EUROGRAPHICS Workshop on Parallel Graphics and Visualization, Girona, Spain, pp.39-52, ISBN 84-8458-025-3, 2000.

Triangular Mesh Decimation in Parallel Environment

Martin Franc Václav Skala
{marty,skala}@kiv.zcu.cz

Department of Computer Science
University of West Bohemia, Plzeň, Czech Republic

Abstract. The aim of computer graphics is to visualise models of real-world objects. The visualisation of large and complex models is required more and more often frequently. This is followed by number of operations which must be done before own visualisation, whether it be an analysis of input data (e.g. searching for an iso-surface) or a model simplification. In spite of huge progress made in graphics hardware field in last years, we still need to increase a performance using optimal algorithms and programming techniques. One of the techniques that enhance the power is parallel computation. It can be seen that multiprocessor computers are more often available even for ordinary users. Together with Microsoft Windows expansion we have easy and comfortable tools for multiprocessor (multithread) programming as well.
We present an original efficient and stable algorithm for triangle mesh simplification in parallel environment. We use a method based on our original super independent set of vertices to avoid critical sections. Programs have been verified on MS Windows platform using standard Borland Delphi classes for multithread programming.

1. Introduction

Simplification of large complex models is a common task in visualisation. The simplification of models finds its use in virtual reality, whenever the object is in large distance from the observer or when details of the model are irrelevant or we just need rough view for fast manipulation (e.g. rotation). There are plenty of simplification techniques and approaches. We have chosen some well-known methods and combined them into an effective parallel algorithm. Our algorithm works with a *super* independent set of vertices for vertex elimination to avoid critical sections in program code, which normally decrease the speed of computation. As a programming tool we use standard Borland Delphi tools together with a *TThread* class, which encapsulates attributes and methods given by MS Windows for multithread programming.

In section 2 of this paper we describe the simplification in general and techniques we use in our algorithm. Section 3 introduces tools for multithread programming under MS Windows. This section is followed by section number 4, where we discus general problems of shared memory and explain *super* independent set term. In section 5 we present our algorithm and section 6 shows achieved results.

2. Mesh decimation

2.1. Popular techniques

Decimation methods are simplification algorithms that start with a polygonisation (typically a triangulation) and successively simplify it until the desired level of approximation is achieved. Most of decimation algorithms fall into one of the three categories, discussed below, according to their decimation technique.

Vertex decimation methods

One of the most used methods is vertex decimation, an iterative simplification algorithm originally proposed by Schroeder [1]. In each step of decimation process, a vertex is selected for removal. All the facets adjacent to that vertex are removed from the model and the resulting hole is triangulated. Each vertex is evaluated by its importance in the mesh. The vertex with low importance is eliminated.

In general, there are plenty of techniques simplifying triangular meshes by vertex elimination. The difference among them is the way of the vertex importance evaluation and kind of triangulation. Since the triangulation requires a projection of the local surface onto a plane, these algorithms are generally limited to manifold surfaces. Vertex decimation methods preserve the mesh topology as well as a subset of original vertices.

Edge decimation

Other subset of decimation techniques is pointed to the elimination of the whole edge. When an edge is contracted, a single vertex replaces its endpoint. Triangles, which degenerate to an edge, are removed. Hoppe [6] was the first who has used edge contraction as the fundamental mechanism accomplishing surface simplification. It is necessary to evaluate the importance of edges before the contraction. One of the best-known technique [3] uses *quadric error metrics* for the edge (vertex pairs) evaluation. The edges are contracted according to their importance – the less important edges first, similarly to the case of vertex removal. Unless the topology is explicitly preserved, edge contraction algorithms may implicitly alter the topology by closing holes in the surface.

Patch (triangle) decimation methods

Techniques, which eliminate either one triangle or any larger area, belong to the last group. These methods delete several adjacent triangles and triangulate their boundary. In case of one triangle, this one is deleted together with three edges and the neighbourhood is triangulated. The evaluation of the reduced elements requires more complex algorithms, in these methods.

2.2. Our approach – framework of our algorithm

Each of the above mentioned approaches have their advantages and disadvantages [2]. We have tried to extract the advantages of all approaches as will be presented in the following part.

We have started with vertex decimation methods and used the Schroeder's approach because of its simplicity and generality in meaning of vertex importance evaluation, and combine it with edge contraction. The methodology of vertex decimation is in fact closely related to the edge contraction approach (discussed above). Instead of the vertex elimination and arising hole triangulation, one of adjacent edge can be contracted as well. Removing a vertex by edge contraction is generally more robust than projection of neighbourhood onto a plane a triangulation. In this case, we do not need to worry about finding a plane onto which the neighbourhood can be projected without overlap.

Firstly, each vertex in a mesh is evaluated according to its importance. Then the one with the lowest importance is marked and the most suitable edge for the contraction is searched in its neighbourhood. As the most suitable edge for contraction we take the one, which goes out of the eliminated vertex, that does not cause the mesh to fold over itself, and is best preserving the original surface according to our criterion. As the criterion we use either the contraction of the shortest edge, or the criterion of minimal triangulated area. To prove the method independence of our algorithm, we have tested some more heuristic based on vertex decimation, besides Schroeder's method (see section 6.4.). These heuristics are described in [5] in detail and their comparison can be found in section 5 of this paper.

Since the contraction can potentially introduce undesirable inconsistencies or degeneracies into the mesh, we must apply some consistency checks to a proposed contraction. If one of the checks fails, we discard the contraction and use another edges if any still remains.

The main point of this work was to find an algorithm for simplifying a mesh of large and complex data sets in short time. The advantage of this approach is the simplicity of vertex importance evaluation as well as the triangulation by edge contraction.

3. Multithread programming

We have developed the algorithm for the Windows NT platform, using Borland Delphi. There are quite easy and efficient tools for multithread programming. In this section we shortly introduce the standard Delphi class *TThread* for multithread programming, which has been used in our implementation.

3.1. TThread class

A thread is an operating system object, where program code is run. For every application at least one (*primary*) thread is created. Each thread can create other new threads, during its run. These threads share the same address area and can perform either the same or different action. After the *primary* thread is finished (together with all its threads), the application is terminated and the process is erased from the system. Threads allow all program routines to run all at once. If there is one CPU only, threads alternate (so called pre-emptive multitasking), otherwise they run concurrently.

There are some issues and recommendations to be aware of when using threads. Keeping track of too many threads consumes CPU time; the recommended limit is 16 threads per process on single processor systems. With multiple threads updating the same resources, keep threads synchronised to avoid conflicts.

In Borland Delphi, there is a standard class named *TThread*, which encapsulates all attributes and methods for multithread programming that MS Windows allows. The *TThread* is an abstract class that enables to create separate threads for execution in an application. Each new instance of a subclass *TThread* object is a new thread of execution. Multiple instances of a *TThread* derived class make a Delphi application multithreaded. When an application is run, it is loaded into memory ready for execution. At this point it becomes a process containing one or more threads that contain the data, code and other system resources for the program. A thread executes one part of an application and is allocated CPU time by the operating system. All threads of a process share the same address space and can access the process's global variables.

Note that we have obtained very good speedup of parallel parts with the standard programming tools.

4. Super independent set

4.1. Independent set

A basic idea, which has been used in theoretical work [4] recently, is that decimation by deleting an independent set of vertices (no two of which are joined by an edge) can be run efficiently in parallel. The vertex removals are independent and they leave one hole per one deleted vertex, which can be triangulated independently. This decreases the program complexity and run time significantly. Since deletion and triangulation is related to the degree of vertices being removed (in the worst case with O (d^2) time complexity, where d is the vertex degree), Kirkpatrick [4] has advocated deleting low degree vertices ($d < 10$) and proved that this still allows large independent sets (>1/6 of all vertices). However, this approach ignores the preservation of the model shape. Therefore we use a technique [5] when we assign an importance value to each vertex, then select an independent set to delete by greedily choosing vertices of low importance relative to their neighbours.

It is natural to use a greedy strategy to construct an independent set from an assignment of importance values. It means to go through all the vertices in order of their importance and take a vertex if none of its neighbours have been taken. It means that only those vertices that do not share an edge with the each other can be in the independent set.

4.2. Shared memory problems

To store the information about vertices we use a data structure similar to a winging edge. With each vertex is kept the number (vertex degree) and indexes of triangles (neighbours), that share the vertex.

After several experiments we found, that the independent set described above is not fully adequate in our case. The criterion of no sharing edge between two independent vertices is not sufficient for effective parallelisation. Using such an independent set of vertices leads to situations when we have to use any synchronisation mechanism to protect the memory shared by threads. Two eliminated vertices from the independent set can share the same neighbours, so two threads change attributes of one neighbouring vertex at once, during the triangulation. To avoid such conflicts we have used mechanism called critical sections. Since the computation is quite fast, these critical sections rapidly decrease the effectivity of the algorithm. We found that the sequential algorithm is even faster than

the parallel one with critical sections on two processors machines (each processor works only on 45 % of its performance).

4.3. Independent set – without need of critical sections

In view of problems explained in previous paragraph, we have developed and use a *super* independent set, where every two triangles including two independent vertices can not share an edge, see Figure 1.

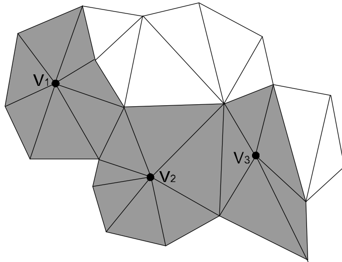


Figure 1: Vertices v_1, v_2, v_3 are independent to each other, but only the vertices v_1 and v_3 are super independent.

If we remove one vertex from the independent set, the removal change the properties of the vertex neighbours. That affects neighbourhood of other vertices in the set. Even these vertex neighbours are independent in this *super* independent set, so vertices are completely independent and the parallelisation can be done without critical sections in program code.

Due to the data structures used we can create the independent set in $O(d*n)$ time, where n is the number of vertices and d is the vertex degree ($d=6$ on an average).

5. Algorithm

5.1. Sequential algorithm

Having described a framework upon which our new algorithm is based, we can look at it more in detail (for the present, just the sequential version):

- The topology[1] of each vertex has to be evaluated. The importance of vertices is computed according to their topology, using Schroeder's method.
- All the vertices are sorted (in ascending order) of their importance.
- The vertices are reduced in order by the importance. For reduction we need to specify the *optimal* edge for contraction, which goes from the reduced vertex. Then the consistency is checked. If any of the consistency checks fails, other edge is to be taken; otherwise the vertex is removed (moved to the endpoint of the chosen edge and two neighbouring triangles are removed).

The whole process is repeated until the desire approximation is reached.

5.2. Algorithm analysis

Our original goal was to implement an efficient triangle mesh simplification algorithm in parallel environment in order to obtain faster response for very large data sets.

We can see that our sequential algorithm described in previous article, works in three steps.

[1] According to the vertex position in a mesh we recognize 5 types of vertices: simple, boundary, corner, interior edge and complex.

- *Vertex topology and importance evaluation.* This part can be done in parallel, because all vertices are obviously independent to the others.
- *Sorting*. When this paper is written, there is an implementation using sequential *Quick Sort* algorithm. The reason is that this work is still in progress. We know that there are well-known parallel sorting algorithms, so we have pointed our effort to the other parts of the algorithm foremost.
- *Decimation*. This part of the algorithm could run effectively in parallel, but we have to expect some important restrictions.

Using the independent set of vertices, we can split third step of the algorithm (the decimation) in to two parts – making an independent set and own decimation (now easy to run parallel). As we discussed the shared memory problem we use the *super* independent set rather then just independent set of vertices.
To decrease the system service overhead with managing threads, we split a set of computed vertices to number of parts equal to the number of free processors (used threads) roughly, and each thread computes one part. Due to very fast computation dynamic load balancing has not been considered.

On the other hand there is one disadvantage of independent sets of vertices in general. Imagine that we have vertices sorted by their importance. If several less important vertices lies close to each other (are neighbours), the first one is chosen to be in the independent set, but all the rest vertices are ignored. Thus instead of some unimportant vertices, any more important vertex is removed in one iteration of the algorithm.

5.3. Parallel version of the algorithm

Our new parallel algorithm can be described as:

1. Divide the set of vertices into *N* parts, where *N* is equal to the number of free processors.
   - Get the number of processors.
   - Divide the set of vertices into *N* parts of the same number of vertices.
2. Run *N* threads to evaluate vertex importance according to its topology. Each thread makes a computation on its own set of vertices.
   - Determine a vertex topology.
   - For *simple*, *boundary or interior edge* vertices, compute their importance. The importance for any other type of vertex is set to any high value ("infinite").
3. After all threads finish their task, sort (Quick Sort algorithm) all the vertices according to their importance in increasing order.
4. Find a *super* independent set of vertices. For each vertex do:
   - If the vertex is mark as unused in the independent set (initially all vertices are marked as unused), check its neighbours. If all the neighbours are unused, put the vertex into the *super* independent set and mark the vertex and all its neighbour vertices as used.
   - Used vertices and their neighbours are skipped.
5. Divide the *super* independent set of vertices into *N* parts.
6. Run *N* threads for decimation. Each thread makes the decimation on its own set of vertices.
   - For each eliminated vertex, find the optimal edge for contraction that includes it.
   - Test the consistency of the mesh if this edge is contracted (removed).
   - If the consistency test is OK, remove the vertex and triangulate the arising hole, otherwise find another short edge and go to the previous point.
7. Repeat steps 1– 6 until the required degree of the mesh reduction is reached.

6. Experimental results

In this section we present results of our experiments that compare an acceleration and efficiency obtained with different data sets, using different number of processors (threads).

We use 6 different data sets for the experiments, see Table 1.

Model name	No. of triangles	No. Of vertices
Horse	96,966	48,485
Bone	137,072	60,537
Bell	426,572	213,373
Hand	654,666	327,323
Dragon	871,414	437,645
Happyb	1,087,716	543,652

Table 1: Data sets used.

Table 2 shows time comparison achieved by reducing the models using 1 to 8 – processor computer (DELL Power Edge 8450 – 8xPentium III, cache 2MB, 550MHz, 2GB RAM, running on the Windows 2000).

Model name	Time [sec] obtained with different number of processors (threads) used							
	1	2	3	4	5	6	7	8
Horse	8.9	6.3	5.5	4.9	4.7	4.5	4.4	4.3
Bone	13.5	9.4	8.2	7.6	7.0	6.8	6.6	6.4
Bell	62.7	48.1	42.4	39.4	37.8	36.9	35.4	34.6
Hand	69.2	51.5	44.8	41.0	39.7	38.4	37.5	36.7
Dragon	93.6	69.5	61.5	57.6	54.3	52.6	51.3	50.6
Happyb	118.3	89.1	78.1	73.2	69.3	67.8	65.9	64.5

Table 2: Obtained time (in seconds) for 90% reduction on 1 to 8 processors active.

We investigated the acceleration and the efficiency for different size of data sets according to the number of processors used.

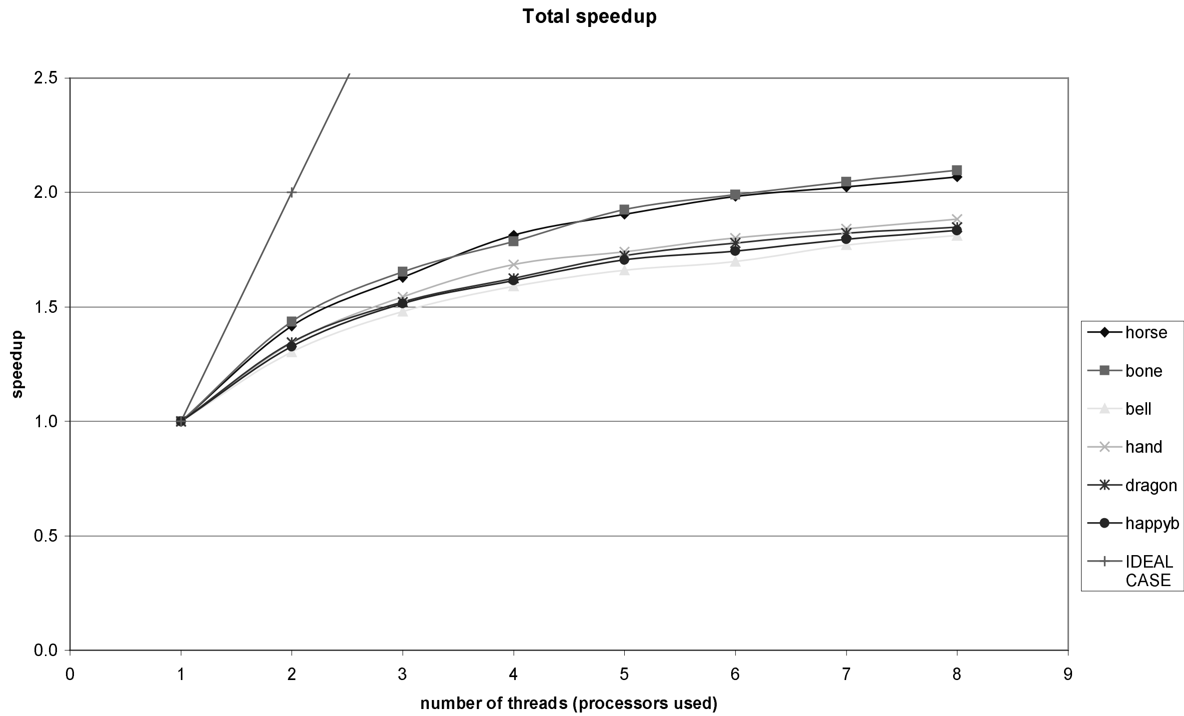
6.1. Speedup comparison

Figure 2a shows a graph of the speedup comparison. The speedup a is computed from total times (sequential and parallel parts of the algorithm together) using expression (1).

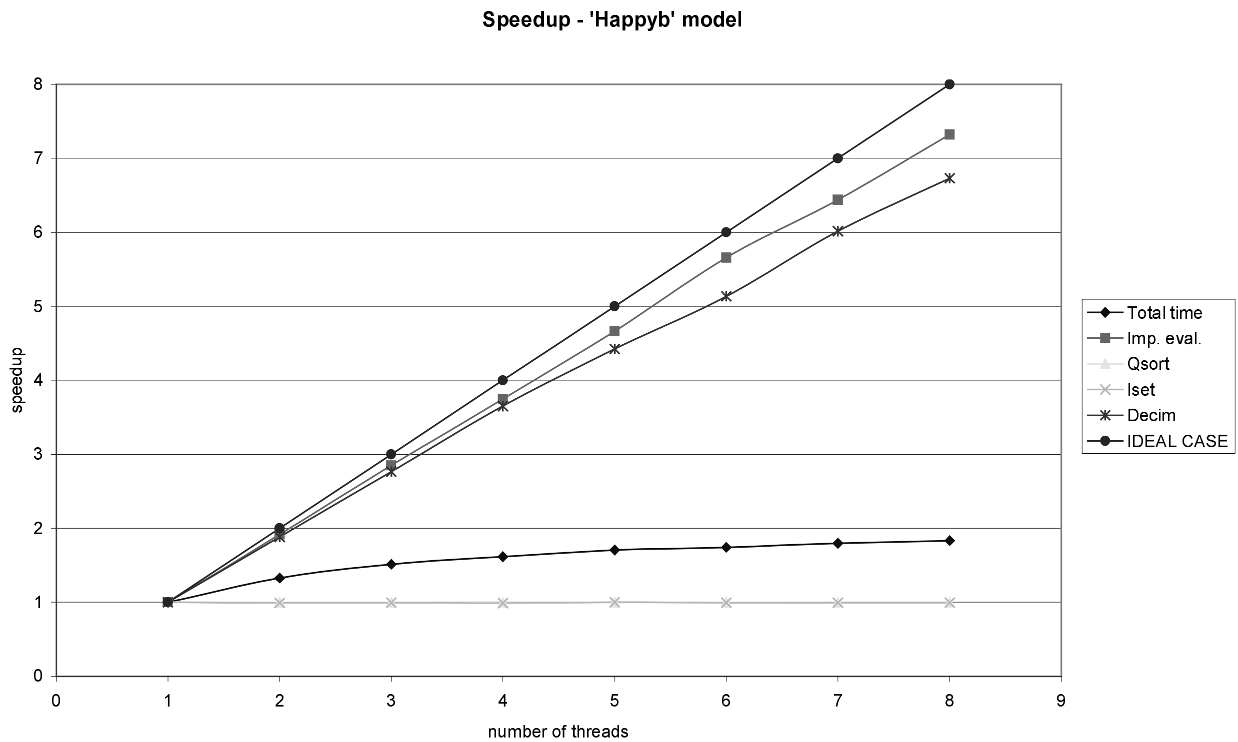
$$a = \frac{time_1}{time_N}, \qquad (1)$$

where N=1..8 is a number of processors used and $time_N$ is the time obtained if N processors (threads) are used.

Figure 2b shows the speedup of individual sequential and parallel part of the algorithm for the *Happyb* model. Total time means run time of the whole algorithm, Imp. eval. is time for importance evaluation, Qsort is sequential sorting time, Iset is time for selecting the super independent set and Decim is time of the decimation part.



(a)



(b)

Figure 2: *The acceleration of total computation (total time), parallel and sequential parts together; the acceleration is computed for several models of different amount of triangles (a) and comparison of acceleration of parallel and sequential code computed for the Happyb model (b).*

In Figure 3 you can see the actual runtimes (in percent) of the various algorithm steps for *Happyb* model. You can see that sequential sorting rapidly decreases the algorithm effectivity. Also the creation of *super* independent set seems to be limiting and the maximal speedup of this approach is approximately 15.

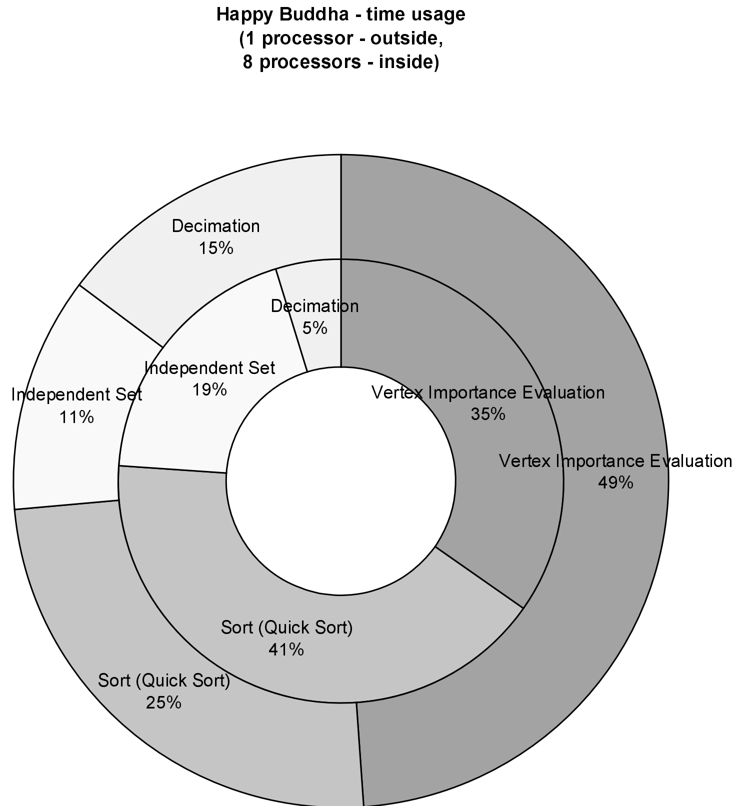


Figure 3: The time ratio of various parts of the algorithm for Happyb model with one and eight processors used.

6.2. Efficiency comparison

In Figure 4 we can see a comparison of the computational efficiency according to the size of data set and number of processors used. The efficiency e is defined as a speedup divided by the number of processors used:

$$e = \frac{time_1}{N * time_N}, \qquad (2)$$

where N=1..8 is a number of processors used and $time_N$ is the computational time if N processors are used.

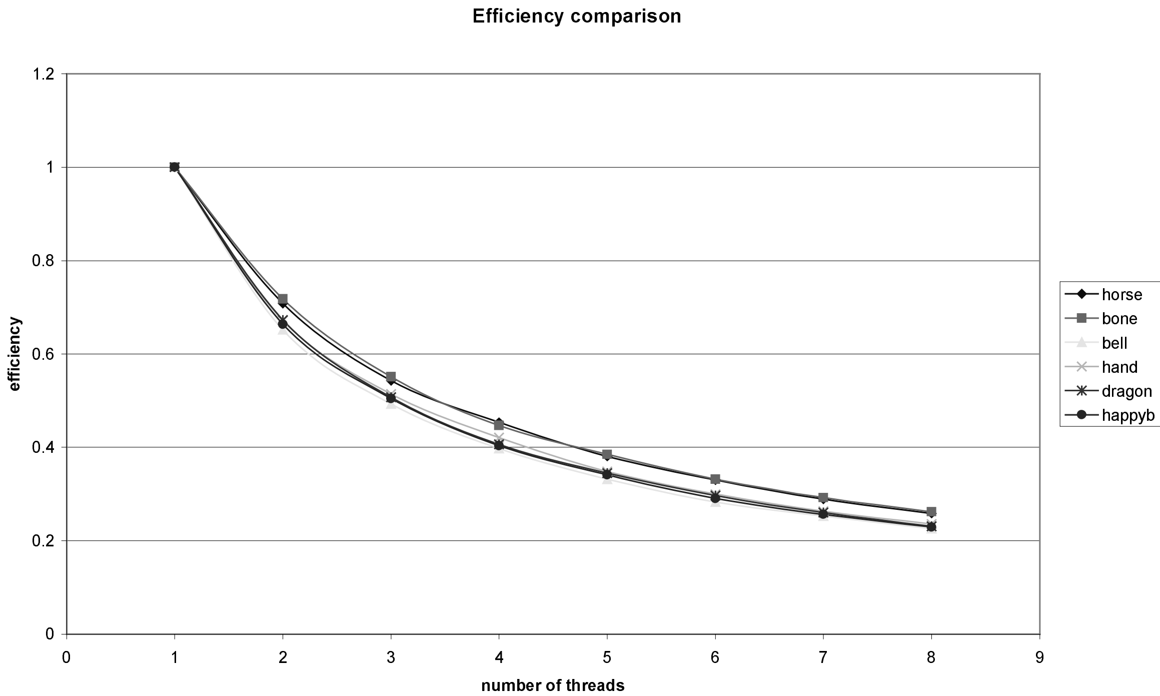


Figure 4: The efficiency of total computation; the efficiency is computed for several models of different amount of triangles.

6.3. Amdahl‘s law

The experiments proved that the method is stable according to the number of processors used and all the results meet the Amdahl’s law (3) perfectly.

$$a_{teor} = \frac{1}{(1-p)+\frac{p}{N}}, \qquad (3)$$

and therefore

$$p = \frac{N*(1-a_{teor})}{a_{teor}*(N-1)}, \qquad (4)$$

where p is potentially parallel code and N is the number of processors used.

The value of potentially parallel code is independent from the number of processors used, see Table 3 and and for the large model Happyb the value $p = 0.51$ was reached for the whole algorithm.

	Number of processors (threads) used							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
e	1	0.66	0.5	0.4	0.34	0.29	0.25	0.22
a	1	1.32	1.51	1.61	1.7	1.74	1.79	1.83
a_{teor}	1	1.32	1.51	1.61	1.7	1.74	1.79	1.83
p	X	0.49	0.51	0.50	0.51	0.51	0.51	0.52

Table 3: The experimental results and theoretical calculations according to Amdahl’s law; computed for the Happyb model.

Figure 5 shows the amount of potentially parallel code according to the number of triangles, for different number of processors used.

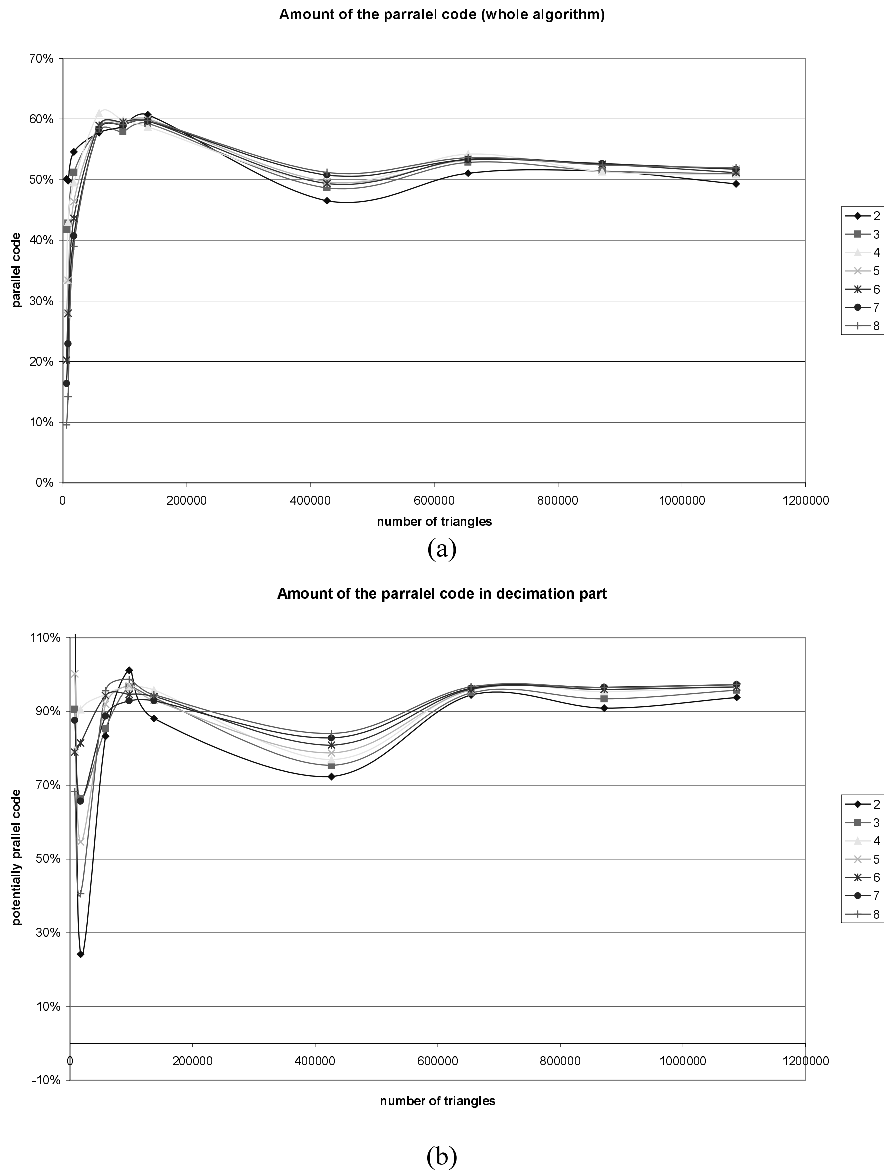


Figure 5: The amount of the parallel code for the whole algorithm (a) – the potentially parallel code according to Amdahl's law is approx. 51%, the amount of the parallel code for the decimation part (b).

6.4. Independence on decimation method

As we have already mentioned, we use some more vertex decimation heuristics to test the independence of our algorithm. In Figure 6, there is a graph of the speedup comparison according to partial parts of algorithm (computation) for 7 heuristic. On the left side, there is a graph of decimation according to the criterion of minimal area after the triangulation, on the right is a graph of decimation according to the shortest edge contraction criterion, see chapter 2.2. The heuristics are intimately described in [5] and the difference among them is the method of vertex importance evaluation. The following methods have been used: Schroeder's method, Average Normals, Height difference, Absolute Binary, Scape Order, Random (any of previous five) and Edge Switching.

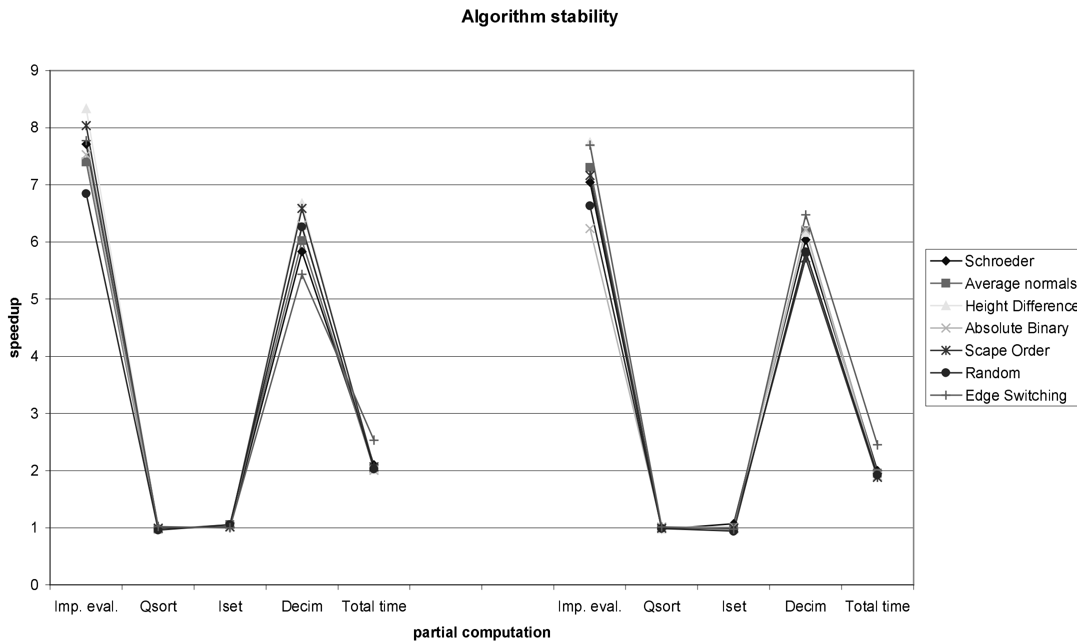


Figure 6: *The speedup of partial parts of the algorithm for several vertex importance evaluation heuristics. Criterion of minimal area triangulation on the left side, the shortest edge contraction criterion on the right side of the graph.*

It is obvious that the algorithm is independent of the heuristic used in meaning of the efficiency and the speedup.

6.5. Program output

Figure 7 and Figure 8 show examples of original and reduced models.

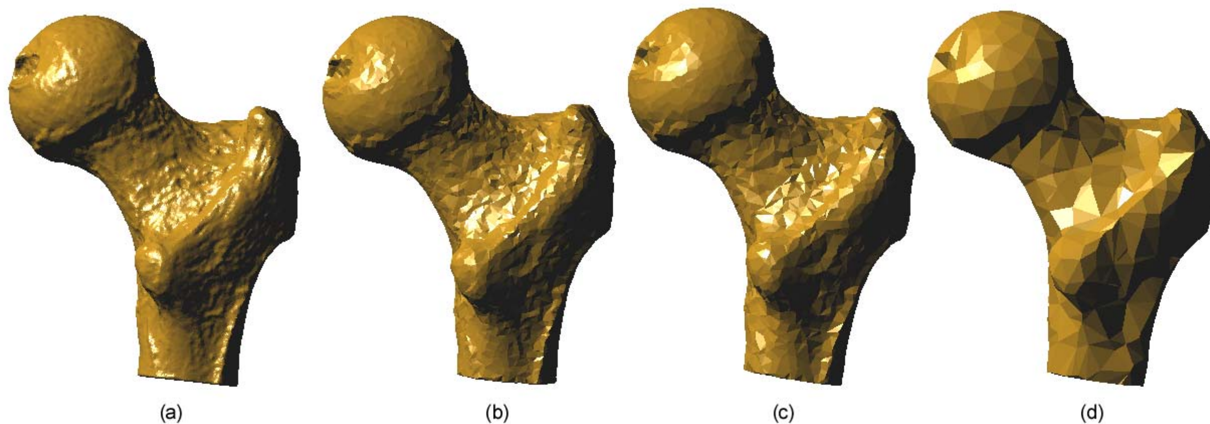


Figure 7: A bone model (courtesy Cyberware) at different resolutions; the original model with 137.072 triangles (a), reduced to13.706 triangles (b), 6.854 triangles (c), 1.248 triangles (d).

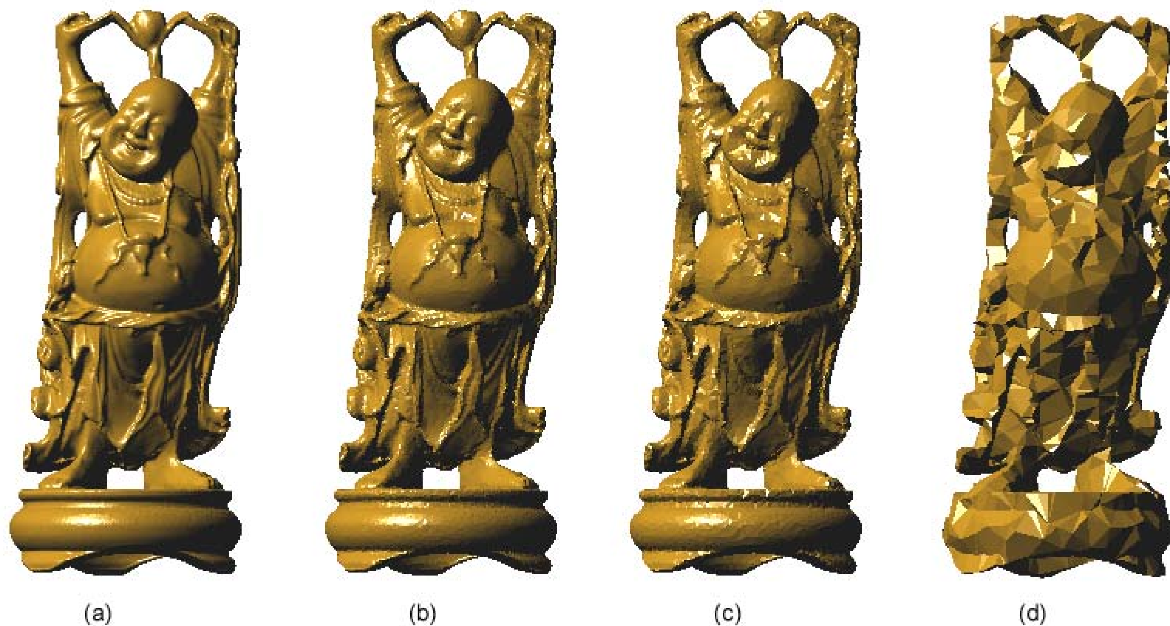


Figure 8: The Happyb model (courtesy GaTech) at different resolutions; the original model with 1.087.716 triangles (a), reduced to 105.588 triangles (b), 52.586 triangles (c), 10.974 triangles (d).

Conclusion

We have described the new original algorithm for triangular mesh simplification with its parallel modification.
The algorithm combines vertex decimation method with the edge contraction to simplify object models in a short time. We have used the *super independent* set of vertices to improve the parallelisation. It enables us to make fast parallel algorithm without use of any synchronisation technique such as critical sections, so the system overhead is minimal.

The obvious disadvantage of present implementation is the sequential sorting of vertices. Since this work is still in progress we have to take into consideration other possibilities too, because a parallel sort is only one of several solutions available. The question is what is going to be better; if the parallelisation of sorting algorithm or another approach of *super* independent set creating without need of sorted vertices.

Our experiment proved that we reached high effectivity of parallelisation $p = 0.87$ (if the overhead cost by the *TThread* class is included) or $p = 0.97$ (if the overhead cost by the *TThread* class is not included) for parallel parts. This also proved that high level constructions and object-oriented programming can be used at the application level with high efficiency guarantee.

The proposed method proved its stability according to the number of processors and the size of the data set used.

Acknowledgements

We would like to express our thanks to DELL Computer Czech Republic for enabling us to carry out all the experiments on their 8-processor computer type, DELL Power Edge 8450 – 8xPentium III, cache 2MB, 550MHz, 2GB RAM. We also benefited from the large model repository located at Georgia Institute of Technology; URL: http://www.cc.gatech.edu/projects/large_models.

References

[1] W. Schroeder, J. Zarge, W. Lorensen. *Decimation of Triangle Meshes.* In SIGGRAPH 92 Conference Proceedings, pages 65-70, July 1992

[2] M. Garland. Multiresolution Modeling: Survey & Future Opportunities. In the SIGGRAPH 97 course notes, 1997

[3] M. Garland, P. Heckbert. *Surface Simplification Using Quadric Error Metrics.* In SIGGRAPH 97 Conference Proceedings, 1997

[4] D. Kirkpatrick. *Optimal Search in Planar Subdivisions.* SIAM J. Comp., pages 12:28-35, 1993

[5] B. Junger, J. Snoeyink. *Selecting Independent Vertices for Terrain Simplification.* In WSCG 98 Proceedings, Pilsen University of West Bohemia, pages 157-164, February 1998

[6] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle. *Mesh optimization.* In SIGGRAPH 93 Conference Proceedings, pages 19-26, 1993

[7] H. P. Seidel, S. Campagna, L. Kobbelt, R. Schneider, J. Vorsatz. *Mesh Reduction and Interactive Multiresolution Modeling on Arbitrary Triangle Meshes.* In SCCG 99 Proceedings, Comenius University Bratislava, pages 34-44, 1999

This work was supported by The Ministry of Education of The Czech Republic – project VS 97155, Academy of Sciences of The Czech Republic – project A 2030801.