Efficiently Keeping an Optimal Stripification over a CLOD Mesh

Massimiliano B. Porcu
Dip.to Matematica e Informatica
Università di Cagliari
Via Ospedale, 72
I-09124, Cagliari, Italy
massi@dsf.unica.it

Nicola Sanna
Dip.to Matematica e Informatica
Università di Cagliari
Via Ospedale, 72
I-09124, Cagliari, Italy
nsanna@unica.it

Riccardo Scateni
Dip.to Matematica e Informatica
Università di Cagliari
Via Ospedale, 72
I-09124, Cagliari, Italy
riccardo@unica.it

ABSTRACT

In this paper we present an algorithm of simple implementation but very effective that guarantees to keep an optimal stripification (in term of frames per seconds) over a progressive mesh. The algorithm builds on-the-fly the stripification on a mesh at a selected level-of-details (LOD) using the stripifications built, during a pre-processing stage, at the lowest and highest LODs. To reach this goal the algorithm uses two different operations on the dual graph of the mesh: when the user changes the mesh resolution the mesh+strips local configuration is looked up in a table and, after a vertex split operation, the strips are rearranged accordingly, immediately after a sequence of special topological operation called "tunneling" with short tunnel length are started till the number of isolated triangles in the mesh get under 10% of the total number of strips. Moreover, when the user select a relevant LOD it can trigger a tunnelling with higher tunnel length to optimize the stripification. Using these operations we are able to keep the progressive mesh stripified in a time of the same order of magnitude of the time needed to change the resolution and, only if required, to perform a time-demanding optimization. Only the stripifications generated by explicit user requests are stored to serve as optimal starting points for further inspection. In this way we can always feed the graphics board with a triangle strip representation of the mesh at any LOD.

The results we present demonstrate that we can tightly couple each sequence of vertex splits used to increase the resolution of the progressive mesh with: a simple rearrangement of the strips followed by a very cheap stripification search with a predetermined strategy. A strong feature of the method is that the local rearrangement leads to an implementation that keeps almost constant the execution time. The results of the visualization benchmarks are very good: comparing the rendering of the stripified (using this strategy) and the non stripified meshes we can, on average, double the frames per seconds rate.

Keywords

Geometry compression – Stripification – Progressive meshes.

1. Introduction

Three different lines of research are active in trying to improve the management of large meshes: developing efficient algorithms for the compression of the meshes representation; improving the methods for the construction of a multiresolution data structure and easily select a mesh among all the ones stored in the structure; developing efficient ways to best render these meshes on current computer graphics hardware.

A good example of the first type of investigation is represented by the Edgebreaker algorithm and all its improvements [Tau98, Ros99, Paj00, Gan02]. This kind of algorithms allow to lossless encode meshes and collection of meshes (simplicial complexes) of any type using a reduced number (even less than two) of bits per vertex. The methods start from a seed triangle and grow on the free frontier (the boundary with other triangles not already encoded) till all the triangles are encoded.

The most popular method for building multiresolution structure is the progressive mesh method (PM) and all its improvements [Hop96, Hop97, Pop97, Hop98, Paj00]. Its great popularity derives also from the fact to be available as part of Microsoft's DirectX since the release 5.0.

The Journal of WSCG, Vol.13, ISSN 1213-6964
WSCG'2005, January 31-February 4, 2005
Plzen, Czech Republic.

Another way to try to compress the geometrical representation of a triangle mesh is the attempt to reduce the throughput between the CPU and the Graphics Processing Unit (GPU). The most common and diffused way to reach this goal is to rearrange the information describing each single triangle in the mesh in structured forms as the triangle strips and the triangle fans [Hae03].

A triangle strip is a set of connected triangles where a new vertex implicitly defines a new triangle. Triangle strips are used to accelerate the rendering of objects represented as triangle meshes, in a pre-processing stage the mesh is partitioned in a set of triangle strips (each of one can possibly be composed of one single triangle) and then each strip is passed to the GPU for rendering. The advantage of the strip representation over rendering each triangle separately, is that it makes it theoretically possible to reduce the number of vertices sent to the GPU from $3n$ (where n is the number of triangles in the mesh) to $n+2$ in the best case.

In this work we couple a selection and a stripification technique: the choice of a LOD over a PM with a method to accelerate its rendering.

The rest of this work is organized as follows: in section 2 we briefly go over the previous work done in geometry compression, focusing on selection methods and stripification algorithms; we then show, in section 3, the relations existing between the triangle mesh and its dual graph, introduce the tunnelling operator and explain how to build a stripification over the lowest resolution LOD of a progressive mesh; section 4 is dedicated to detail how the stripification is kept consistent while varying the LOD in the progressive mesh; in section 5 we show the results we obtained using our algorithm on a mesh we acquired from cultural heritage manufacts; and, finally, in section 0 we draw our conclusions and describe the future evolutions of this work.

2. Previous Work

Deering [Dee95] was the first to introduce the term *geometry compression*, to describe a set of techniques capable of reducing the space occupancy of a *generalized triangle mesh* statistically encoding XYZ positions, RGB colors and normals. These techniques operate mainly on the *geometry* of the mesh (i.e., the positions and the attributes of the vertices) relying on the triangle mesh structure to compress the information on the *topology* (i.e., how the vertices are connected to form the triangles). Since the goal of the work was to suggest a series of different operations the designer can perform to reduce the space occupancy of a triangle mesh, there wasn't any conclusion on the real possibility to move on the graphics board some of these stages.

Even if it is quite a rough classification, since there can be found many mixing approaches, we can divide the geometry compression methods in three main families:

- **Compression methods**. Allow to reduce the data needed to represent a mesh; they are well suited for transmitting and/or archiving the meshes;
- **Selection methods**. Allow to select the resolution that best fits the graphics hardware available for rendering; they are well suited for transmission with a preview effect; they are used to select a representation of the object described by the mesh, given a triangle/frame-rate budget;
- **Rendering accelerating methods** Allow to reduce the time spent in sending the information describing the mesh from the CPU to the GPU thus resulting in getting a higher frames per second rate without changing the number of triangles of the mesh (its geometry).

Compression Methods

After Deering [Dee95] several subsequent works [Tau98, Tou98], centered their attention on the problem of compressing the description of the topology arriving at a relevant result with the Edgebreaker method [Ros99, RsS99] which claims to reach less than two bits per triangle to encode a planar mesh homeomorphic to a disc.

All these techniques need a decompression stage that is not yet implemented in commercial graphics hardware, even using new programmable boards. This means that they are very efficient for transmission and archiving but cannot be used for feeding the GPU.

It is worth to mention that a useful consequence of the Edgebreaker encoding is the easy production of triangle strips while processing and decoding the compressed dataset [RsS99].

Selection Methods

Many authors presented solutions to generate multiresolution structures from an original mesh allowing the user to select a given LOD. We just limit ourselves to remind it's possible to divide the methods presenting a fixed number of LODs (usually less than ten) from the methods ranging on a continuous variation of LODs (CLODs).

Even if we don't want to rehearse all these works let's just briefly remind the main characteristics of the one we used in our implementation.

Progressive meshes (PM) represent the most popular type of continuous LOD meshes. They allow the users to easily encode a complex mesh using a single topological operation (Figure 1) called *edge collapse* (EC) and its complement, *vertex split* (VS). On the PM is possible to perform two different but equally important tasks: to select the representation best fit for the available hardware, and to progressively transmit the mesh.

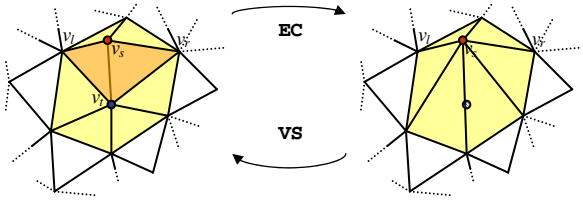


Figure 1: The two complementary operations performed on a progressive mesh

The original proposal [Hop96] has been refined during the last years: a hybrid compression and selection scheme trying to get the best of Edgebreaker compression and progressive meshes [Paj00], a further improvement, in term of bits per vertex [All01], and the extension of these techniques to arbitrary simplicial complexes [Gan02].

In our implementation we built a PM representation from the original following the longest edge rule: we collapse edges in order of decreasing length. We decided to use such a simple approach since the pre-processing in which we build the PM can be changed without affecting the rest of the process and, at this stage of development, we wanted to focus on the stripification scheme.

Stripification Techniques

The greatest advantage in using triangle strips consists of the availability of such a primitive in the OpenGL graphics library. Generating the stripification of a mesh means to be able to feed the GPU with the obtained structure without any further effort. It is actually to point out that OpenGL supports, without any vertex replication, only the sequential triangle strips. Generalized strips could thus bring to send more than once some vertices to the GPU.

Rearranging the order in which the vertices are stored is the typical way to face the problem of reducing the CPU-GPU throughput. The strips obtained are smaller than the original mesh when coming to the final rendering since, while the single triangle needs 3 vertices for its visualization to be sent to the GPU, the sequential triangle strip needs $n+2$ vertices to be sent to the GPU to render n triangles, and the generalized triangle strip $n+s+2$ where s is the number of swaps. The optimal single sequential strip encoding the whole mesh would reduce the number of vertices sent to the GPU by a factor of three.

Several papers illustrate geometrical and topological properties of a stripification [Ark96] and many variations of algorithms to partition a triangle mesh in strips [Eva96, Cho97, Ise01, Est02]. The most relevant work coupling multiresolution structure and stripification techniques is due to El-Sana et al. [ElS99, ElS00]. They introduce a data structure called Skip Strip that is used to generate the triangles strips. The method maintains a stripified progressive mesh during the refinement and coarsening process. This is an approach similar to the one we propose, but it relies on a much more complex data structure.

Working on the Dual Graph

Each triangle mesh can be alternatively represented by its *dual graph*. It is a graph in which each node is associated to a triangle of the original mesh and an edge represents an adjacency relation. One trivial property of such a graph is that each node has, at most, three incident arcs. In case the original mesh is homeomorphic to a sphere and has genus 1, each node has exactly three incident arcs.

It is quite common to use this representation to elaborate stripification algorithms: it allows to use a regular and compact data structure to represent the mesh and one can use all the results obtained from the graph theory. Unfortunately it has been proven [Ark96,. Gar76] that a problem equivalent to searching the optimal single strip (finding a Hamiltonian path on the dual graph) is an NP-complete problem, thus the stripification process should be based on local heuristics.

Two approaches for finding a stripification on the mesh's dual graph have been proposed: one is to compute a spanning tree on the dual graph, partition it into triangle strips, and then concatenate these strips into larger ones [Tou98], the second one is the so-called tunnelling algorithm and it is explained in detail in section 3.

3. Triangle Strip over the Progressive Mesh

Let us first briefly summarize the steps our method performs to keep the stripification at its best. They are:

1. Build the PM over the given mesh;
2. Build the stripification on the lowest LOD meshes using the procedure detailed in section 3;
3. Move over the PM performing either a vertex split operation (VS) on the mesh to increase the LOD or an edge collapse (EC) to decrease the LOD;
4. Rearrange the stripification using topological operations described in section 4;
5. Minimize the isolated triangles generated at the previous step using the tunnelling algorithm with short paths;
6. Build an optimal stripification using the tunnelling algorithm with longer paths, on demand and store it in the stripification data structure.

The step number *1* and *2* are pre-processing steps, we perform them on the mesh and then we can store the results in two supplementary data files, one for the PM and another one for the stripifications.

The Tunnelling Algorithm

The tunnelling algorithm, as initially proposed by Stewart [Ste01] and substantially improved by Porcu and Scateni [Por03], performs the stripification of the mesh using a simple topological operation on its dual graph.

To do so we need to think the graph edges as *colored* in two possible ways (see Figure 2):

- **solid edges** linking nodes associated to triangles in the same strip;
- **dashed edges** linking nodes associated to adjacent triangles not belonging to the same strip.

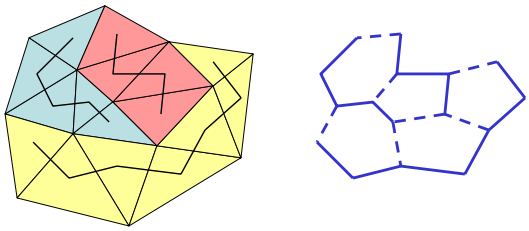


Figure 2: A stripified mesh (each color encodes a different strip) and its dual graph.

In every node there are, at most, two incident solid edges. The nodes with only one incident solid edge are *terminal nodes* (corresponding to terminal triangles of the stripification). The nodes with three incident dashed nodes correspond to isolated triangles in the stripification.

The first step of the operation consists, then, of searching a special kind of path in the graph called a *tunnel*. A tunnel is an alternating sequence of solid and dashed edges, starting and ending with a dashed edge, connecting two terminal nodes. Its length is always odd and we denote by *k*-tunnel a tunnel of length *k*.

If a tunnel is found, the second step consists, simply, of complementing the path, that is, changing each solid edge in a dashed edge and vice-versa. After this operation the number of solid paths (strips in the triangulation) on the graph is reduced by one. See Figure 3 for example.

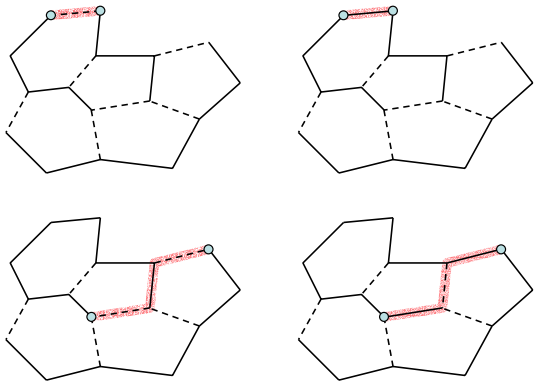


Figure 3: An example of tunnelling. In the top row a 1-tunnel is found; in the bottom row there are no 1-tunnels but only a 3-tunnel. Notice that the number of strips decreases from three to two after the first operation and to one after the second.

This technique can be used both to improve an existing stripification or to create a stripification from scratch. In the latter case the starting dual graph will have only dashed edges and every path of length one can be chosen as a tunnel. It is worthwhile to point out that isolated triangles are always considered as terminal nodes of a one-triangle strip.

The main problem when implementing the algorithm is the possibility that the graph traversal for tunnelling could select paths that, when complemented, would generate loops. It is thus necessary to follow two additional rules (we call them the *no-loop rules*) during the tunnel search to avoid this situation:

1. The last edge in a tunnel cannot connect two nodes belonging to the same strip (see Figure 4).
2. When a non-final dashed edge, *e* say, in the tunnel joins two nodes belonging to the same strip, the next solid edge should go back in the direction of the leading node of *e* (see Figure 5).

To be able to respect the no-loop rules, we need to distinguish between the different strips in the graph. This is done tagging each node of the graph (triangle) with an identifier corresponding to the strips it belongs to.

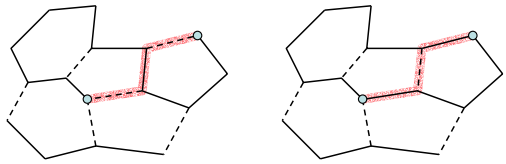


Figure 4: An incorrect tunnelling that generates a loop.

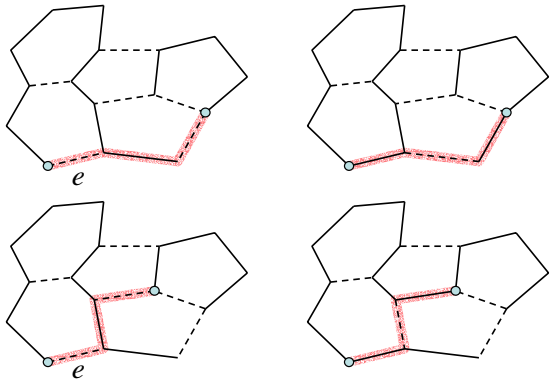
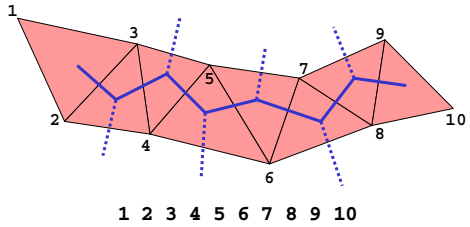


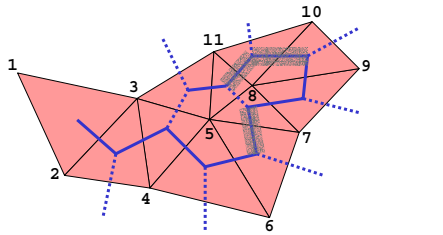
Figure 5: The non-final edge *e* in the tunnel joins two nodes belonging to the same strip. Of the two next possible steps, we must select the one corresponding to the direction that comes back to the leading node of *e* (bottom row), otherwise it will generate a loop (top row). One such step always exists because the leading and trailing nodes of e are in the same strip.

The only minor drawback of the tunnelling algorithm is that we are not able to keep the strips sequential, we are forced to use generalized strips and then introduce swap operations. This is to the fact that by its definition, the tunnelling operation *change the turns in the graph*.

A sequential strip (Figure 6.a) is, in the dual graph, a path of solid edges in which, at each node, we alternatively make a left and right turn. When tunnelling over the graph it is not possible to keep the strips sequential and we are thus obliged to use generalized strips (Figure 6.b).



(a)



(b)

Figure 6:A sequential strip (a) and a generalized one (b). For both we list the vertices to be sent to the CPU. The extra vertices (swaps) are in red. The grey edges mark the *wrong turns*.

4. Strip Rearrangement

The core of the algorithm is the rearrangement (a graph expansion or contraction) of the stripification when changing LOD. There are two method to consecutive operations applied to recolor the augmented graph: the first one is totally local to the triangle loop where the new vertex has been inserted and uses a look-up table; the second is *glocal*, it consists in launching a tunnelling on the modified stripification with a predefined stop rule.

First Step: Using a Look-up Table

We classified many different configurations that can be used to restructure the strips after a VS. Each single VS split operation insert two new triangles in the mesh, and three new edges in the dual graph.

We actually completed the task only for 4-vertices (loops of length 4 in the dual graph), where the VS can lead to two different topologies: two 4-vertices (two 4-loops sharing an edge) or a 3-vertex plus a 5-vertex (a 3-loop and a 5-loop sharing an edge). In this case all the possible configuration (*9+9*) allow graph recoloring without isolated triangles. In Figure 11 we list the nine configurations of the 4-loop transforming in a 3-loop plus 5-loop. Each couple of new triangles can be assigned to a single triangle strip, increasing its size by two. As it is possible to notice from the figure the strip section added is always a sequential one.

When dealing with higher degree vertices (longer loops) the cases increase rapidly. Splitting a 6-vertex, the most commonly found in triangular mesh, can lead to three different topologies: a 3-vertex plus a 7-vertex, a 4-vertex plus a 6-vertex and two 5-vertices. The problem is that, in this case, we are no longer able to always recolor the graph without leaving isolated triangles. We can see in Figure 7 a split with complete recolor while in Figure 8 there is a split leaving an isolated triangle. With 8-vertices, that appear very seldom in triangular meshes, we can be obliged even to leave both the inserted triangles isolated.

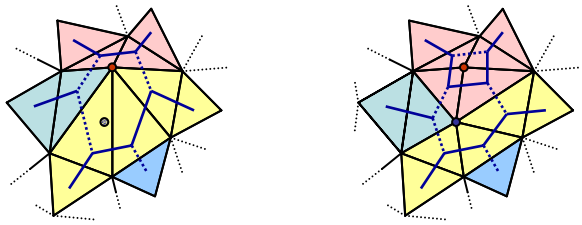


Figure 7: An example of possible graph recoloring after a VS.

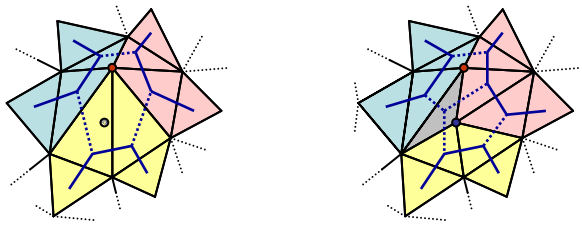


Figure 8: A configuration where the graph recoloring after a VS leaves an isolated triangle (marked grey).

In Figure 9 we can appreciate how the mechanism works. Passing from a LOD to a finer one the strip form stays more or less the same while its average length increases and a lot of isolated triangles appear.

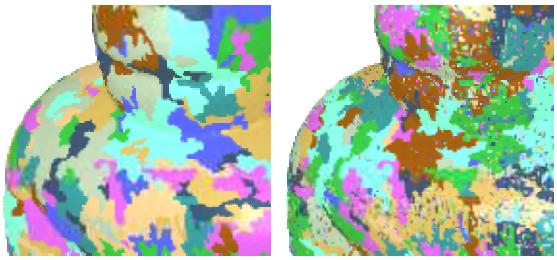


Figure 9: A close-up view of a local rearrangement performed on the *Dea madre* dataset.

Second Step: Using the Tunnelling Operator

Extensive benchmarks performed over different datasets, of different genus and size resulted in a percentage of recoloring operations introducing isolated triangles quite

constant: it varies in the range 45%-50%. In other words this corresponds to the insertion of an isolated triangle every second VS operation.

We thus need to repair the strips structure using what we have called a *glocal* tunnelling. The tunnelling operation is performed transparently from the user, and uses the isolated triangles as seeds for searching very short tunnels (starting from 1-tunnels). Since we apply the global operator in a local surrounding of these triangles we can say that it is used glocally. The tunnelling is then iterated until the number of isolated triangles reach a number that is smaller than 10\% of the total number of strips. This value is quite empirical: we noticed that when reaching this ratio, the frames per second rate almost doubles at any resolution for any dataset we used.

In Figure 10 we can appreciate how the rearrangement via the tunnelling works. The strips are completely restructured, there are many less isolated triangles and the average length increase while the maximal length tends to decrease.

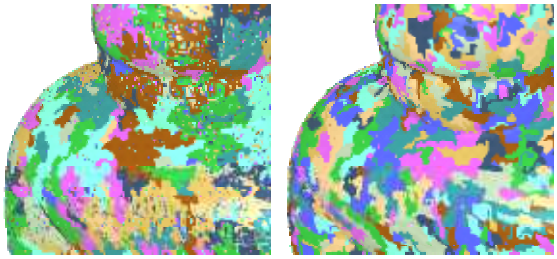


Figure 10: A close-up view of a rearrangement performed on the *Dea madre* dataset using a tunneling operation.

At any stage the user has the possibility to invoke the stripification process explicitly, specifying the maximal tunnel length. We decided to leave this possibility more for completeness than for real need. It is, in fact, quite difficult to significantly improve the results obtained automatically.

5. Results and Discussion

We have performed all our benchmark on a PC with a Pentium IV 1.5 GHz CPU with 512 MB of RAM, and a NVIDIA GeForce TI 4600 GPU with 128 MB of RAM.

For sake of simplicity we present here only the results obtained on the largest dataset we used.

In Table 1 we list the characteristics of the obtained stripifications. We can notice that the number of isolated triangles depends more on the tunnel length than on the overall number of triangles in the mesh.

LOD%	Only local	Maximal tunnel length		
		5	9	13
13	6.139	1.676	841	489
	4.456	489	140	46
	12.11	44.38	88.44	152.10
27	17.592	3.025	1.536	869
	9.017	496	141	52
	8.77	51.04	88.13	147.59
52	27.766	5.115	2.617	1.489
	11.741	785	281	107
	10.70	58.13	113.62	199.70

Table 1: In each cell the first row shows the number of strips, the second the number of isolated tri's and the last the mean strip length.

In Table 2 we show the time, in seconds, used to refine the stripification obtained with only local refinement. We remind that the cost of the local refinement is included in the cost of performing a resolution change.

LOD%	Maximal tunnel length		
	5	9	13
13	6.484	4.438	5.312
27	4.360	1.750	2.579
52	8.750	5.155	4.156

Table 2: Time in seconds to refine the stripifications.

Average Cache Miss Ratio

The number of vertex cache misses plays a fundamental role in rendering efficiency [Dee95]. If we want to achieve a good rendering sequence than, the Average Cache Miss Ratio (ACMR), whose value ranges from *0.5* to *3*, should be kept as low as possible. To get this goal, several reordering algorithms has been proposed, for standard meshes [Cho97], triangle stripes [Hop99] and progressive meshes [Bog02].

We evaluated ACMR for several data set, using stripes generated with our system using the tunnelling algorithm. Without any kind of reordering mechanism, ACMR is *~0.7* for a cache of 32 positions for all data sets, compared to a typical ACMR of *~1.0* for standard stripification procedures. This suggests that stripes calculated with tunnelling algorithm have a *built-in* cache friendly attitude.

In Table 3 several ACMR values for different data sets are listed, depending on cache size.

The tunnelling algorithm behaves well because of the stripes' shape. As one can see, for instance, in Figure 12, stripes appear to be *packed* instead of being elongated as usual. This preserves locality also in vertex ordering and then cache friendly behavior.

Data Set	Cache size		
	16	32	64
Oilpump	0.77	0.70	0.64
David	0.78	0.71	0.68
Dea Madre	0.78	0.71	0.67

Table 3: ACMR values for three different data sets based on the cache size.

6. Conclusions and Future Work

We presented a simple but very effective algorithm allowing to compute an optimal stripification on a progressive mesh. Optimal, in this context, means to at least double the frames per second rate with respect to non stripified mesh.

The method we used is a two steps one: first we recolor the dual graph of the mesh using a look-up table and then we transparently launch a tunneling algorithm with a short tunnel length.

We are already planning to get a better insight about the look-up table. As we already mentioned we are not able to automatically avoid the creation of isolated triangles only looking at the strips passing through the loop of triangles sharing the vertex to split. We think that extending the analysis also to the neighbor triangles (say, the triangles that can be reached from the split vertex traversing two edges) can help to increase the number of recoloring without creation of isolated triangles.

Another line of development regards a better analysis of the capabilities of vertex arrays on the GPU. At present we don't clip the triangle strips in chunks the best fit on the arrays and we should insert a further parameter in the visualization tool to take this into account.

The last improvement we are planning is on the fine tuning of the rendering. At present we can select the LOD and then verify the fps obtainable. In the next release it will be possible to set the fps budget and let the system select the possible LOD visualizable.

Acknowledgements

The *Dea madre* dataset was obtained from tridimensional scans of manufacts exposed at the Museo Archeologico Nazionale in Cagliari. We are indebted to its director, Carlo Tronchetti, for letting us use these digital data and to the VCG of the ISTI-CNR in Pisa for the hardware and software used in the acquisition and reconstruction.

We thank Daniele Vacca for his work on the visualization tool.

7. References

[All01] Alliez P. and Desbrun M. *Progressive compression for lossless transmission of triangle meshes*. In Proceedings of the 28th annual conference on Computer graphics and interactive techniques, pages 195–202. ACM Press, 2001.

[Ark96] Arkin E. M., Held M., Mitchell J. S. B., and Skiena, S. S. *Hamiltonian triangulations for fast rendering*. The Visual Computer 12, 9 (1996), 429–444.

[Bog02] Bogomjakov A. and Gotsman, C. *Universal Rendering Sequences for Transparent Vertex Caching of Progressive Meshes*. In Computer Graphics Forum 21 , 2 (June2002)

[Cho97] Chow M. M. *Optimized geometry compression for real-time rendering*. In IEEE Visualization '97 (Nov. 1997), pp. 346–354.

[Dee95] Deering, M.F. *Geometry Compression*. In Proceedings of SIGGRAPH 95, pp. 13–20.

[ElS00] El-Sana, J., Evans, F., Kalaiah, A., Varshney, A., Skiena, S., and Azanli, E. *Efficiently computing and updating triangle strips for real-time rendering*. Computer-Aided Design 32, 13 (Oct. 2000), 753–772.

[ElS99] El-Sana J. A., Azanli E., and Varshney A. *Skip strips: Maintaining triangle strips for view dependent rendering*. In IEEE Visualization '99 (Oct. 1999), pp. 131–138.

[Est02] Estkowski R., Mitchell J. S. B., and Xiang, X. *Optimal decomposition of polygonal models into triangle strips*. In Proceedings of the eighteenth annual symposium on Computational geometry (2002), ACM Press, pp. 254–263.

[Eva96] Evans F., Skiena S. S., and Varshney A. *Optimizing triangle strips for fast rendering*. In IEEE Visualization '96 (Oct. 1996), pp. 319–326.

[Gan02] Gandoin P.M. and Devillers O. PROGRESSIVE LOSSLESS COMPRESSION OF ARBITRARY SIMPLICIAL COMPLEXES. In 2002 Proceedings of the 29th annual conference on Computer graphics and interactive techniques, San Antonio, Texas, ACM Press, pp. 372–379.

[Gar76] Garey M. R., Johnson D. S., and Tarjan R. E. *The planar hamiltonian circuit problem is NP-complete*. SIAM Journal of Computing 5, 4 (Dec 1976), 704–714.

[Hae03] Haeyoung L., Desbrun M. and Schröder, P. Progressive encoding of complex isosurfaces. In ACM Trans. Graph. 22, 3, pp. 471–476.

[Hop96] Hoppe H. *Progressive meshes*. In Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, 1996, ACM Press, pp. 99–108.

[Hop97] Hoppe, H. *View-Dependent Refinement of Progressive Meshes*. In Proceedings of SIGGRAPH 97, pp. 189–198.

[Hop98] Hoppe, H. *Efficient implementation of progressive meshes*. In Computers & Graphics 22, 1, pp. 27–36.

[Hop99] Hoppe, H. *Optimization of mesh locality for transparent vertex caching*. In Proceedings of SIGGRAPH 99 (Aug. 1999), Computer Graphics Proceedings, Annual Conference Series, pp. 269–276.

[Ise01] Isenburg M. *Triangle strip compression*. Computer Graphics Forum 20, 2 (2001), 91–101.

[Paj00] Pajarola R. and Rossignac J. *Compressed Progressive Meshes*. In 2000 IEEE Transactions on Visualization and Computer Graphics 6, 1 (Jan. - Mar. 2000), pp. 79–93.

[Pop97] Popovic J. and Hoppe H. *Progressive Simplicial Complexes*. In Proceedings of SIGGRAPH 97, pp. 217–224.

[Por03] Porcu M. and Scateni R. *An Iterative Stripification Algorithm Based on Dual Graph Operations*. In Proceedings of Eurographics 2003 (short presentations) (Sep. 2003) pp. 69–75.

[Ros99] Rossignac J. *Edgebreaker: Connectivity Compression for Triangle Meshes*. In 1999 IEEE Transactions on Visualization and Computer Graphics 5, 1 (Jan. - Mar. 1999), pp. 47–61.

[RsS99] Rossignac J. and Szymczak A. *Wrap&Zip decompression of the connectivity of triangle meshes compressed with Edgebreaker*. In Computational Geometry 14, 1-3 (1999), pp. 119–135.

[Spe97] Speckmann B. and Snoeyink J. *Easy triangle strips for TIN terrain models*. In Canadian Conference on Computational Geometry (1997), pp. 239–244.

[Ste01] Stewart A. J. *Tunneling for triangle strips in continuous level-of-detail meshes*. In Graphics Interface (June 2001), pp. 91–100.

[Tau98] Taubin G. and Rossignac J. *Geometric Compression Through Topological Surgery*. In 1998 ACM Transactions on Graphics 17, 2 (Apr. 1998), pp. 84–115.

[Tou98] Touma C. and Gotsman C. *Triangle Mesh Compression*. In Graphics Interface '98 (Jun. 98), pp. 26–34.

[Xia99] Xiang X., Held M., and Mitchell J. S. B. *Fast and effective stripification of polygonal surface models*. In 1999 ACM Symposium on Interactive 3D Graphics (Apr. 1999), pp. 71–78.

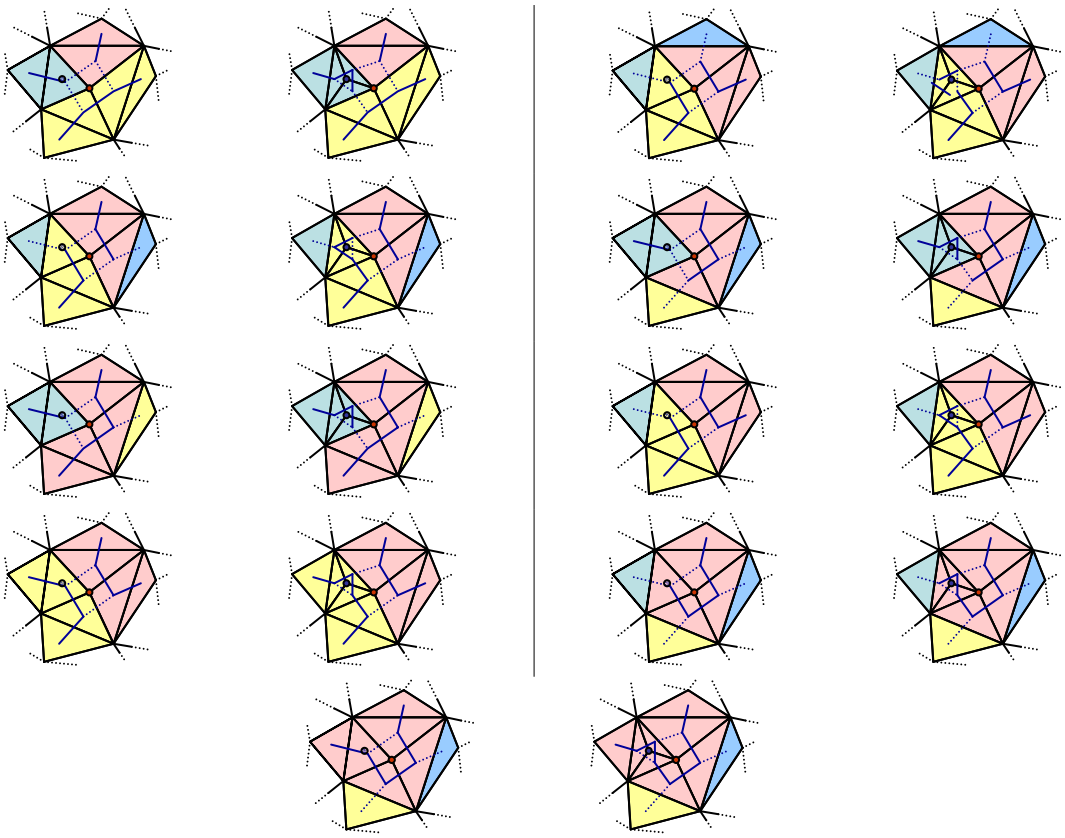


Figure 11: The graph rewriting rules to apply when inserting a new vertex with a VS operation. In each couple, on the left the configuration before the VS (the vertex to be split is marked in red), on the right the configuration after the VS (the new inserted vertex is marked in blue).

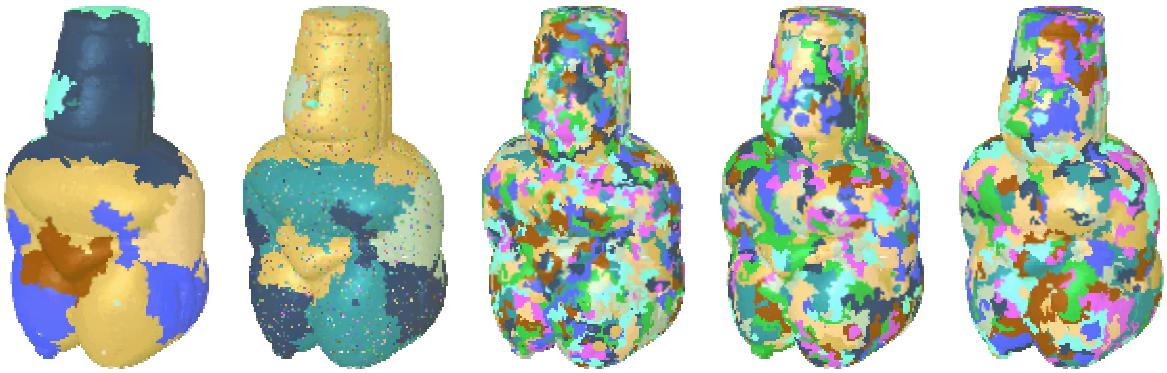


Figure 12: An example of LOD change on the "Dea madre" dataset. From left to right: the 9% LOD optimally stripified (51,511 tri's and 10 strips); the 13% LOD obtained from the 9% one only with local mesh restructuring operations (this and the subsequent are meshes of 74,381 tri's, 6,139 strips with 4,456 isolated tri's); the same mesh after a 6-tunnels search (1,676 strips with 489 isolated tri's); after a 10-tunnels search and graph recoloring (841 strips with 140 isolated tri's); after a 14-tunnels search and graph recoloring (489 strips with 46 isolated tri's).