World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:10, No:1, 2016

GPU-Accelerated Triangle Mesh Simplification Using Parallel Vertex Removal

Thomas Odaker, Dieter Kranzlmueller, Jens Volkert

Abstract—We present an approach to triangle mesh simplification designed to be executed on the GPU. We use a quadric error metric to calculate an error value for each vertex of the mesh and order all vertices based on this value. This step is followed by the parallel removal of a number of vertices with the lowest calculated error values. To allow for the parallel removal of multiple vertices we use a set of per-vertex boundaries that prevent mesh foldovers even when simplification operations are performed on neighbouring vertices. We execute multiple iterations of the calculation of the vertex errors, ordering of the error values and removal of vertices until either a desired number of vertices remains in the mesh or a minimum error value is reached. This parallel approach is used to speed up the simplification process while maintaining mesh topology and avoiding foldovers at every step of the simplification.

Keywords—Computer graphics, half edge collapse, mesh simplification, precomputed simplification, topology preserving.

I. Introduction

SIMPLIFICATION of meshes has been a well established research area for many decades. Reducing the complexity of meshes and reducing the triangle count to lessen the performance needed for rendering is still an important factor even considering the computational power of today's GPUs. A wide variety of algorithms has been devised to create a representation of a triangle mesh that contains less data than the original. The different approaches vary in terms of performance and quality of the resulting mesh. While some are designed to create a simplification that resembles the original mesh as closely as possible, others mainly provide short processing times while paying less attention to the quality of the resulting mesh. We present an algorithm that is designed to create an approximation of a triangle mesh. Our goal is to provide a parallel algorithm that can be executed on a GPU. This results in reduced processing times and can provide a significant speed up compared to previous algorithms. Despite the GPU-accelerated approach, our algorithm focuses on overall quality rather than real-time execution.

II. Related Work

Ever since the idea of mesh simplification has first been presented in [1], various approaches have been described that aim to reduce the complexity of a triangle mesh. A simplification algorithm usually relies on a simplification operator that is applied to a triangle mesh and takes care of the removal of either vertices or primitives.

T. Odaker and D. Kranzlmueller are with Ludwig-Maximilians Universitaet Muenchen, Germany (e-mail: odaker@a1.net, kranzlmueller@ifi.lmu.de).

J. Volkert is with Johannes Kepler University Linz, Austria (e-mail: jv@gup.jku.at)

The vertex pair collapse is a simplification operator that replaces two vertices of a mesh with a single one. The vertices are not required to be connected by an edge. This operator can close holes in the mesh and merge multiple surfaces. An example for the usage of this operator can be found in [11]. A quadric error metric is used to compute which pair of vertices to collapse and to determine a position for the new vertex that minimizes the changes to the mesh caused by the operation. The removal of vertices is iteratively repeated until a target number of remaining points is reached. Another simplification operator is the cell collapse presented in [2]. A number of cells is superimposed over the mesh and all vertices within a cell are collapsed into a single vertex. The number of cells is used to control the quality of the simplified mesh. While this operator can be used to compute a coarse mesh with fast processing times, the overall quality of the result is usually inferior to other operators. Several variations to cell creation and shape have been proposed to improve the quality of the simplification (e.g. [3], [4]). A parallel implementation of this approach that makes use of a GPU to accelerate the computation of the simplified mesh has been presented in [8].

The idea of the edge collapse is described in [6]. This operator replaces an edge of a triangle mesh with a single vertex, removing a vertex, an edge and one or two triangles from the mesh. Usually an error metric is chosen to assign an error value to each vertex or edge. Then the edge with the lowest error value is selected and collapsed. This process is iteratively repeated until a desired simplification is reached. The half edge collapse is a more specialised form of the edge collapse. It replaces an edge with one of its endpoints and does not allow the replacement position to be chosen freely. The edge collapse has the disadvantage that care has to be taken how an edge is to be collapsed. Some edge collapses can create mesh foldovers which have to be avoided (example in Fig. 1) [10]. A mesh foldover is usually created if the normal of a triangle that is manipulated by a collapse is rotated by more than 90 degrees.

One algorithm that relies on edge collapses is progressive meshes [9]. Here the executed simplification operations are stored in a data structure. A mesh is represented by its simplified version and this datastructure. The inverse operation to the edge collapse, the vertex split - which replaces a vertex with an edge - can then be executed until a desired grade of detail is reached. The data structure for this approach can be adapted to allow execution on a GPU [7] [12]. The approach in [13] also uses the edge collapse and tries to speed up iterative mesh simplification using edge collapses by executing multiple collapses in parallel, taking advantage of a GPU-accelerated implementation. In order to avoid foldovers

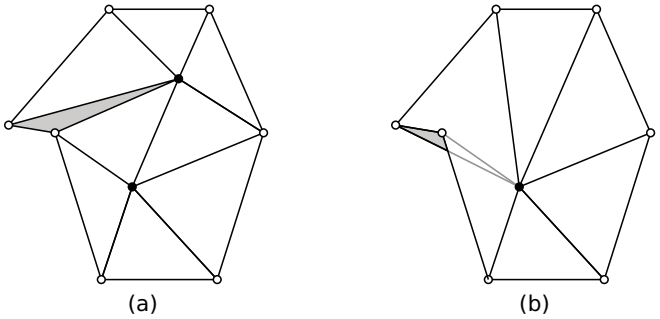

Fig. 1 Mesh foldover example: mesh before (a) and after (b) the collapse that caused a foldover

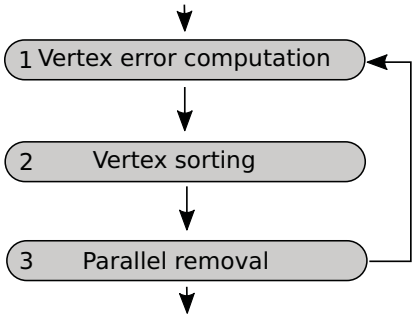

Fig. 2 Algorithm steps

a number of independent areas of the mesh is determined and an edge collapse executed in each area. This can achieve a significant speed up over a simpler, iterative solution.

The idea of vertex removal is presented in [5]. Here a vertex is chosen and removed from the mesh along with all edges and triangles that contain it. The resulting hole in the mesh is then triangulated. Removal of vertices and triangulation of holes is repeated until the desired simplification is created.

In [14], we presented our approach to mesh simplification. We described a parallel approach based on the half edge collapse that uses a set of per-vertex boundaries to allow parallel execution of half edge collapses while avoiding mesh foldovers from the perspective of the camera. [14] was mainly designed for very fast processing times and made compromises in terms of quality to achieve this speed up. We now present a modification of this approach that is designed for precomputing a simplified mesh. We modify our approach to work without the camera data while still avoiding foldovers and allowing a parallel execution. In addition we adapt how vertices are selected for removal to increase the quality of the coarse mesh.

III. Algorithm Overview

Our approach presented in [14] chose a number of vertices for removal upfront and then processed them in multiple iterative steps while allowing vertices marked for removal to be reclassified based on changes done to the mesh. In order to improve the quality of the coarse mesh we divert from this approach for the simplification method presented in this paper. We calculate a per-vertex error, then execute a step which removes N vertices with the lowest error in parallel. This is followed by the recomputation of the vertex error for all vertices remaining in the mesh. We execute the removal of a vertex by performing a half edge collapse. Since N vertices are to be removed in parallel, we define only edges that connect a vertex selected for removal to a neighbour that is not to be removed as a valid half edge collapse that may be executed.

The algorithm presented in this paper iteratively executes three steps (see Fig. 2): Computation of a vertex error for each vertex (1), sorting of all vertices according to the vertex errors (2) and removal of N vertices with the lowest vertex error (3). These three steps are repeated until either a number of remaining vertices/triangles is reached or the minimal calculated vertex error reaches a threshold. We use a modified version of the per-vertex boundaries presented in [14] to avoid foldovers in the resulting mesh. The original boundaries were created using the camera position which is not applicable to a simplification created in an off line preprocess. The application of the boundaries enables us to use a GPU-accelerated implementation of this algorithm which greatly speeds up the execution of the simplification.

IV. Vertex Error Calculation

The vertex error assigned to each vertex is used to determine which vertices will be removed in each iteration. We rely on the usage of the quadric error metric (QEM) presented in [11]. While the QEM originally uses the vertex pair collapse and not the (half) edge collapse, the authors point out that only choosing vertex pairs which are connected by an edge is an intended usage for their metric. The algorithm for the QEM creates a set of vertex pairs, orders them according to the error metric and executes the pair collapse with the lowest error. We have to modify this approach since - due to the per-vertex boundaries - we do not operate on vertex pairs but on vertices. Instead of selecting vertex pairs we analyse all vertices with regards to their neighbours. Every vertex is analysed individually. Each edge that connects a vertex V to one of its neighbours is a possible half edge collapse for the removal of V. The vertex error should represent the error caused by removing V and replacing it with one of its neighbours. The algorithm for the QEM always executes the collapse with the lowest error. We calculate the error value e(E) for an edge E that connects V to a neighbour V'. The value e(E) is the cost of replacing E with V' according to the metric in [11]. Since our algorithm removes N vertices with the lowest assigned error value we would ideally compute the value e(E) for each edge that contains a vertex V and use the minimum of these error values as the vertex error e(V). This causes two problems: First, our approach only allows collapses on edges between vertices that are supposed to be removed and those ones that remain unchanged in the current removal step. Given that the edge E with the lowest error e(E) connects the vertices V and V', the collapse is only possible if V' is not supposed to be removed at the same time, which cannot be guaranteed. Second, the per-vertex boundaries may block the collapse with the lowest error and force us to perform one with a high error according to the QEM instead. To avoid these issues we do not assign the minimum error of all the edges

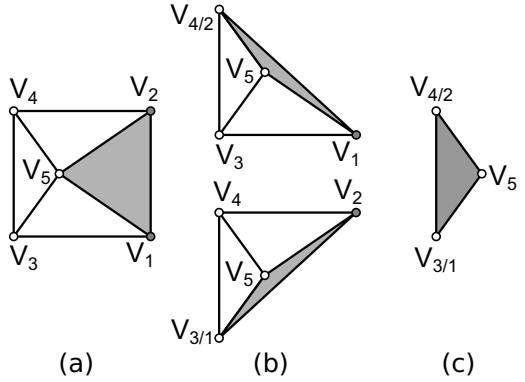

Fig. 3 Parallel removal of vertices: original mesh (a), individual half edge collapses (b) and parallel collapses (c)

but instead use the average error of all edges that contain a vertex V.

After all vertices have been assigned a vertex error, we perform a sort pass that orders the vertices according to their errors. The goal is to select the N vertices with the lowest vertex error and process them in the following removal step.

V. Parallel Vertex Removal

This step removes N vertices in parallel. Each vertex has at least two edges that connect it to neighbouring vertices. The removal step tries to select an edge for each vertex chosen for removal and executes a half edge collapse on it. It is however not guaranteed that a possible half edge collapse can be found for each vertex. By our definition a half edge collapse may only be performed on an edge that connects a vertex to be removed to a neighbour that is to remain unchanged in the current removal step. If all neighbours of a vertex selected for removal are currently also marked to be removed, no valid edge for a half edge collapse can be found and the vertex cannot be removed. This case is simply handled by not executing any operation on the vertex in the current removal step. It may be removed in a future iteration however. The vertex removal step is executed per vertex. It uses our per-vertex boundaries to eliminate all half edge collapses that could cause a mesh foldover and then selects one of the remaining ones for each vertex to be removed. The need for the boundaries is created by the parallel nature of the removal step. We define a mesh foldover to occur when the normal of a triangle manipulated by a collapse rotates by more than 90 degrees. While an iterative approach may simply check for rotations of triangle normals, this is not possible when two neighbouring vertices are subjected to a half edge collapse at the same time. Fig. 3 shows an example for this situation. While the two half edge collapses executed individually are valid, the parallel execution is not.

Our per-vertex boundaries presented in [14] were designed for view-dependent mesh simplification. For that purpose, they were constructed using the camera position to guarantee that no foldover could occur form the perspective of the camera. For the computation presented in this paper, however, we do not have a camera position available and therefore need to alter the boundary construction. Boundary construction differs depending on how many vertices selected for removal a triangle is made up of. For each vertex to be removed the boundaries B(V) are a set of planes. Each triangle that contains a vertex adds one or two planes to this set. For boundary testing all triangles containing a vertex V have to be found first. Then the number of removal candidates in each triangle is determined, the appropriate boundary planes are constructed and added to per-vertex boundary B(V). Any edge that has been selected as a possible half edge collapse is tested against all planes in B(V). Testing is done by checking if an intersection exists between the edge and any plane in B(V). If one can be found, the collapse may cause a foldover and is disregarded. Only half edge collapses that do not have an intersection with any plane in B(V) are considered valid and can safely be executed.

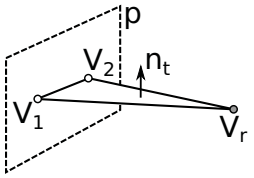

Fig. 4 Plane for boundary 1

A. Boundary Construction

Our approach used the vector between the camera position and a point on the triangle to construct the boundary planes. For the computation, here we substitute this vector with the triangle normal since no camera position is available. It is however not possible to use the triangle normal of an intermediate mesh created by one or more removal steps. Since the triangle normal can be changed by half edge collapses, using the triangle normal of the intermediate mesh could allow further rotation of the normal with each iteration. This could effectively rotate it by more than 90 degrees compared to the triangle of the original mesh which may cause a mesh foldover. To avoid this issue the triangle normal of the triangle in the original mesh is stored and maintained for boundary computation, even when the triangle is modified by one or more of its vertices being removed by a half edge collapse. As mentioned earlier, boundary construction differs, depending on how many vertices of a triangle are marked for removal.

1) Boundary 1: In the first case, a single vertex is selected for removal in the triangle (the vertex we construct the boundary for). Here only a single plane needs to be constructed and added to the set of per-vertex boundaries. Given the triangle is constructed from vertices V_1, V_2 and the vertex selected for removal V_r with the triangle normal of the unmodified triangle in the original mesh being n_t, the boundary plane p is constructed using the points V_1, V_2 and $V_1 + n_t$ (example in Fig. 4).

2) Boundary 2: Boundary 2 is used when the triangle contains two vertices marked for removal. Here two planes are constructed. For a triangle consisting of V and the vertices marked for removal V_{r1} and V_{r2}, the plane p_1 is constructed using the points V, $V + (V_{r2} - V_{r1})$ and $V + n_t$. The second plane p_2 contains the points V, $(V_{r1} + V_{r2}) * 0.5$ and $V + n_t$.

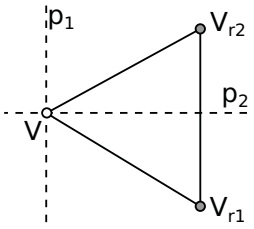

Fig. 5 Planes for boundary 2 (top view)

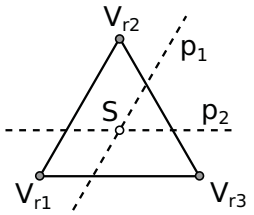

Fig. 6 Planes for boundary 3 (top view)

Fig. 5 shows an example for the planes looking down onto the triangle.

Unlike boundary 1, these boundary planes can block valid combinations of edge collapses. While this can potentially limit the simplification, it is an accepted trade off to increase the parallelism of the execution and avoid any communication between the simplification operations.

3) Boundary 3: The third and last boundary is used when all three vertices of a triangle are marked for removal. Here two planes are constructed for each vertex using the centroid S.

For a triangle V_{r1}, V_{r2}, V_{r3} plane p_1 for vertex V_{r1} contains the points S, $S + n_t$ and $S + (V_{r2} - V_{r1})$. Plane p_2 for V_{r1} contains S, $S + n_t$ and $S + (V_{r3} - V_{r1})$. Fig. 6 shows an example for the planes looking down onto the triangle.

B. Half Edge Collapse Selection

After boundary testing is completed each vertex selected for removal has a list of valid half edge collapses that would not cause a foldover. One of those edges needs to be selected and removed from the mesh. We rely on the use of the quadric error metric (QEM, [11]) that has already been chosen for the calculation of the vertex error. Each edge is assigned an error value according to the QEM. Given that an edge E is made up of the vertices V_r and V' where V_r is selected for removal, the replacement position for the edge is V'. We construct the matrix according to the QEM and calculate the error using V' as the replacement position. The valid edge with the lowest value is chosen and the half edge collapse executed which results in the removal of V_r from the mesh.

VI. Implementation

We have implemented our algorithm using Nvidia CUDA to accelerate the execution by taking advantage of the GPU. Our algorithm - especially the per-vertex boundaries - is designed to isolate half edge collapses and avoid all need for communication between individual operations which makes it well suited for execution on a GPU. To speed up the computation we maintain a list of neighbours for each vertex. This is needed for the vertex error calculation as well as the boundary computation during the parallel removal step. It does however have the disadvantage that the neighbour list needs to be updated after each parallel removal step to take executed half edge collapses into account and to make sure that the correct neighbours can be selected. We have to modify the algorithm steps to include the computation of the neighbour list (Fig. 7). The computation of the neighbour list can be reduced by only updating it for vertices that were neighbours of vertices subjected to a half edge collapse in the previous parallel removal step to minimize the necessary runtime for the operation.

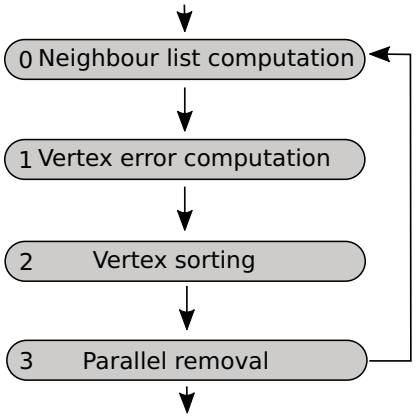

Fig. 7 Implemented algorithm steps

In step 1 (Fig. 7) ,we write a list of vertex errors and then use it to sort all vertices remaining in the mesh. For this operation, the Nvidia Thrust library is used which offers GPU accelerated sorting by key values. The algorithm operates on a list of vertices and indices. Whenever a vertex is removed from the mesh, its replacement position is stored. This value is then used to update the index list, which serves as a base for the computation of the neighbour list. After the simplification process has been completed, we can use the resulting index list of the last iteration to build the triangle indices for the coarse mesh the algorithm created.

VII. Results

We used our implementation to test the algorithm with a Nvidia Geforce GTX 670 GPU with 1344 cores. Since we are applying an algorithm that removes up to N vertices in a single pass, we have to run several iterations until the simplification process is completed. We observed that the number N is crucial to the processing time for the simplification as well as the overall quality of the coarse mesh. A high number of vertices being removed in each pass can greatly reduce the number of necessary iterations and therefore reduce the overall runtime of the simplification. On the other hand, however, the quadric error metric used to select which vertices are removed is not able to predict parallel operations. While collapsing one edge at a time always chooses the ”optimal” edge, this is not possible for the parallel approach. The more vertices are removed in parallel, the greater the deviation from this optimal sequence may occur.

TABLE I
VERTICES AND TRIANGLES OF THE ORIGINAL MODELS

	vertices	triangles
Stanford Bunny	35 947	69 451
Armadillo	172 974	345 944
Dragon	437 645	871 414
Happy Buddha	543 652	1 087 716

TABLE II
SIMPLIFICATION TIMES AND NUMBER OF ITERATIONS FOR ALL MODELS AND TEST CASES

	50%	25%	5%	1%
Stanford Bunny				
time(ms)	27.9	42.7	57.2	74.5
iterations	4	7	12	18
Armadillo				
time(ms)	358.7	496.7	590.1	598.8
iterations	33	49	63	66
Dragon				
time(ms)	856	1237	1513	1567
iterations	54	81	104	111
Happy Buddha				
time(ms)	1101	1712	2088	2129
iterations	68	102	130	136

The test cases were executed using various numbers of vertices being removed in each pass and the overall results presented here being averages to represent the performance of the algorithm. We used four different models to test our approach that were taken from the Stanford 3D Scanning Repository: ”Stanford Bunny”, ”Armadillo”, ”Dragon” and ”Happy Buddha”. The numbers of vertices and triangles for these models are shown in Table I.

We used all four models to run several test cases. We set various targets for the simplification using a target number of vertices expressed by a percentage of the number of vertices of the original model. For these test cases we measured the overall runtime until the simplification process was completed as well as the number of iterations necessary to compute the coarse mesh. The percentage targets for the simplification were set at 50%, 25%, 5% and 1% of the vertices of the original mesh. Figs. 9 and 10 show the resulting meshes of all test cases for the models ”Dragon” and ”Happy Buddha”. These are the two meshes with the highest polygon count in our tests. The original meshes are not shown in this comparison, the left images show the 50% target with decreasing number of vertices towards the right. The rightmost image (the image in the second line for the dragon) shows the 1% target.

Table II shows the results for all test cases and all models that were measured using our implementation. It should be noted that these results are averages of several tests using a different number of vertices being removed in each iteration. In general we noticed that a higher number N reduces the number of iterations and the processing time, while having an impact on the quality of the resulting mesh. Some unfavourable triangles (e.g. long and narrow triangles) can occur due to the parallel execution being processed in an isolated fashion and not taking the chosen half edge collapse for a neighbouring vertex into account. These artefacts are however reduced when lowering parallelism. The best trade off between speed and overall quality was achieved when starting the simplification

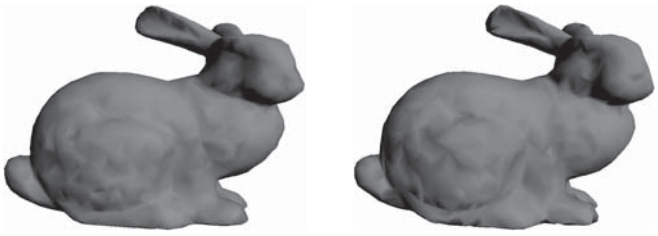

Fig. 8 Coarse mesh comparison: 5% target with N=1344 (a) and N=5376 (b) vertices removed in each iteration

with a high number of vertices being removed and reducing this number in later iterations to avoid these artefacts in the final coarse mesh. Fig. 8 shows a comparison between two simplifications of the Stanford Bunny. Both were created using the 5% target and a fixed number N of removed vertices in each iteration. On the left the result for N=1344 is shown. On the right the simplification was created using N=5376. The simplification using N=1344 took 83ms to be completed, while the result for N=5376 was computed in 51ms. There are some rougher edges visible on the coarse mesh computed using higher parallelism, especially around the paws of the Stanford Bunny, but also along the tail.

A more detailed comparison of the tail of the Stanford Bunny is shown in Fig. 11. Here the left side shows the simplification created using less parallelism again. A smoother triangulation is visible compared to the result on the right.

VIII. COMPARISON TO OTHER APPROACHES

We compare the results of our algorithm to other approaches. We limit this comparison to simplifications created using the edge collapse to achieve valuable results. Our main goal was to see the difference in quality of the overall mesh in contrast to an iterative simplification as well as the speed up compared to iterative and other parallel approaches.

A. Iterative Simplification

We compared our approach to an iterative simplification using Meshlab to simplify the ”Dragon” mesh with a target of 50% and 5%. We did not measure the simplification time for the iterative approach as Meshlab already takes over 5 seconds for the 50% target.

Fig. 12 shows the comparison of the two targets. We can see some loss of quality in the comparison in the 5% target, but very little visible difference in the 50% target. The quality in the 5% target for our algorithm could be further increased by removing less vertices in parallel in the later iterations.

B. Comparison to Parallel Approaches

The algorithm presented in [13] has a similar idea as our approach. Instead of allowing removal of neighbouring vertices, it searches independent areas in which half edge collapses are executed. This results in the same issues as we found with our algorithm: an increase in parallelism causes a decrease of overall quality of the resulting coarse mesh

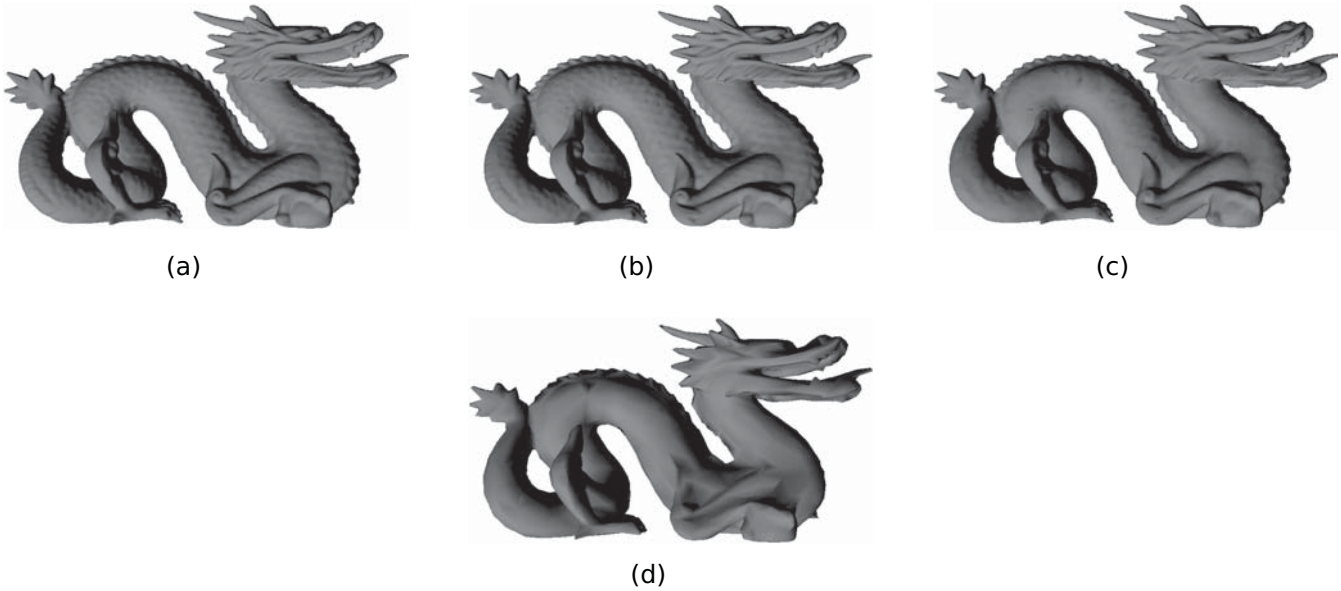

Fig. 9 Meshes created for the model "Dragon" by the simplification for 50% (a), 25% (b), 5% (c) and 1% (d) targets. See Table II for runtimes.

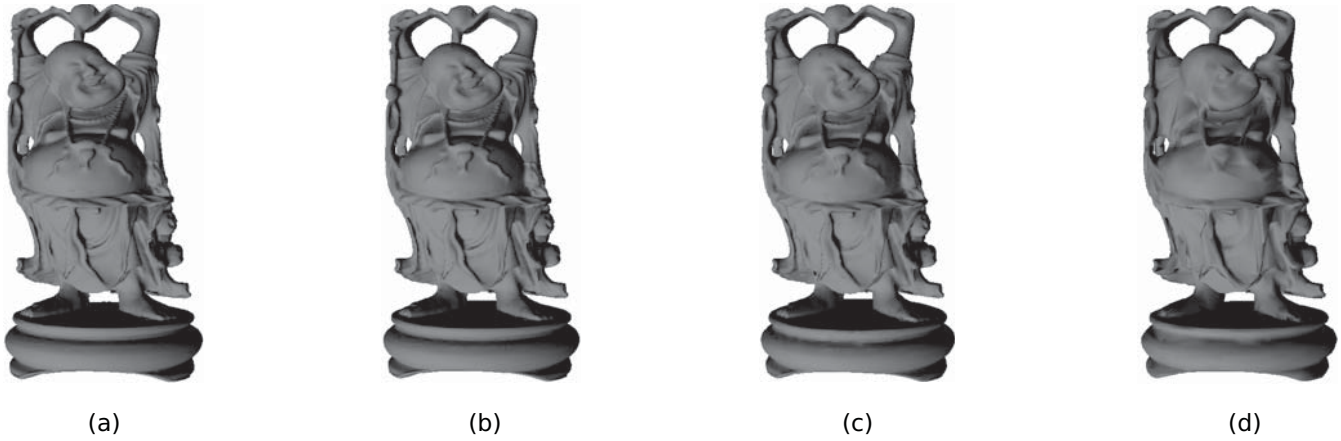

Fig. 10 Meshes created for the model "Happy Buddha" by the simplification for 50% (a), 25% (b), 5% (c) and 1% (d) targets. See Table II for runtimes.

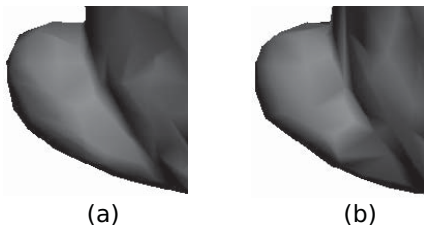

Fig. 11 Detailed coarse mesh comparison of tail: 5% target, N=1344 (a) and N=5376 (b) vertices removed in each iteration

when compared to a strictly iterative simplification [13]. While [13] achieves a significant speedup compared to the iterative simplification, the algorithm is significantly slower than our approach. The model "Gargoyle" used for benchmarks in [13] consists of 450 000 vertices. Computing a simplification with a target of 90% already takes over 850 ms using the approach from [13], while the 5% target is computed in almost 5.7 seconds. We use the Dragon model from our tests with about 440 000 vertices to compare these results. Our approach

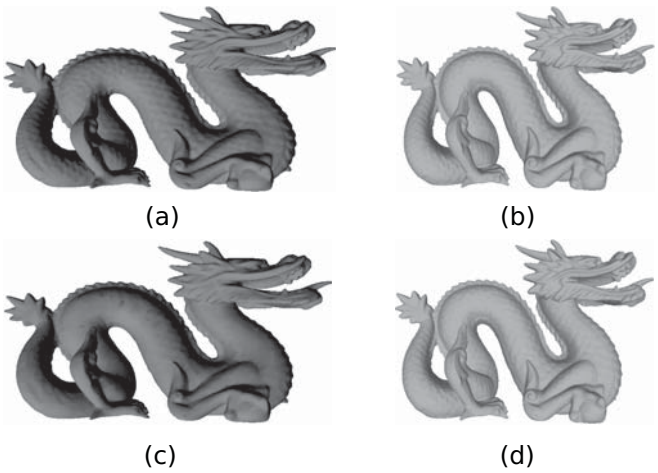

Fig. 12 Comparison: Our approach (a, c) and iterative simplification (b, d) with 50% (a, b) and 5% (c, d) target

creates the 50% target in 856 ms while computing the 5% target in under 1.6 seconds which shows a significant speedup over the algorithm presented in [13].

IX. Conclusion

We have devised an algorithm for computing a mesh simplification that can make use of the parallel processing power of the GPU to reduce processing times. Our approach is considerably quicker than previous work while maintaining good overall quality of the resulting meshes, but is not designed for real-time simplification. The number of vertices removed in each pass of the process has been observed to be a key factor to managing the trade off between the processing time and the quality of the algorithm. Even with a relatively high number of parallel removals we observed a good quality result that is comparable to previous approaches while gaining a significant speed up.

X. Future Work

Since the most important factor for this algorithm proved to be the number of vertices reduced in each iteration, this factor should be subject to further development. The highest quality for the coarse mesh was achieved with a low parallelism, which had an impact on processing times. The algorithm can be improved by dynamically choosing the number of vertices to remove at each iteration to increase quality with minimal impact on processing time. Overall a lower number should be used in later iterations to increase quality. On the other hand a threshold for the error value of removed vertices might be applied to further refine the selection. This might also lead to usage of a hybrid approach, where the parallel algorithm presented here is used to quickly remove a large number of triangles whilst then switching to a CPU implementation for the latter iterations and reduce the overhead of the GPU implementation whilst further improving the quality of the coarse mesh. Another factor that has potential to greatly increase overall quality of the coarse mesh is the use of feature extraction. This has to be added to the error metric used to choose the vertices to be removed and the executed half edge collapses to better conserve defining features of the original model.

References

[1] Clark, J. H. Hierarchical geometric models for visible surface algorithms, Com. of ACM 19, No. 10, pp.547-554, 1976
[2] Rossignac, J., and Borrell, P. Multi-resolution 3D Approximations for Rendering Complex Scenes, Modeling of Computer Graphics: Methods and Applications, pp.455-465, 1992
[3] Schaefer, S., and Warren., J. Adaptive vertex clustering using octrees, Proceedings of SIAM Geometric Design and Computing 2003, Vol. 2, pp.491-500, 2003
[4] Low, K.-L., and Tan, T., S., Model simplification using vertex-clustering, SI3D Proceedings 1997, pp.75-ff., 1997
[5] Schroeder, W., J., Zarge, J., A., and Lorensen, W., E. Decimation of triangle meshes, ACM SIGGRAPH Computer Graphics Vol. 26, No. 2, pp.65-70, 1992
[6] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., A., and Stuetzle, W. Mesh optimization, ACM SIGGRAPH Proceedings 1993, pp.19-26, 1993
[7] Hu, L., Sander, P., V., and Hoppe, H. Parallel view-dependent refinemnet of progressive meshes, I3D 2009 Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games, pp.169-176, 2009
[8] DeCoro, C., and Tatarchuk, N. Real-time mesh simplification using the GPU, I3D 2007 Proceedings of the 2007 Symposium on Interactive 3D Graphics Vol. 2007, pp.161-166, 2007
[9] Hoppe, H. Progressive meshes, ACM SIGGRAPH 1996 Proceedings, pp.99-108, 1996
[10] Xia, J., C., El-Sana, J., and Varshney, A. Adaptive real-time level-of-detail-based rendering for polygonal models, IEEE Transactions on Visualization and Computer Graphics Vol. 3, No. 2, pp.171-187, 1997
[11] Garland, M., and Heckbert, P., S. Surface simplification using quadric error metrics, SIGGRAPH Proceedings 1997, pp.209-216, 1997
[12] Hu, L., Sander, P., and Hoppe, H. Parallel view-dependent level of detail control, IEEE Transactions on Visualization and Computer Graphics Vol. 16, No. 5, pp.718-728, 2010
[13] Papageorgiou, A., and Platis, N. Triangular mesh simplification on the GPU, The Visual Computer: International Journal of Computer Graphics Vol. 31, Issue 2, pp.235-244, 2015
[14] Odaker, T., Kranzlmueller, D., Volkert, J. View-dependent Simplification using Parallel Half Edge Collapses, WSCG 2015 Conference Proceedings, pp.63-72, 2015