric Design and Computing* (2003).