ISSN 2464-4617 (print) WSCG 2016 - 24th Conference on Computer Graphics, Visualization and Computer Vision 2016
ISSN 2464-4625 (CD-ROM)

# View-dependent Triangle Mesh Simplification using GPU-accelerated Vertex Removal

Thomas Odaker
Ludwig-Maximilians-Universitaet
Muenchen, Germany
odaker@a1.net

Dieter Kranzlmueller
Ludwig-Maximilians-Universitaet
Muenchen, Germany
kranzlmueller@ifi.lmu.de

Jens Volkert
Johannes Kepler University
Linz, Austria
jv@ica.jku.at

## ABSTRACT

We present an approach to view-dependent triangle mesh simplification based on vertex removal, which focuses on allowing the execution of a large number of operations in parallel. The individual vertex removal operations are designed to be applied without any need for communication or synchronisation between operations, thus allowing an efficient implementation on modern GPUs to reduce the computation time for the coarse mesh.
Since we cannot compute the entire simplification in a single step and have to perform several iterations of parallel vertex removal, we aim to maximize the number of vertices removed from the mesh in each iteration to efficiently use the available hardware and reduce the number of necessary iterations. The removal operation is based on the half edge collapse and avoids mesh foldovers and topological inconsistencies at each step.

## Keywords

mesh simplification, level of detail, half edge collapse, computer graphics, view-dependent simplification, real-time rendering

## 1 INTRODUCTION

Simplification of triangle meshes is a commonly used approach to reduce geometric data and the performance necessary for processing a mesh. Ever since it was first introduced in [Cla76a], a wide variety of techniques and algorithms that compute a coarse mesh have been presented.

In [Oda15a] we introduced our approach to view-dependent simplification that is designed for an execution on a GPU. In this paper we introduce further developments and improvements to this approach that increase parallelism and quality of the simplification. This leads to a significant reduction of the number of necessary iterations and shorter processing times.

## 2 RELATED WORK

We classify simplification algorithms into those used in a preprocessing step and algorithms executed at runtime. Creating a coarse mesh in a preprocessing step eliminates the need for fast processing times and allows higher quality simplifications. Creating a coarse mesh in real-time before rendering each frame enables view-dependent simplification that minimizes visible artefacts for a given camera position, but calls for fast run times not to slow down the overall rendering process.

A variety of simplification operators have been defined, some of which have been adapted for real-time usage and even execution on modern GPUs to further speed up the simplification process. We now present the three operators most important to us:

**Vertex clustering** First presented in [Ros92a] this operator superimposes a number of cells over the volume of a mesh. All vertices within a cell are collapsed into a single vertex and the model data is updated accordingly. This approach can be used for fast processing times, but it may create low quality simplifications since the topology of the mesh is not preserved. In [DeC07a] a GPU-accelerated algorithm based on this operator designed for real time simplification is presented.

**(Half) Edge collapse** The edge collapse originally presented in [Hop93a] is the replacement of an edge and its two endpoints with a single vertex. The half edge collapse is a more restricted version that replaces the edge with one of its endpoints. The edge collapse is widely used. One common example for an algorithm relying on the edge collapse is progressive meshes ([Hop96a]). An improved version of this algorithm (modified for execution

on a GPU and real-time execution) is presented in [Hu09a] and [Hu10a].

**Vertex removal** This operator presented in [Sch92a] removes a vertex from the mesh and retriangulates the resulting hole. Different algorithms for retriangulation are available, one of them being the half edge collapse. For a vertex to be removed, one of the edges to a neighbouring vertex is chosen and a half edge collapse executed on it. The replacement position is the neighbouring vertex.

In [Oda15a] we presented our approach to triangle mesh simplification, with further discussion of details and results in [Oda15b]. All vertices of a given triangle mesh are analysed and classified as to be removed or to remain in the mesh. A parallel vertex removal technique based on half edge collapses is used to remove as many vertices as possible in parallel. Additionally we introduced a set of per-vertex boundaries that prevent mesh foldovers and topological inconsistencies despite the parallel execution.

In this paper we present further development and improvements to this algorithm that aim to increase parallelism, decrease runtime and improve the quality of the coarse mesh.

## 3 PREVIOUS WORK

This paper is based on our previous work presented in [Oda15a]. We further discussed details and results of this approach in [Oda15b]. In this section, we present an overview of our previous approach that we improved on.

The algorithm executes three steps: classification, parallel removal and reclassification (Fig. 1).

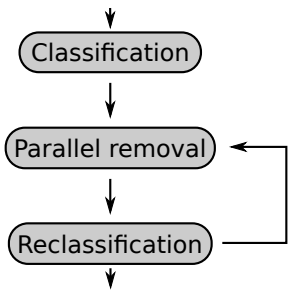


Figure 1: Algorithm steps

**Classification** This step analyses all vertices of a triangle mesh and marks them as to be removed or to remain. A simple metric that relies on the average distance between the tangential plane of a vertex and the neighbours is used to compute a vertex error, which is used for classification. This metric was primarily chosen for short computation times. Additionally, to prevent all vertices from being selected for removal, we introduced a layered error manipulation, artificially increasing the vertex error of some vertices to guarantee that they remain in the mesh. This leads to improved parallelism and guarantees the functionality of the algorithm.

**Parallel removal** We define any vertex that is selected for removal with at least one neighbour that is to remain as a removal candidate. A parallel removal step tries to remove all current removal candidates simultaneously, using a set of per-vertex boundaries to prevent foldovers and topological inconsistencies. The removal of vertices changes edges of the mesh which results in a new set of removal candidates being created.

**Reclassification** After each removal step, the classification of all vertices still selected for removal is updated to adapt to changes to the mesh and improve the quality of the coarse mesh.

Several iterations of parallel vertex removal and reclassification are executed. Due to how removal candidates are defined, only a part of vertices selected for removal can usually be processed in a single iteration. When a vertex is removed from the mesh, one or more edges are changed. If a vertex $V$ is removed by a half edge collapse, any neighbour of $V$ selected for removal becomes a removal candidate. This approach allows all vertices selected for removal to be processed over the course of several iterations.

We also addressed the issue of deadlocks. The per-vertex boundaries are designed to allow the removal of neighbouring vertices without any communication and can block valid combinations of half edge collapses. This potentially causes a situation, where multiple neighbouring vertices block each others' removal. Due to the parallel nature of the algorithm, it is not possible to identify deadlocks until the subsequent iteration. This can result in additional iterations being necessary to resolve deadlocks and delaying the completion of the simplification.

While our original approach worked well and resulted in fast processing times, we identified several shortcomings of the algorithm that limited parallelism and potentially the quality of the simplification. In this paper we present an improved algorithm that is based on our previous work and focuses on improving the classification and overall parallelism.

## 4 OVERVIEW

We identified several limiting issues with our original approach that we want to improve on:

**Vertex error manipulation** During the vertex classification we introduced a vertex error manipulation

based on points arranged in a grid to avoid all vertices being selected for removal. While this guaranteed that the algorithm could remain functional at all times, it ignored irregular triangle sizes and did not always suit the given mesh well.

**Removal candidates** We defined a removal candidate as a vertex selected for removal that has at least one neighbour that is to remain in the mesh. In each iteration, all removal candidates are processed. This results in a potentially large number of vertices selected for removal being ignored, since they currently have no neighbour that is to remain in the mesh, which reduces parallelism.

**Replacement positions** Only vertices selected to remain in the mesh are defined as valid replacement positions, potentially ignoring neighbours that would be better suited as a replacement position.

For the improved algorithm we rely on the vertex error we presented in [Oda15a]. The layered error manipulations used in our original approach helped to guarantee the functionality of the algorithm and improve the parallelism. In this paper we still rely on error manipulation to guarantee that some vertices are always selected to remain in the mesh. This is, however, not done using regularly spaced points in multiple layers. Instead a number of vertices is selected based on the minimum number of edges between them and their vertex error is set to a very high value to guarantee they are always selected to remain in the mesh.

To improve parallelism, we introduce the concept of auxiliary vertices (see subsection 5.1). For a mesh a set of auxiliary vertices is precomputed. An auxiliary vertex, that is not currently a removal candidate, can be used as a replacement position for neighbouring vertices, potentially greatly increasing the parallelism of the algorithm.

After computing the classification for all vertices of a mesh, the initial removal candidates are computed. We modify the definition of a removal candidate to include vertices selected for removal, that have at least one neighbouring auxiliary vertex, that is not currently a removal candidate. We execute the parallel removal step based on the half edge collapse using the per-vertex borders from [Oda15a] to prevent foldovers and topological inconsistencies.

The last step is the reclassification of vertices. It updates the vertex error of vertices selected for removal, that have not yet been removed. This step is used to adapt the classification to changes made in the previous parallel removal step and improve the quality of the simplification.

# 5 CLASSIFICATION

The classification step in [Oda15a] computes an error value for each vertex of a mesh. This error represents the difference between the mesh before and after a vertex removal operation. The original metric computes the error based on the geometric data and then scales it using the view vector and position of the camera. All vertices with a scaled error below a user-defined threshold are selected for removal. In our previous work we identified a problem with this approach: all vertices being selected for removal prevent the execution of our algorithm. Additionally a low number of vertices remaining in the mesh results in few removal candidates, which reduces parallelism. In order to avoid these problems we introduced a manipulation of the vertex error. A small number of vertices is assigned a very large vertex error to guarantee, that they always remain in the mesh.

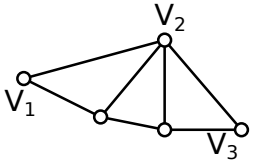


Figure 2: Minimum number of edges between vertices

The approach to selecting vertices for the error manipulation we propose in this paper is based on a minimum number of edges between two vertices with a manipulated error value. An example for the minimum number of edges between vertices is shown in Fig. 2. The vertices $V_1$ and $V_2$ have a minimum of 1 edge, while $V_1$ and $V_3$ have a minimum of 2 edges between them. The amount of vertices selected by this algorithm determines the maximum simplification and is controlled by the number $N$ of edges. This value is chosen by the user. The number $N$ can be used to influence the grade of the maximum simplification as well as the processing time: smaller $N$ can reduce the number of necessary iterations for the simplification and improve processing times, while larger $N$ allow the removal of additional vertices and can create a coarser mesh. Vertices are selected based on the maximum curvature of the surface to better maintain the shape of the object.

The first step to find vertices for error manipulation is to calculate the principal curvatures and store the maximum curvature of the surface in each vertex. We want to determine a set $G$ of vertices that are guaranteed to remain in the mesh.

Initially, the vertex with the maximum curvature is selected and added to $G$ ($G = \{g_0\}$). It is the first vertex selected for error manipulation and the starting point for the further steps of the algorithm. Given the maximum curvature for each vertex and the value $N$, the vertices are selected as follows:

1. Find all vertices $C = \{c_0, c_1, ..., c_n\}$ that have a minimum of $N$ edges to any vertex in $G$ and a maximum of $N$ edges to at least one vertex in $G$.

2. Find the vertex $c_i$ with the maximum curvature of all vertices in $C$.

3. Add $c_i$ to $G$.

These three steps are repeated until no more vertices can be found in step 1. Then all vertices in $G$ are assigned a very large error value to guarantee that they remain in the mesh.

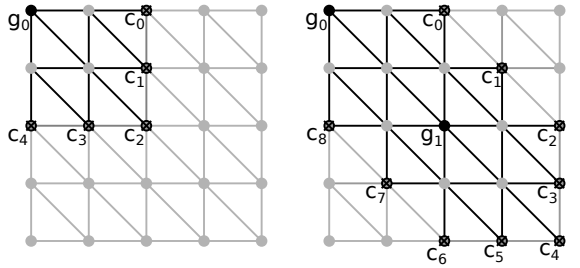


Figure 3: Vertex selection for error manipulation

Fig. 3 shows an example for this approach. On the left side the top left vertex ($g_0$) has been selected as the initial vertex in $G$ and the list C has been filled in step 1 of the first iteration ($C = \{c_0, c_1, c_2, c_3, c_4\}$). For this example N has been set to 2. The right side illustrates the mesh after the first iteration is completed. The central vertex ($g_1$) has been selected and added to $G$. The set $C$ has been updated in step 1 of the second iteration, resulting in a new set of candidates $C$ for error manipulation.

The layered approach also has the goal to create additional removal candidates. Besides Layer 0 with vertices that always remain in the mesh, additional layers are created. Vertices selected in these have their vertex error increased to have additional vertices remain in the mesh at certain distances between the object and the camera. While this approach increases parallelism and reduces processing times, it can result in a simplified mesh, where the vertices are arranged in a grid like structure. Fig. 4 shows an example for a simplification of the Stanford Bunny illustrating this phenomenon. In this paper we replace the layered error manipulation with the concept of auxiliary vertices. In a preprocessing step a number of vertices with an unmodified vertex error is selected and marked as auxiliary vertices. During the simplification process any auxiliary vertex, that is not currently a removal candidate, can be used as a replacement position for neighbouring vertices marked for removal. Auxiliary vertices can be selected for removal. They are no longer considered auxiliary vertices and become removal candidates once they have a neighbour that is to remain in the mesh.

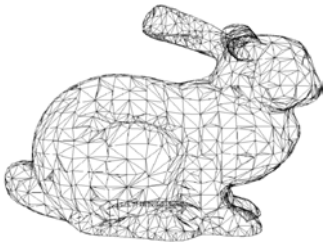

Figure 4: Simplified mesh with vertices arranged in a grid-like structure

After the vertices for error manipulation have been selected, we determine the auxiliary vertices.

## 5.1 Auxiliary vertex computation

Since auxiliary vertices can be replacement positions for neighbouring vertices selected for removal, we modify the definition of the removal candidates. In [Oda15a] any vertex selected for removal, that has at least one neighbour that is to remain in the mesh, is a candidate. In this paper, we include the auxiliary vertices in this definition so that a removal candidate is any vertex of a given mesh that:

- is selected for removal.
- has at least one neighbour, that is to remain in the mesh, or has at least one neighbour, that is an auxiliary vertex, which is not currently a removal candidate.

The idea of the auxiliary vertices is to make sure that as many vertices selected for removal as possible can be processed in any iteration. Since auxiliary vertices are determined in a preprocessing step and the classification is done at runtime, we do not know which vertices are to remain in the mesh when selecting the auxiliary vertices (with the exception of vertices in $G$).

For the purpose of computing auxiliary vertices, we assume that all vertices in $G$ are to remain in the mesh and all other vertices are selected for removal. Given this assumption, we want all vertices of the mesh to be either vertices selected to remain in the mesh, auxiliary vertices or removal candidates. Based on this we compute auxiliary vertices as follows:

**1. Find removal candidates** First the list of removal candidates is determined, taking into account all vertices in $G$ selected to remain in the mesh as well as the current list of auxiliary vertices (empty list initially).

**2. Auxiliary vertex candidates** The list of candidates $C = \{c_0, c_1, ...c_n\}$ for the auxiliary vertices is selected. It contains all vertices chosen for removal, that have a neighbouring removal candidate and are not currently a removal candidate or an auxiliary vertex.

**3. Auxiliary vertex selection** Selection of one of the candidates in $C$ as auxiliary vertex. This is done using the candidate with the maximum curvature.

These steps are repeated until all vertices of the mesh not in $G$ are either removal candidates or auxiliary vertices. This approach allows a greater number of removal candidates and therefore increases parallelism and prevents a large number of vertices selected for removal from being ignored during the parallel removal step.

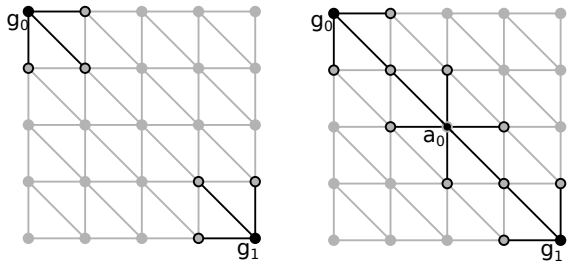


Figure 5: Auxiliary vertex selection

Fig. 5 shows an example of the selection of the auxiliary vertices. On the left side the vertices in the upper left ($g_0$) and bottom right corner ($g_1$) are in $G$. Their neighbours are assumed to be removal candidates. On the right side the center ($a_0$) vertex has been chosen as an auxiliary vertex. Additional removal candidates are available. The steps 1-3 are repeated until no more auxiliary vertices can be created.

During classification auxiliary vertices are treated as any other vertex and can be classified as to remain in the mesh or to be removed. If the vertex is selected to remain in the mesh, it is a valid replacement position for neighbouring removal candidates and no longer an auxiliary vertex. If it is selected for removal, it remains an auxiliary vertex until it has a neighbour that is selected to remain in the mesh and it becomes a removal candidate.

## 6 VERTEX REMOVAL

Compared to [Oda15a] the removal step remains mostly unchanged. All removal candidates are processed in parallel. For each candidate the possible replacement positions are determined, the per-vertex boundaries (which block any half edge collapse that may cause a mesh foldover or topological inconsistency) computed and one half edge collapse is executed.

We do, however, change the definition of possible replacement positions. Our original approach only used vertices that were selected to remain in the mesh. This was chosen to make sure that each vertex was moved to its final position and allow for an efficient implementation. With the introduction of auxiliary vertices a vertex can be moved to the position of a neighbour that has to be removed in a later iteration.

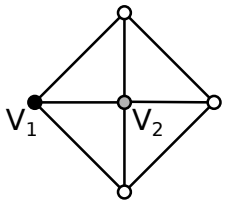


Figure 6: Possible replacement positions

We therefore change possible replacement positions for a vertex to include every neighbour that is not currently a removal candidate. This definition differs from that of the removal candidates, since it also includes vertices, that are selected for removal but are neither auxiliary vertices nor removal candidates. Fig. 6 shows an example. $V_1$ is to remain in the mesh, $V_2$ is a removal candidate. The remaining vertices are selected for removal but currently not removal candidates. In our previous approach, only $V_1$ is considered a possible replacement position, while the improved algorithm can move $V_2$ to any of its neighbouring vertices. The definition of a removal candidate is chosen to guarantee that each vertex that is processed has at least one possible replacement position and to avoid vertices that cannot be removed. At the same time we want to allow the most amount of freedom when choosing a replacement position.

In [Oda15a] we discuss the problem that our per-vertex borders can block combinations of half edge collapses when applied to neighbouring vertices. This can lead to multiple neighbouring vertices blocking each others' removal, causing a deadlock and delaying completion of the simplification process. Allowing any neighbours that are not removal candidates to be chosen as a replacement position has the potential to avoid blocked replacement positions, reduce deadlocks and improve the quality of the coarse mesh.

## 7 RESULTS

We devised an implementation of our improved algorithm using Nvidia CUDA and ran multiple tests on a Geforce GTX 670 GPU with 1 344 cores. Several models from the Stanford 3d Scanning Repository (Stanford Bunny, Armadillo, Dragon and Happy Buddha) were chosen for testing purposes.

Fig. 7 shows simplifications of the Stanford Bunny and Armadillo and compares them to the original meshes (from left to right: Stanford Bunny original, Stanford Bunny simplified, Armadillo original, Armadillo simplified). Fig. 8 shows the same comparison for Dragon and Happy Buddha. For these examples about 90% of the triangles of the original meshes have been removed. The models vary in terms of vertex and triangle count and were chosen to analyse how the improved algorithm scales with an increasing number of vertices.

Table 1 shows the number of vertices and triangles the models we used for testing contain.

Figure 7: Comparison: Original (left) and simplified (right) mesh for Stanford Bunny and Armadillo

Figure 8: Comparison: Original (left) and simplified (right) mesh for Dragon and Happy Buddha

| Model | 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 1: Number of vertices and triangles in the models used for testing

For testing purposes a simplification was computed for each model that removed a majority of the triangles of the original mesh. The most important factor to us is the overall runtime of the simplification process. As described earlier, the maximum simplification is determined by the number $N$ for the vertex error manipulation. We chose this number separately for each model to incorporate its vertex count.

| Model | Triangles rem. | N | Time (ms) |
|---|---|---|---|
| Stanford Bunny | 62 100 | 4 | 5.2 |
| Armadillo | 324 164 | 6 | 26.1 |
| Dragon | 821 161 | 7 | 52.6 |
| Happy Buddha | 1 027 314 | 7 | 66.5 |

Table 2: Triangles removed, number $N$ and processing time for the simplification in milliseconds for each model

Table 2 shows an overview of the results of the simplification process. For each model, the number of triangles removed, the number $N$ chosen and the processing time in milliseconds are listed. The Stanford Bunny - being the model with the fewest vertices and triangles - was simplified in only 5.2 ms with a triangle reduction of about 90%. The triangle count of the largest model in these tests (Happy Buddha) was reduced by about 94% with a runtime of less than 67 ms.

In addition to the overall processing time we measured the number of necessary iterations.

| Model | Iterations |
|---|---|
| Stanford Bunny | 7 |
| Armadillo | 15 |
| Dragon | 18 |
| Happy Buddha | 20 |

Table 3: Number of iterations necessary to complete the simplification for each model

Table 3 shows the necessary iterations for all models. The simplification process for the Stanford Bunny took 7 iterations while the coarse mesh for Happy Buddha was created in 20 iterations. In addition to the number of iterations, we measured the number of removal candidates in each iteration for the Stanford Bunny.

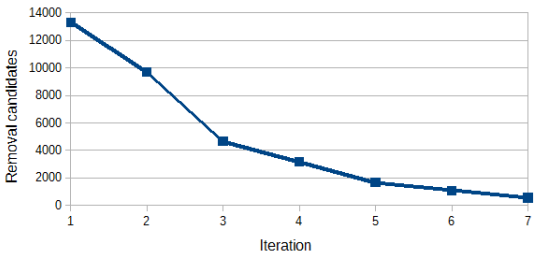


Figure 9: Removal candidates per iteration for the Stanford Bunny

Fig. 9 shows the number of removal candidates in each iteration for the simplification of the Stanford Bunny. This graph does not show how many vertices were actually removed in each iteration. Some vertices may not have a valid replacement position due to the per-vertex borders and cannot be removed. They remain in the mesh and are again removal candidates in later iterations. In the first iteration 13 321 removal candidates are available for processing. This number drops throughout the process with 1 089 and 555 removal can-

didates in the last two iterations. While the later iterations cannot fully utilize all available cores of the GPU (1 344), the majority of the iterations offers more removal candidates than cores.

In addition to the performance measurements we compared the results to those of our previous algorithm.

### 7.1 Comparison to previous work

In [Oda15b] we presented the results of our previous work and compared it to existing algorithms. In this section we will compare the results of the improved algorithm to those in [Oda15b]. Overall we expected to see the highest performance gain when simplifying the models Dragon and Happy Buddha due to the higher vertex count.

For the performance measurements in this paper we chose simplifications of the models that result in a similar number of vertices/triangles to the ones in [Oda15b]. This allows us to directly compare the results of the two algorithms and easily determine any performance gains. Since the algorithm in this paper is designed to improve parallelism and reduce the number of iterations, we expect to see improved run times and a reduced number of iterations compared to previous results. At first we compare the overall runtimes.

| Model | Impr. alg. | [Oda15b] |
|---|---|---|
| Stanford Bunny | 5.2 ms | 5.7 ms |
| Armadillo | 26.1 ms | 29.1 ms |
| Dragon | 52.6 ms | 80.1 ms |
| Happy Buddha | 66.5 ms | 96.1 ms |

Table 4: Comparison of runtimes between the improved algorithm and the results in [Oda15b]

Table 4 shows the runtime comparison for all four models. The reduction of runtime for the models Stanford Bunny and Armadillo is about 9%, while the improved algorithm can greatly reduce the processing time for models with a larger number of triangles (greater 30%). This comparison shows that the new algorithm can greatly reduce the runtime of the simplification.

The second measurement we showed earlier is the number of iterations and removal candidates in each iteration for the Stanford Bunny. Our new algorithm computes the coarse mesh of the Stanford Bunny in 7 iterations, while the results in [Oda15b] show that 12 iterations were necessary. The simplification of Happy Buddha took 33 iterations using our previous approach, while the improved algorithm finished after 20 iterations.

Fig. 10 shows the comparison of the number of removal candidates in each iteration between our original and improved algorithm for the Stanford Bunny. The 12 iterations for our original algorithm processed between 7 436 and 74 vertices with a very low number of

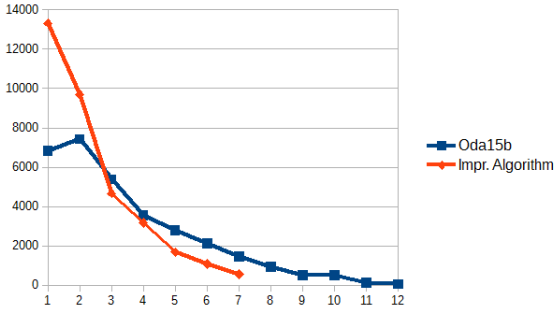


Figure 10: Comparison: Removal candidates per iteration for the Stanford Bunny

removal candidates in the last iteration (74 compared to 555 removal candidates for the improved algorithm). The modified vertex error manipulation and the introduction of auxiliary vertices improve parallelism and allow a better utilization of the hardware.

## 8 FUTURE WORK

Future work can improve the classification and reclassification steps. As the classification is currently based on a geometric error value and does not take the removal of neighbouring vertices into account, an unnecessary large number of vertices can be selected for removal. All these vertices need to be processed and may be reclassified. An improved error metric and better selection of vertices for removal can reduce the number of vertices selected for removal, reduce the number of vertices that need to be processed and therefore lead to an increase in performance. Greatly improving the classification may even render the reclassification step obsolete. This can lead to a better quality of the coarse mesh as well as shorter processing times.

## 9 CONCLUSION

The improvements made to our original algorithm have shown the potential to significantly reduce the runtime of the simplification by increasing the parallelism and reducing the number of necessary iterations. Especially when using models with a large number of vertices and triangles the improvements lead to a reduction in processing times of over 30%. The increased parallelism allows us to better utilize the parallel processing power of modern GPUs. The modified vertex error manipulation is better suited for models with irregular triangle sizes that proved to be disadvantageous to our previous approach as it could lead to a higher number of necessary iterations. Additionally the introduction of auxiliary vertices helps to reduce the number of vertices that cannot be processed in an iteration and improve parallelism.

On the other hand our algorithm still relies on error metrics designed for speed, fast update times and the isolated execution of the vertex removal operations. These

factors may lead to a decrease in overall quality of the coarse mesh and be limiting factors for the simplification.

## 10 REFERENCES

[Cla76a] Clark, J. H. Hierarchical geometric models for visible surface algorithms, Com. of ACM 19, No. 10, pp.547-554, 1976

[DeC07a] 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

[Hop93a] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., A., and Stuetzle, W. Mesh optimization, ACM SIGGRAPH Proceedings 1993, pp.19-26, 1993

[Hop96a] Hoppe, H. Progressive meshes, ACM SIGGRAPH 1996 Proceedings, pp.99-108, 1996

[Hu09a] Hu, L., Sander, P., V., and Hoppe, H. Parallel view-dependent refinement of progressive meshes, I3D 2009 Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games, pp.169-176, 2009

[Hu10a] 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

[Oda15a] Odaker, T., Kranzlmueller, D., and Volkert, J. View-dependent Simplification using Parallel Half Edge Collapses, Proceedings of WSCG 2015, pp.63-72, 2015

[Oda15b] Odaker, T., Kranzlmueller, D., and Volkert, J. GPU-Accelerated Real-Time Mesh Simplification Using Parallel Half Edge Collapses, Mathematical and Engineering Methods in Computer Science: 10th International Doctoral Workshop, MEMICS 2015, pp.107-118, 2016

[Ros92a] Rossignac, J., and Borrell, P. Multi-resolution 3D Approximations for Rendering Complex Scenes, Modeling of Computer Graphics: Methods and Applications, pp.455-465, 1992

[Sch92a] 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