The Visual Computer manuscript No.
(will be inserted by the editor)

Triangular mesh simplification on the GPU

Alexandros Papageorgiou · Nikos Platis

Abstract We present a simplification algorithm for triangular meshes, implemented on the GPU. The algorithm performs edge collapses on manifold triangular meshes driven by a quadric error metric. It uses data parallelism as provided by OpenCL and has no sequential segments in its main iterative structure in order to fully exploit the processing power of the GPU. Our implementation produces results faster than a corresponding sequential implementation and the resulting models are of comparable quality.

Keywords Mesh simplification · edge collapse · GPU · OpenCL

1 Introduction

Mesh simplification algorithms aim to create new, simpler triangular models based on highly complex ones, at a level of detail suitable for their use and for the available computational resources. Various methods have been proposed for this task, with techniques like successive vertex removals or edge collapses producing high quality results in acceptable time. Nevertheless, the ever-increasing size of triangular models being produced calls for even more efficient simplification algorithms.

Most existing simplification algorithms are sequential, suitable for execution on the CPU. However, the advent of modern programmable graphics processors (GPUs), which possess far greater processing power than that of current CPUs, offers the possibility of significantly accelerating the simplification process. Unfortunately, the architecture of GPUs is very different from that of the CPUs and, therefore, porting existing algorithms to the GPUs is, most often, not trivial. The main hurdles to overcome concern: the massively parallel architecture of the GPUs, which has vastly different requirements from the serial one of the CPUs; the limitations on memory management imposed by the architecture and programming model of the GPUs, as opposed to the freedom enjoyed on the CPU; and the need to minimize the data transfers between the main memory and the graphics card, since they slow down the procedure remarkably.

In this paper we present a simplification algorithm designed and implemented to fully exploit the capabilities of modern graphics processors, using the OpenCL framework. Our simplification algorithm is based on edge collapses and driven by the quadric error metric.

In contrast to existing algorithms, it is fully adapted to the data-parallel architecture of the GPU, containing no sequential segments in its main iterative structure; it utilizes only fixed-length data structures on the GPU; and it employs the least possible data transfers between the main memory and the graphics card.

Our results show that our algorithm is able to simplify large models considerably faster than the corresponding sequential implementation without significant loss in simplification quality.

Our implementation is currently optimized for manifold meshes. However, the data structures that we use are generic enough to accomodate even non-manifold meshes; therefore our implementation could easily be expanded to handle such meshes as well.

A. Papageorgiou and N. Platis
Department of Informatics and Telecommunications
University of the Peloponnese
End of Karaiskaki Street, GR-22 100 Tripolis, Greece
E-mail: {alexp|nplatis}@uop.gr

2 Background and related work

2.1 Quadrics based simplification

One of the most successful mesh simplification algorithms is the one described in [4]. This algorithm simplifies the mesh by successive edge collapses. For each prospective edge collapse, the algorithm finds the optimal position for the resulting vertex, the one that gives the minimum simplification error (deviation from the original mesh).

The potential error of an edge collapse is estimated using an error metric based on the planes of the faces around the edge. Conceptually, a set of planes is assigned to each vertex of the model; this set is initialized with the faces incident to the vertex in the original model. When an edge is collapsed into a single vertex, the set assigned to this new vertex is the union of the two sets associated with the endpoints of the collapsed edge. The error for a vertex is defined to be the sum of the squared distances of the vertex from all the planes in the corresponding set. This error can be expressed as a quadratic form (a *quadric*), which can be minimized efficiently to determine the optimal position for the resulting vertex. See Sections 3.2.2 and 3.2.4 below for more details.

The algorithm orders potential edge collapses in a priority queue, by increasing error, and performs them in succession. After each collapse, the mesh is locally altered in the vicinity of the edge, and therefore the error associated with collapsing neighboring edges needs to be re-computed.

2.2 Related work

In the literature, there are two characteristic works in which the GPU is used to simplify triangular models.

Hjelmervik and Léon [5] describe a simplification algorithm whose major part is executed on the CPU and only the part that performs the edge collapses is executed on the GPU; the continuous data transfers between the main memory and the GPU incur a high performance cost.

On the other hand, DeCoro and Tatarchuk [2] describe a global simplification algorithm that is executed on the GPU and partitions the 3D space with an octree structure. The detail level of the resulting mesh depends on the partitioning of the 3D space rather than on the particular features of the original model.

2.3 Bitonic sorting network

A sorting network is an abstract mathematical model consisting of wires and comparators that is used to sort a sequence of elements. Each comparator connects two wires and if the values do not satisfy the comparator's sorting order, the values of the wires are swapped. In order to be considered a sorting network, a structure of wires and comparators must be able to sort any sequence given as input.

The *bitonic sorter* [6] is a sorting network that can be used for the creation of a parallel sorting algorithm. It consists of $O(n \log^2(n))$ comparators and sorting is done in $O(\log^2(n))$ steps, where n is the number of elements to sort. This particular type of sorting network uses the notion of the bitonic sequence, which is a sequence of numbers $x_0 \leq \cdots \leq x_k \geq \cdots \geq x_{n-1}$ for some $k, 0 \leq k < n$, or a circular shift of such a sequence.

The function of the bitonic sorter is the following: initially sequences of length 2 are sorted, the first ascending, the second descending, the third ascending, etc. On the next step, sequences of length 4 are sorted by merging the length 2 sequences, where half of them are sorted in ascending and the others in descending order. The execution continues in the same manner, each time doubling the size of the bitonic sequence used, until the last step where the complete sequence is sorted.

From this description, it should be obvious that the basic bitonic sorter works on arrays whose length is a power of 2; extensions that sort arrays of arbitrary length are available, see Section 3.2.5 below.

3 Simplification algorithm for the GPU

The design of our simplification algorithm was significantly influenced by the capabilities of OpenCL [7].

The architecture of modern GPUs, as it is available through OpenCL, can be specified as a shared-memory multiprocessor system that performs calculations through data-parallelism. Thus our algorithm is modelled as a series of parts that perform calculations on the GPU through OpenCL kernels, and a series of dependencies, implemented by OpenCL events, that determine the sequence of execution of the parts of the algorithm.

The way that parts and dependencies form the complete algorithm is shown in Figure 1. The arrows denote the flow of execution for the algorithm as well as the data dependencies between the parts. The meaning of these dependencies is that a part has to create or process data before the next part can be executed.

All parts of the algorithm are executed on the GPU using data parallelism as provided by OpenCL, except

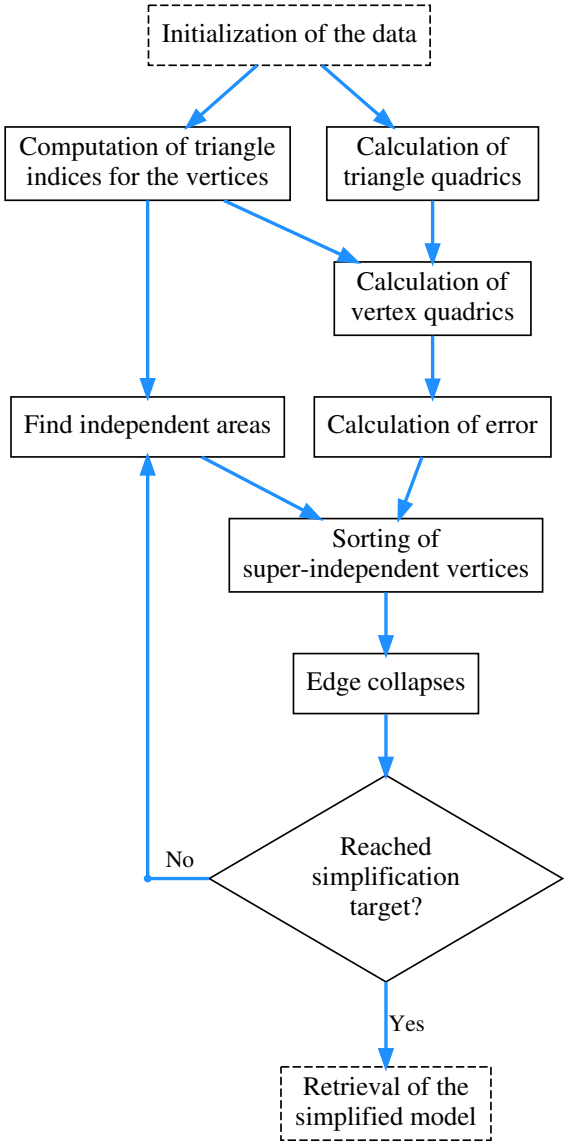


Fig. 1 The structure of the simplification algorithm. All parts are executed on the GPU except for those denoted by dashed rectangles.

for some initialization of the data structures and the retrieval of the resulting simplified model.

3.1 Data structures

The input data for the algorithm are the model to be simplified and the simplification target as the number of vertices that the final model will have.

The model is passed in as two arrays, one with the vertices (each given simply as three floating point coordinates) and one with the triangles (each specified by three integers, indices to the vertex array).

For our algorithm we create some auxiliary arrays in which, for each vertex of the model, we keep information about the quadrics, the error and whether the vertex has already been used in the current pass of the simplification algorithm.

Furthermore we need to keep, for each vertex, indices to the triangles incident on it. This structure is composed of two arrays, a *headers* array and a *data* array. For each vertex, the headers array contains a structure with fields {*position, size, continuesTo*}. The fields {*position, size*} describe the area in the data array where the indices of the incident triangles reside. The field {*continuesTo*} signifies whether we should continue to another area of the arrays and where this area is (Figure 2).

Ultimately, we create a series of unrolled linked lists, one for each vertex. This structure is necessary since OpenCL does not support dynamic memory management, because the number of triangles that each vertex is incident on is variable and changes during the course of the algorithm.

3.2 Parts of the algorithm

Apart from the initialization of the data structures and the retrieval of the model that have serial sections, the parts of the algorithm are executed using data parallelism. The important parts of the algorithm are described below.

3.2.1 Computation of incident triangles for each vertex

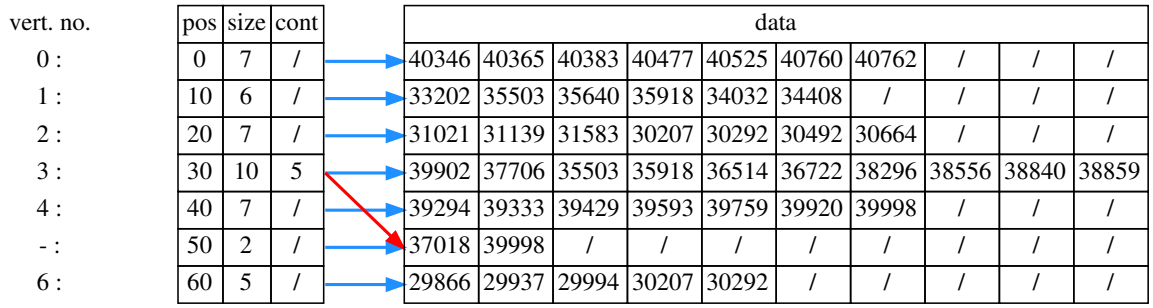
To find the indices of the triangles incident on each vertex, we work in parallel on the triangles. For every vertex of a triangle we increase, with an atomic operation, its counter of triangle indices and register the triangle index in the respective position in the data array of the triangle indices structure. As soon as we have processed all the triangles, the indices structure will represent the model completely.

3.2.2 Computation of quadrics

Quadrics are computed as defined in [4]. Specifically, consider the standard equation of a plane, $\mathbf{n}^T\mathbf{p}+d=0$ where $\mathbf{p}$ is a point on the plane, $\mathbf{n}$ is the unit normal of the plane and d is a scalar constant. Then, the squared distance of a vertex $\mathbf{v}$ (here, the vertex resulting from an edge collapse) from the plane is given by the formula

$$\begin{aligned} D^2(\mathbf{v}) &= (\mathbf{n}^T\mathbf{v}+d)^2 = \mathbf{v}^T(\mathbf{n}\mathbf{n}^T)\mathbf{v} + 2d\mathbf{n}^T\mathbf{v} + d^2 \\ &= \mathbf{v}^T\mathbf{A}\mathbf{v} + 2\mathbf{b}^T\mathbf{v} + c, \end{aligned} \tag{1}$$

where $\mathbf{A}=\mathbf{n}\mathbf{n}^T$ is 3×3 matrix, $\mathbf{b}=d\mathbf{n}^T$ is a 3-vector and $c=d^2$ is a scalar. The quadric Q of the plane is defined as the triplet $Q=(\mathbf{A},\mathbf{b},c)$ and then the distance



vert. no.	pos	size	cont	data									
0 :	0	7	/	40346	40365	40383	40477	40525	40760	40762	/	/	/
1 :	10	6	/	33202	35503	35640	35918	34032	34408	/	/	/	/
2 :	20	7	/	31021	31139	31583	30207	30292	30492	30664	/	/	/
3 :	30	10	5	39902	37706	35503	35918	36514	36722	38296	38556	38840	38859
4 :	40	7	/	39294	39333	39429	39593	39759	39920	39998	/	/	/
- :	50	2	/	37018	39998	/	/	/	/	/	/	/	/
6 :	60	5	/	29866	29937	29994	30207	30292	/	/	/	/	/

Fig. 2 The data structure used to hold the indices of triangles incident on each vertex. For each vertex, a *headers* array holds index information necessary to access the (one-dimensional, depicted here as two-dimensional only for convenience) *data* array. For example, vertex 1 has 6 incident triangles, whose indices are listed starting from position 10 of the *data* array; vertex 3 has 12 incident triangles, starting at position 30 and continuing, as indicated by row 5 of the *headers* array, to position 50 (vertex 5 was deleted as part of an earlier edge collapse). Notice that for each vertex a minimum number of positions for incident triangles is reserved (in the figure, 10 positions) in order to minimize the need to use the {*continuesTo*} field and optimize memory look-ups.

of the vertex **v** from the plane can be expressed using the previous formula. It is interesting that the sum of squared distances of the vertex from a *set* of planes can be expressed by an aggregate quadric created by simply adding (component-wise) the quadrics of these planes. Accordingly, for the purpose of this computation, the set of planes can be compactly represented by its corresponding aggregate quadric.

Using these formulas, in our implementation we first compute the quadric of each triangle of the model (as the quadric of its supporting plane), and store these in a temporary array. Then, we compute a quadric representing the set of triangles incident on each vertex of the model, by adding the quadrics of the triangles incident on it, and store these alongside the corresponding vertices; for this computation we refer to the data structure holding the indices of incident triangles described in Section 3.1. Both operations are executed with data parallelism on the respective arrays.

3.2.3 Identification of independent areas

As mentioned in Section 2.1 above, the standard simplification algorithm that utilizes edge collapses performs them in a sequence; after each collapse the model is locally modified around the collapsed edge and thus the error assigned to the neighboring edges must be re-computed.

This algorithmic structure cannot be parallelized efficiently. To perform edge collapses in parallel we find areas of the model that are *independent*, in the sense that an edge collapse may be performed in one area without affecting the data that are used for the simplification in another area. Based on the connectivity of the model, such an area is identified by one vertex (that is the central vertex of the area) and its surrounding vertices. For the areas to be independent, it is required that their central vertices be *super-independent* [3], meaning that their distance (on a graph representing the mesh, having the model's vertices as nodes and the model's faces as edges) must be at least three.

We compute the super-independent vertices with a parallel adaptation of the greedy serial algorithm that finds a maximal independent set on the graph representing the mesh. We logically split the vertex array in as many parts as the available processors and for every part we execute the serial algorithm to find super-independent vertices: if the current vertex or one of its surrounding vertices is marked as being used, we ignore it and proceed to the next one; if not, the current vertex is considered super-independent and it and its surrounding vertices are marked as being used. Vertices are marked as being used in a single auxiliary array which is shared between all the execution threads. When this part is completed we check again, using parallelism on the elements of the auxiliary array, all the vertices that have been marked as super-independent for having distance at least three to each other. This final check is performed in order to avoid the inclusion of a vertex in the result due to race conditions between the execution threads. If a vertex also passes this final check, it is added to the array of the super-independent vertices.

It should be noted that the aforementioned possibility of race conditions between the threads that determine independent areas, makes this part of the algorithm non-deterministic: different results may be obtained even when running it on the same system configuration. However, the effect of this behavior to the quality of the final simplified model is minimal, since, in subsequent steps, the best candidate edge is selected in each of the independent areas (see Section 3.2.4 below) and these are sorted (see Section 3.2.5) so that collapses

are performed first in areas where the least error will be introduced.

In terms of complexity, this step is dominated by the greedy serial algorithm that finds a minimal independent set of vertices, which is $O(n)$ where n is the number of vertices examined (by each processor).

3.2.4 Edge collapse error

The edge that will be collapsed in each of the independent areas is one of those sharing the central vertex; specifically, it is the one that will incur the smallest error on the model, provided that the manifold property of the model surface is maintained.

For each candidate edge collapse, the position of the resulting vertex and the corresponding error are calculated by using the quadrics. In order to determine the position of the vertex that minimizes its total squared distance to the corresponding set of planes it suffices to zero the derivative of formula (1), which gives $\mathbf{v}^* = -\mathbf{A}^{-1}\mathbf{b}$ for the optimal position of the vertex and $\epsilon = \mathbf{b}^T\mathbf{v}^* + c$ for the error.

Having selected the edge collapse for an independent area, we assign it to the representative super-independent vertex.

3.2.5 Vertex sorting

To sort the super-independent vertices we use a variation of the bitonic sorter (see Section 2.3) whose parallel implementation supports arrays of size different than a power of 2 while maintaining its complexity to $O(log^2(n))$ [8]. We add virtual padding at the end of the array up to the next power of 2; we treat these padding elements as max-values so that when a comparison-exchange is underway, no values from the array are moved past the actual end of the array.

The final sorting order is not affected by the sorting order of the previous steps (i.e. whether the first sequence is in ascending or descending order), as long as the sequences are bitonic. The bitonic sorting algorithm is modified so that at each step, the elements of the last sequence that lies on the actual array are sorted in ascending order, and with the virtual padding we avoid moving array elements past the actual end of the array. The sorting order of the other sequences is selected so that the sequences for the next step are bitonic.

Although there is a small impact on the running time due to the virtual padding, by using it we can perform parallel in-place sorting of arbitrarily sized arrays.

3.2.6 Edge collapses

This is the part that finally uses all the data that were prepared in the previous steps of the algorithm and performs edge collapses in parallel on the independent areas. We mention early on that we do not perform *all* available edge collapses in parallel, since in this way we would even simplify areas where a large simplification error would occur; see Sections 3.3 and 4.1 for a relevant discussion.

For each edge that will be collapsed we locate the triangles (at most two, based on the manifold property) that will be deleted, by using the incident triangles indices data structure.

Then we modify the connectivity of the triangles in the affected area. We scan the indices of the triangles that use the central vertex and change the triangles so that the central vertex is replaced by the other vertex on the collapsing edge. After that, the central vertex will not be referenced by any triangle and thus, in the end, we will mark it as deleted. After we modify the connectivity of the triangles, we update the indices structure, appending to the list of the second vertex the list of indices of the central vertex (using the *continuesTo* field of the respective header, and not moving any elements at this point).

Further on, we modify the connectivity of the vertices. We must remove the triangles that will be deleted from the incidence lists of their three vertices; we scan these lists and erase the appropriate indices.

While we erase the indices, we also compact the lists so that they have no gaps; this optimizes the time required to traverse the lists of incident triangles in subsequent iterations of the algorithm, since less memory accesses will be needed (otherwise the positions of deleted triangles would be accessed only to be ignored), and also cache coherence will be able to automatically accelerate the procedure.

The next step is to reposition the second vertex to its final position and update appropriately the error array, using the value computed by the quadrics.

Now the edge collapse has been essentially completed and the final remaining task for this part of the algorithm is to mark the triangles and the vertex that we no longer need as deleted.

Overall, the steps performed by this part of the algorithm in the order they are executed are the following (Figure 3):

- Find the best edge.
- Find the triangles to be deleted.
- Calculate the optimal position for the resulting vertex.
- Modify the connectivity of the triangles.

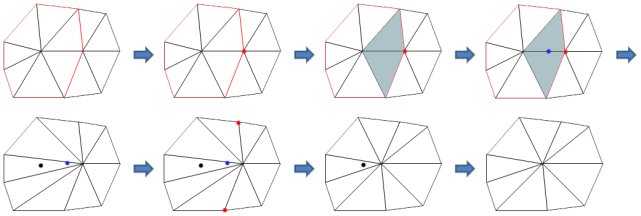


Fig. 3 Edge collapse steps

- Append the central vertex's list to the list of the 2nd vertex.
- Remove the indices of the triangles to be deleted and compact the lists.
- Reposition the 2nd vertex.
- Update the error array.
- Remove the triangles and the central vertex.

3.3 Parameters of the algorithm

There are some parameters of the algorithm which affect the execution time and the quality of the resulting model.

The first one is the percentage of the independent regions to be used in each iteration of the algorithm to perform simplifications. The serial algorithms perform only one edge collapse (the best one) each time, but we need to perform multiple collapses in parallel. In each iteration we scan the whole model to find the independent areas and sort them based on their error, as described in the previous sections. If we use all the independent areas that we find, we are going to perform collapses that will incur significant error on the model. On the other hand, if we use too few of them, we will get a better result but we will need more iterations of the algorithm to reach the goal of the simplification, with the corresponding cost in total execution time. We present the effect of this parameter to the simplification result in Section 4.1.

The second parameter is the number of execution threads for the part that finds the super-independent vertices. We want to have enough execution threads so as to maximize the utilization of the available processors; but we do not want more than these, because we increase the dependencies between regions of the model, which may reduce the final number of selected areas. In our implementation we handle this parameter automatically; the number of threads is calculated based on the number of execution units so that there are no more than 64 threads for each one, but also so that each thread processes at least 100 vertices.

The last parameter concerns the size of the array for the super-independent vertices (which is allocated in the first pass of the algorithm and reused in the rest). Ideally the exact size that we will need should be calculated, but as the connectivity of the model is generally irregular, the only thing we can do is use an approximate value. The value used in our implementation was set to 5% of the initial number of vertices of the model.

4 Results

We measured the performance of our algorithm by simplifying several models in various configurations of the algorithm. All the measurements were performed on a computer with a quad-core Intel Core i7 920 CPU at 2.67GHz, 6GB of DDR3 RAM and an Nvidia GeForce GTX275 GPU with 896MB of GDDR3 RAM.

In the subsections that follow we present

- the effect of altering the percentage of independent regions used in each iteration of the algorithm,
- performance results, in comparison with a serial implementation of the original surface simplification algorithm that uses the quadric error metric on the CPU, and
- some results for the (parallel) execution of our algorithm on the CPU.

We tested our algorithm on several well known models of varying complexity, ranging from the *cow* model with 5,804 triangles and a rather smooth surface up to a *gargoyle* model with 1,000,000 triangles and a finely detailed surface. The behavior and the performance of our algorithm were consistent along all these models, therefore we only present the most representative and interesting cases below.

We note here that we were unable to perform comparisons with the other GPU simplification methods mentioned in Section 2.2 because their implementations were not available.

4.1 Percentage of independent regions

Table 1 explores the effect of using different percentages of independent regions in each iteration of the algorithm. The data refer to the simplification of the *horse* model, consisting of 45,485 vertices and 96,966 triangles, down to 1,500 vertices and 2,996 triangles (approximately 3% of the original). As already mentioned, by simplifying more independent regions simultaneously in each step, we need fewer steps to reach the simplification target and therefore less time; unfortunately this results in models of lower quality, as illustrated in Figure 4. It can be observed that when simplifying a high percentage of independent regions simultaneously, the

% of indep. reg.	Iterations	Time
100	37	983
85	43	1155
50	72	1950
25	144	3635
10	357	8174
5	731	15257

Table 1 Effect of different percentage of independent regions used in each step of the algorithm.

edge collapses tend to be more evenly distributed on the surface of the model, occuring even on areas of fine detail (such as the head or the legs); this happens because we force the algorithm to simplify many regions at once, essentially disregarding the ordering of potential edge collapses by increasing error. On the contrary, when few edge collapses must be performed in parallel, these are, in each step, the ones with the least error which naturally affect areas without fine details (such as the horse body).

The default value that we used in our implementation was set to 85%, which gave quite good quality results in short time. Our tests showed that this value could be lowered in order to increase the quality of the simplified models while still retaining significant speedups with respect to the serial simplification.

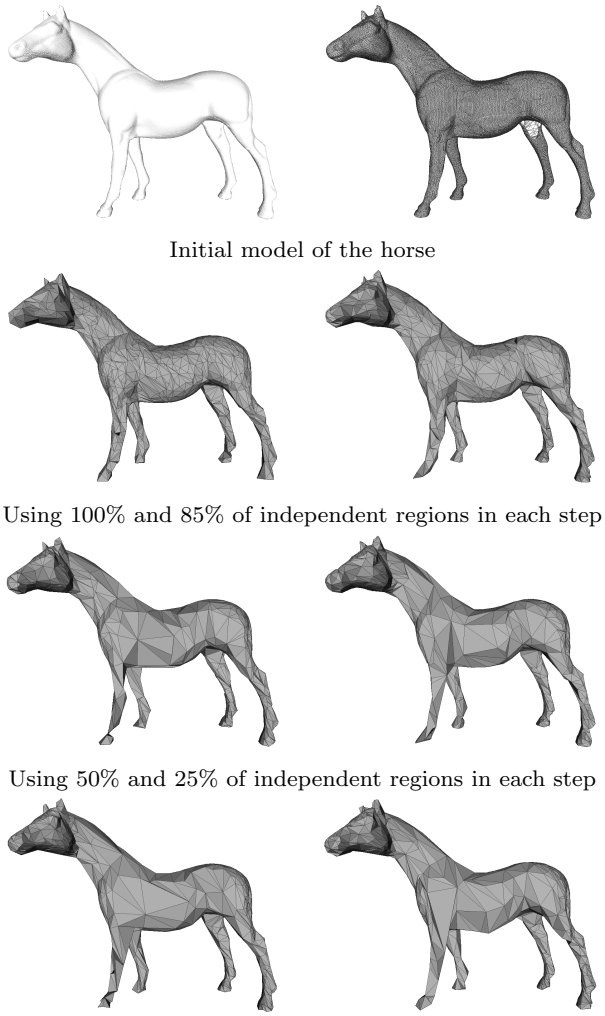


Fig. 4 Effect of different percentage of independent regions used in each step of the algorithm. Top row: model of the horse at full detail (45485 vertices); next three rows: model of the horse at 1500 vertices, using different percentages of independent regions.

4.2 Performance results, comparison with serial implementation

We compared the execution time as well as the quality of the simplified models of our algorithm with those of a corresponding serial simplification algorithm. The serial implementation that we used was the one from the *MeshLab* system [1] (version 1.3.0_64bit) and specifically the "Quadric Edge Collapse Decimation" filter from the "Remeshing simplification and reconstruction" category. The measurements of the quality characteristics of the final simplified models were derived with the use of the *PolyMeCo* software [9] (version 1.1).

In Table 2 and Figure 5 we show the behavior of our algorithm for the simplification of the *gargoyle* model, a model with a fairly high level of detail, composed of 500,002 vertices and 1,000,000 triangles. The initial cost of our algorithm is quite small and the total simplification time is almost linearly dependent on the degree of simplification, up to the point where the number of independent areas on the model is drastically reduced. After that point, the possibility of parallelization is reduced and as a result the required time for further simplification increases almost exponentially. Nevertheless, the running time of the algorithm is still less than that of the serial algorithm (whose execution time also increases almost linearly with the degree of simplification, albeit with a steeper slope than the parallel algorithm). Overall, the speedup of our parallel implementation was in the range 1.5–4.2 for this model.

Similar behavior was observed on smaller models, for example the horse model. The curves depicting the execution time vs. the simplification target had similar shapes as those of Figure 5. The speedup was somewhat lower, 1.1–3.5, which is to be expected since larger models benefit more from the paralellization of the algorithm.

Exemplar qualitative results are shown comparatively in Figure 6 for the simplification of the gargoyle model at 10% of its original vertices. The two algorithms produce models that are not visually very differ-

Target	Vertices	*Parallel simpl.*		*Serial simpl.*
		Iter.	Time	Time
90%	450001	4	858	3596
75%	375001	6	1450	4588
50%	250001	10	2510	7106
25%	125000	18	3900	9498
10%	50002	29	5101	11204
5%	25002	37	5694	11766
1%	5006	54	6662	11798
0.5%	2503	63	7160	11946
0.1%	507	81	8206	12015

Table 2 Simplification of the gargoyle model (500002 vertices).

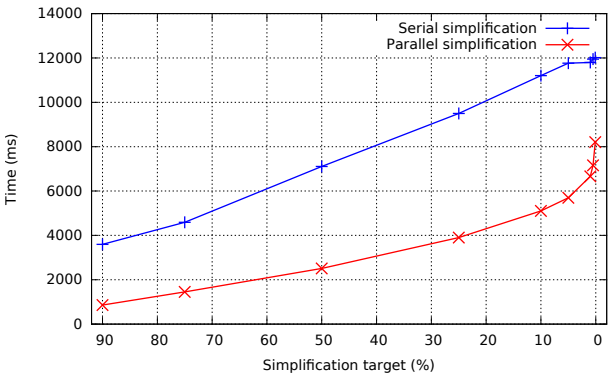


Fig. 5 Execution time for the simplification of the gargoyle model by the two implementations.

ent and both preserve the details of the original model adequately. As expected, the serial algorithm preserves the details better, since it always performs the best edge collapse at each step, whereas the parallel algorithm performs several collapses in each iteration, some of which are not the best choices.

The last two rows of Figure 6 present, over the surface of the model, the geometric deviation as well as the magnitude of the normal vector deviation of the simplified models from the original one. Again, the serial implementation exhibits expectedly better results, since it can avoid simplifying highly detailed areas of the surface. The comparison is quantified in Table 3, showing the mean values of the geometric deviation, the normal vector deviation and the smoothness of the model's surface (computed as the distance of each vertex from the centroid of its immediate neighbors). Although the serial simplification produces better results, the values are of the same magnitude, indicating results of comparable quality.

Qualitative results were similar for the horse model as well. Since this model is very much smoother than the gargoyle model, the geometric deviation and surface smoothness were three orders of magnitude smaller in all cases and the normal deviation was almost half in all cases; the relation between the respective characteristics of the parallel and the serial simplification was almost identical to the one of the gargoyle model.

Simplification	Geometric deviation	Normal vec. deviation	Surface smoothness
Initial model	-	-	0.08035
Parallel 25%	0.01524	0.10102	0.24674
Serial 25%	0.00673	0.07043	0.17030
Parallel 10%	0.03357	0.15738	0.40793
Serial 10%	0.01499	0.10919	0.28482
Parallel 5%	0.05961	0.20834	0.58382
Serial 5%	0.02499	0.14802	0.41982
Parallel 1%	0.22100	0.34857	1.34098
Serial 1%	0.07813	0.27320	1.08336

Table 3 Mean values of quality characteristics for various simplifications of the gargoyle model.

4.3 Parallel execution on the CPU

Since OpenCL is a cross-platform standard, programs using it should be able to run on other parallel architectures apart from GPUs, such as the multi-core current CPUs; they should even be able to run in heterogeneous CPU-GPU environments, making use of all available resources of a computer.

Unfortunately, in practice the situation is, at the time of writing this paper, far from ideal. Of the three OpenCL implementations currently available, the one from Nvidia targets only GPUs with an Nvidia processor, the one from AMD targets GPUs with an AMD processor as well as CPUs, and the one from Intel targets only Intel CPUs. Therefore in a system with an Nvidia GPU one can run OpenCL programs either on the GPU (with the Nvidia implementation) or on the CPU (with the AMD or the Intel implementation) but not on both.

Even more so, the OpenCL drivers seem still immature and not stable enough. While testing our implementation on the CPU we experienced random instabilities and crashes not whitnessed on the GPU – differing between the AMD and Intel implementations and even between versions of these implementations.

Figure 7 shows the results of preliminary testing of our algorithm on the CPU, using the Intel OpenCL implementation. It is interesting that running times on the CPU are comparable or even lower (at high simplification targets) than those on the GPU, even though the former has vastly fewer execution units than the latter. This can be attributed to the rather complex algorithmic structures that our implementation comprises, which run more efficiently on the CPU (eg. branching) than on the GPU.

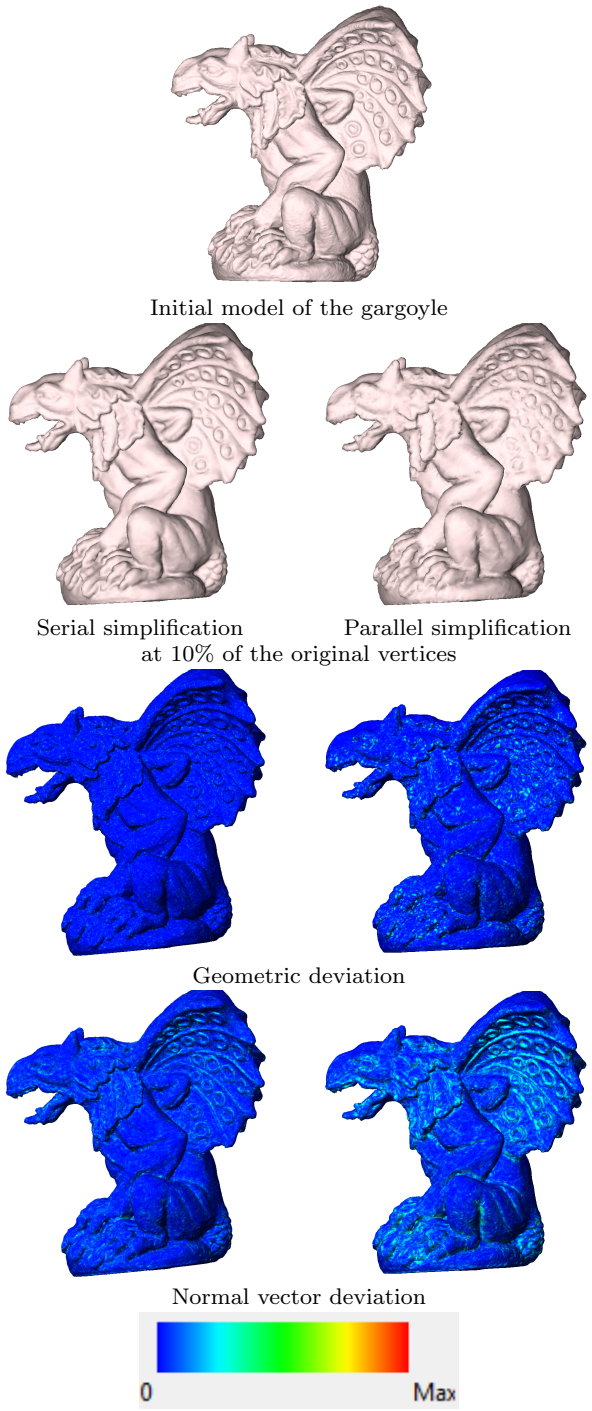


Fig. 6 Comparison of the results for the simplification of the gargoyle model

5 Conclusion

Taking into account the characteristics of the modern graphic processors and with the use of the OpenCL technology that allows us to exploit their capabilities, we designed and implemented a parallel simplification algorithm for manifold triangular meshes that is based on edge collapses and the quadric error metric. Our algorithm achieves significant speedup compared to a serial implementation of the simplification algorithm, while the final models are of comparable quality.

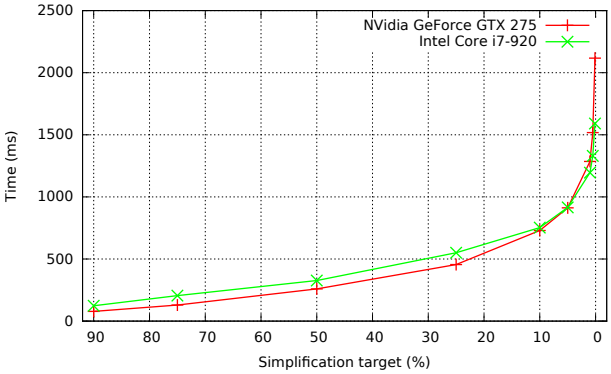


Fig. 7 Execution times for the simplification of the horse model on the GPU and on the CPU.

Our work could be extended towards applications in which multiresolution models are useful. A first step would be to record the simplification steps in order to create a progressive mesh structure for the model; this procedure would need to take into account synchronization issues between the threads that perform edge collapses in parallel. Further on, the progressive mesh structure could be exploited in applications that benefit from the selective refinement of the model; for such applications, the independent regions identified during our simplification algorithm could be used in order to streamline the required processing.

References

1. Cignoni, P., Callieri, M., Corsini, M., Dellepiane, M., Ganovelli, F., Ranzuglia, G.: Meshlab: an open-source mesh processing tool. In: Sixth Eurographics Italian Chapter Conference, pp. 129–136 (2008)
2. DeCoro, C., Tatarchuk, N.: Real-time mesh simplification using the GPU. In: Proceedings of the 2007 Symposium on Interactive 3D Graphics, SI3D, pp. 161–166. ACM (2007). URL http://doi.acm.org/10.1145/1230100.1230128
3. Franc, M., Skala, V.: Parallel triangular mesh decimation without sorting. In: Proceedings of International Conference SCCG, pp. 164–171 (2001)
4. Garland, M.: Quadric-based polygonal surface simplification. Ph.D. thesis, School of Computer Science, Carnegie Mellon University (1999)
5. Hjelmervik, J.M., Léon, J.C.: GPU-accelerated shape simplification for mechanical-based applications. In: Shape Modeling International, pp. 91–102. IEEE Computer Society (2007)
6. Knuth, D.: The art of computer programming, Vol. 3. Addison-Wesley, Reading, MA (1973)
7. Munshi, A.: The OpenCL specification version 1.0. Khronos OpenCL Working Group (2009)

8. Peters, H., Schulz-Hildebrandt, O., Luttenberger, N.: Fast in-place sorting with CUDA based on bitonic sort. In: Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part I, PPAM'09, pp. 403–410. Springer-Verlag (2010)
9. Silva, S., Madeira, J., Santos, B.S.: PolyMeCo–An integrated environment for polygonal mesh analysis and comparison. Computers & Graphics **33**(2), 181–191 (2009)