University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Triangle Strips For Fast Rendering

The State Of The Art And Concept Of PhD. Thesis

Petr Vaněček

Technical Report No. DCSE/TR-2004-05
April, 2004

Distribution: public

Technical Report No. DCSE/TR-2004-05
The State Of The Art And Concept Of PhD. Thesis
April 2004

Triangle Strips For Fast Rendering

Petr Vaněček

Triangle surface models are nowadays most often types of geometric objects description in computer graphics. Therefore, the problem of fast visualization of this type of data is often being solved. The speed of high performance rendering engines is usually bounded by the rate at which triangulated data is sent into the machine. One can reduce the time needed to transmit the set of triangles by compressing the topological information and decompressing at the rendering stage. As neighboring triangles share an edge, it is possible to avoid sending the common vertices twice by special order of triangles, called triangle strip.

This work presents an overview and a comparison of existing stripification methods. It also introduces a new stripification algorithm for terrain models based on Delaunay triangulation that can be modified to handle LOD. Finally an outlook for the future work is sketched out.

This work was supported by by the Ministry of Education of The Czech Republic - project MSM 235200005..

Copies of this report are available on
http://www.kiv.zcu.cz/publications/
or by surface mail on request sent to the following address:

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Acknowledgments

I would like to thank to Prof. V. Skala for providing good conditions under which this work was born. My thanks also belong to Prof. J. Stewart for a big inspiration and for the source code of the tunneling algorithm, to X. Xiang and Prof. J. S. B. Mitchell for providing the source code of FTSG and to D. Kornmann for technical support for his program. I also would like to express my thanks to Prof. B. Žalik for a week full of interesting discussions and new ideas. Last but not least I would like to thank to my advisor Doc. I. Kolingerová for her time, care and patience …

Contents

1 Introduction

Triangle surface models (often called *meshes*) are nowadays the most often types of geometric objects description in computer graphics. These models are often used for various kind of applications such as CAD/CAM, VR, medical data or computer games. Therefore, the problem of fast visualization of this type of data is often being solved.

The performance of today's rendering hardware is usually very high and the speed of the rendering is bounded not only by the power of the GPU but also by the the rate at which the triangulated data is sent into the GPU. To decrease the amount of data, one can use some techniques to prevent sending of unnecessary triangles (e.g., visibility culling) or some kind of simplification of complex objects (e.g., (C)LOD). Still it is important to reduce the time needed to transmit the set of triangles by compressing the topological information and decompressing at the rendering stage. As neighboring triangles share an edge, it is possible to avoid sending the common vertices twice by a special order of triangles, called a *triangle strip*.

Evans et al., showed that covering the mesh by an optimal set of triangle strips is NP-hard [15]. To compute a stripification in a polynomial time, it is necessary to use some heuristic that finds some local optimum. As the number of triangles in meshes grows as fast as the power of GPUs and the bus bandwidths, the stripification topic is still very important and many algorithms on stripification exists.

In this work, an overview of existing methods is presented (Chapter 3). Basic principles of these methods are described with a short conclusion for each method. For all available methods for 3D meshes, I have made an overall comparison of stripification quality, including the running time of the methods and the rendering speed. I have been in contact with many other authors during writing this work (Stewart, Xiang, Mitchell, Kornmann, Pedrini) and all of them appreciated such a complex comparison. On the basis of the communication it seems that a cooperation with some of these authors can be established.

I will also introduce a new stripification method based on the Delaunay triangulation that was first published on the SCCG'03 conference (Chapter 4). This method produces a stripification of a low quality, but it can be adopted for a visualization of LOD of terrain models.

Currently I am working on a new algorithm based on a duality approach. Some sketches of the algorithm as well as some ideas of a future work are discussed in Chapter 5.

2 Triangle Strips

A *sequential tristrip* is a sequence of $n + 2$ vertices that represents n triangles: in Figure 2.1 (a) the sequence (1,2,3,4,5,6) corresponds to triangles $\Delta 123$, $\Delta 234$, $\Delta 345$ and $\Delta 456$. Using the sequential tristrip, the transmit cost of n triangles can be reduced by the factor of three (from $3 \cdot n$ to $n + 2$ vertices).

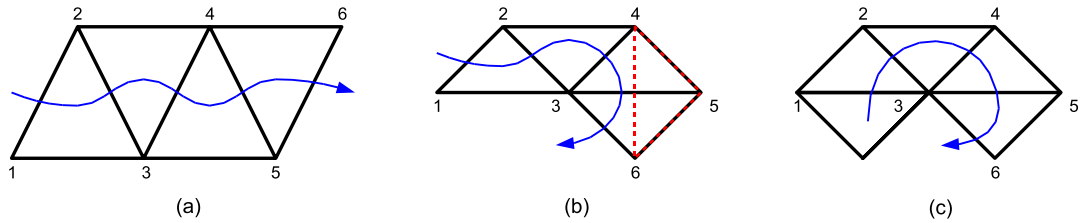


Figure 2.1: An example of a sequential triangle strip (a), a generalized triangle strip (b) and a triangle fan (c).

There also exist situations where the triangle adjacency does not allow a sequential encoding. In Figure 2.1 (b) the sequence (1,2,3,4,5,6) produces an invalid triangle $\Delta 456$. An extra vertex has to be added to change the sequence to (1,2,3,4,3,5,6). This operation is called a *swap* and tristrips with swaps are called *generalized tristrips*. Still, the transmit cost is reduced more than twice (from $3 \cdot n$ to $n + 2 + swaps$ vertices).

In some special cases it is also possible to use a special type of generalized triangle strip called a *triangle fan*. The *fan* is defined by the central vertex and its neighboring vertices. In Figure 2.1 (c) the fan is defined by a sequence (3,1,2,4,5,6). As the length of the *fan* is usually very low (the average number of neighboring vertices in a usual mesh is six), it is not used very often in practice.

3 Methods

In this section, a possible classifications of stripification methods are presented. Also an overview and some comparison of existing methods is done. This comparison is based on the references or on my own measurements if the source code is available.

A variety of different approaches for creating triangle strips were made. As the searching of optimal stripification is NP-hard problem [15], all those algorithms use some kind of heuristic function. According to the type of the heuristic function, it is possible to classify stripification methods into three groups (this classification is used within the scope of this work).

- Direct methods use the information about the triangle mesh – number of neighbors, topological information about a region, etc.
- Duality based methods convert the mesh into a dual-graph [9] (i.e., a graph where a node represents a face and neighboring faces are connected with an edge in the graph). These methods often produce a stripification with better properties than the straightforward methods
- Miscellaneous approaches are using some other techniques to produce the stripification. Usually, these methods build a stripification from a more general type of input data (e.g., creating a triangulation together with the stripification from a set of points)

Very often, the heuristic function only decides in which direction the strip should continue. For such a decision only some local criterion is sufficient. To obtain a better stripification, some global heuristic is necessary.

- Local heuristics use some local criterion to decide whether to connect a triangle to a strip or not. This criterion often leads to a stripification with a high number of triangle strips. Furthermore, such stripification often contains a huge number of short strips.
- Global heuristics are searching for triangle strips by using a global criterion. Usually it takes a longer time to produce triangle strips by such an algorithm, but in most cases, these strips are better. Many global algorithms are based on the graph duality.

Furthermore, the term 'optimal stripification' is not uniquely determined. One can optimize the stripification algorithm to produce low number of vertices needed for strips, to decrease the amount of data sent through the bus to the rendering engine and speed

up the rendering. As the initialization of a new triangle strip costs some extra time, it is also desirable to minimize the number of generated triangle strips. It is not possible to minimize both these parameters at once – decreasing the number of triangle strips often leads to increasing the number of vertices (due to higher number of swaps, needed to preserve the strip) and vice versa. Very often, the stripification algorithms contain more heuristic functions for vertex or strip optimization.

- Strips minimizing algorithms minimize the number of triangle strips. As the initialization phase of a triangle strip takes some additional time, minimizing the number of strips speeds up the rendering.
- Vertices minimizing algorithms minimize the number of vertices (swaps). Reducing the number of vertices leads to higher performances, because there is lower bus traffic and less transformations and lighting operations[1].

To be able to visualize huge data sets, clipping or decimation are often used. To be able to clip invisible regions, it is necessary to optimize the stripification to create local strips (i.e, strips that traverse cross the whole triangulation). While using some model decimation – usually by edge collapsing – triangle strips could be broken. To avoid these breaks, a stripification that preserves triangle strips has to be used. There is also a possibility to use some local repairs during the simplification process.

- Stripification for static meshes does not care about the changes in topology. While using some kind of simplification, the number of triangle strips increases, due to strip breaks.
- Stripification for CLOD tries to construct triangle strips that are being preserved during the simplification.

[1] Some high-end graphic systems has a one-bit flag for triangle swaps, thus there is no necessity to minimize the number of swaps.

3.1 Direct methods

As mentioned above, the direct methods use directly the information from triangle or polygonal mesh, to produce triangle strips. Nearly all methods use the criterion of number of neighbors (local criterion) to decide whether to connect a triangle to a strip or not.

3.1.1 SGI method

Akeley et al. [2] have developed one of the first stripification algorithms, known as *SGI* or *tomesh* that converts a fully triangulated mesh into triangle strips. It is a simple greedy algorithm which uses a local criterion.

This algorithm (Figure 3.1) tries to build triangle strips which do not divide the remaining triangulation into too many small pieces. The strip is starting with the triangle with the least number of neighbors. Then a greedy heuristic is used to add adjacent triangles with the least number of neighbors to a strip. If more triangles with the same number of neighbors exist, the algorithm looks one step ahead. If there is no neighboring triangle, a new strip is created. The algorithm stops after all triangles were added to strips.

```
while there is any triangle in the mesh do
  start a new strip
  choose a triangle with the least number of neighbors
  add the triangle to the current strip
  remove the triangle from the mesh
  update the number of neighbors
  while there exists a neighbor do
    choose a neighbor with the least number of neighbors
    if there is an equality then
        look one step ahead

    add the triangle to the current strip
    remove the triangle from the mesh
    update the number of neighbors
  end while
end while
```

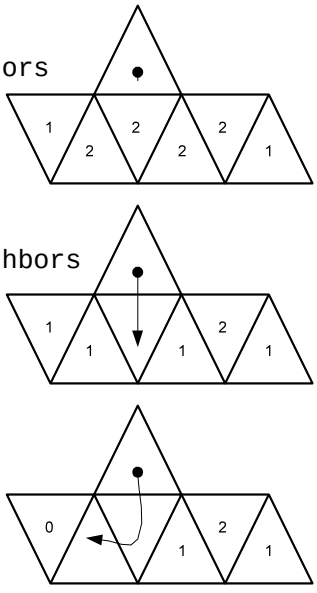


Figure 3.1: Pseudo-code of the basic *SGI* algorithm and an example of strip construction.

The time complexity of this algorithm is $O(n+s.n)$, where n is the number of triangles and s is the number of triangle strips. To reduce the complexity to $O(n)$, it is necessary to use some additional data structures (a hash table, or a priority queue) to be able to find the starting triangle quickly.

It is quite easy to change the heuristic function (i.e., the criterion which chooses the next triangle), thus many modifications of this algorithm exist.

Although this algorithm is very simple, it produces quite a good stripification in a short time. As this algorithm was designed for Iris-GL (which uses 1-bit flag for the swap), it does not care about swaps. It uses a local heuristic function and that is why it produces a big number of short strips.

3.1.2 Fast And Simple Triangle Strip Generation – Weighted SGI

The most important part of the *SGI* algorithm is the heuristic function, which chooses the next triangle. Kornmann [24] implemented an algorithm that combines several weighted heuristic functions to improve the quality of triangle strips.

The first heuristic function is based on the *SGI* criterion, i.e. it returns the number of neighboring triangles (mesh connectivity). As mentioned before, such a heuristic creates strips containing triangles with low number of neighbors. This avoids the emergence of short strips later on. In Figure 3.2 (a) the triangle on the right of the current triangle will be added to the strip because it has no other neighboring triangle.

The second heuristic function uses the triangle vertices' connectivity. It evaluates the number of triangles connected to each vertex and it returns $+1$ for the highest connected node and -1 for the other nodes. This heuristic preserves the highly connected nodes for remaining triangles. In Figure 3.2 (b) the triangle on the right of the current triangle will be added to the strip because the corresponding vertex has lower connectivity.

The last heuristic function analysis locally whether the strip will need a swap to include the next triangle. I returns $+1$ if a swap is needed and -1 if it is not needed. This heuristic leads to straight triangle strips without swaps covered by a minimal number of vertices. In Figure 3.2 (c) the situation on the left side (no swap) will be preferred.

Combining all these functions, the weight for each neighboring triangle is calculated. Then a triangle with the smallest weight is used in the strip. If there is a tie, the triangle is chosen randomly. The strip ends either when there is no neighboring triangle, or when the strip reaches a sufficient length.

With default settings (all heuristics on) the algorithm produces less strips than the *SGI* algorithm, but these strips are covered by more vertices. This is quite surprising. As the algorithm uses the vertices minimizing criterion, I expected that the number of vertices will be lower than *SGI*. Unfortunately, the source code of this algorithm is not available, thus I was not able to go into more details. Furthermore, the algorithm uses a randomization and each time it produces different results.

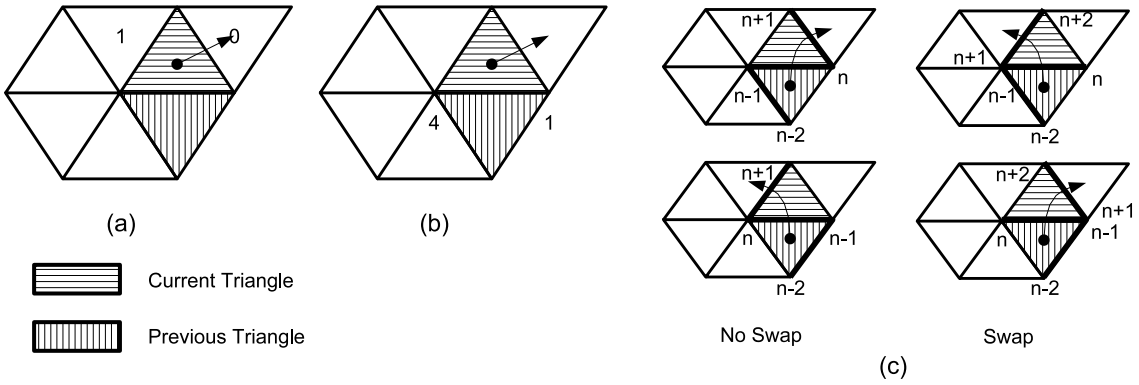


Figure 3.2: Three types of heuristic functions in Kornmann's algorithm. A mesh connectivity criterion (a), a vertex connectivity criterion (b) and vertices minimizing criterion (c). (From [24])

The nice property of the algorithm is that all three functions can be combined to achieve a better stripification for the given purpose. Regrettably, I was not able to make more tests of this algorithm as the source code is probably lost. It can be used for static, fully triangulated meshes only.

3.1.3 Fast Mesh Rendering Through Efficient Triangle Strip Generation – SStrip

Another algorithm based on the *SGI* was developed by da Silva [29]. This algorithm uses a local strategy based on a simultaneous construction of strips. The algorithm maintains s strips being built and at each step adds a triangle to one of the strips.

The algorithm chooses s triangles as beginnings of s strips, following the same criterion as *SGI* method, i.e., the lowest number of neighbors. To avoid an immediate strip concatenation, a new restriction is added – the beginning of a new strip may not be adjacent to the extremities of an existing strip.

The next triangle that will be added to a strip is chosen from all candidate triangles (i.e., all triangles neighboring to both extremities of all triangle strips) following this order:

1. If a neighboring triangle has degree 0, it is added immediately to avoid a singleton strip.

2. If there is no neighbor with degree 0, the neighbor with degree 1 is chosen. In case of a tie, a look-ahead test is performed as follows. If the adjacent triangle has degree 1, it is inserted, otherwise, the triangle that does not produce a swap (vertices minimizing heuristic) or the triangle with neighbor with lower degree (strips minimizing heuristic) is chosen.

3. If all neighbors have degree 2, the chosen triangle is the one that does not produce a swap.

In some cases the insertion of a triangle can cause that extremities of two strips become adjacent and these two strips can be concatenated and a new strip has to be created. The concatenation is performed according to the following rules:

- A triangle $T1$ of degree 0 is adjacent to two strip extremities – both strips are concatenated (in Figure 3.3 (a) the triangle $T1$ is adjacent to Strip 1 and Strip 2; these strips are concatenated).
- A triangle $T1$ of degree 0 is adjacent to three strip extremities – a concatenation that does not produce a swap is chosen (in Figure 3.3 (b) Strip 1 and Strip 2 are concatenated).
- A triangle $T1$ of degree 1 is adjacent to a strip and to a triangle $T2$ – if the triangle $T2$ has a degree 1, it is connected to $T1$ to avoid a singleton strip, otherwise, the two strips are concatenated (in Figure 3.3 (c) both triangles $T1$ and $T2$ are added to Strip 1).

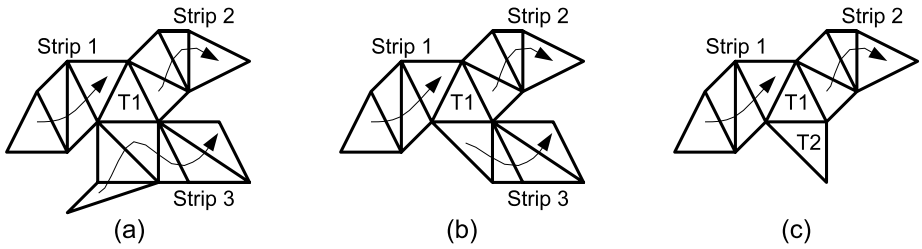


Figure 3.3: Concatenation rules in da Silva's algorithm: If a triangle of degree 0 is adjacent to two strip extremities, the strips are concatenated (a), if it is adjacent to three strip extremities, the concatenation that does not produce a swap is chosen (b). If a triangle of degree 1 is adjacent to two strip extremities and to other triangle, the concatenation depends on the degree of the neighboring triangle (c). (From [29])

A pseudo-code of da Silva's algorithm and an example of simultaneous strip construction is presented in Figure 3.4.

This algorithm is designed for static, fully triangulated meshes. It produces a stripification with lower number of strips and vertices than the *SGI* method. This improvement is partially caused by the multiple strip construction. According to the measurement, it seems that optimal number of simultaneous constructed strips is 2 or 4, but the differences are not significant. The source code of this algorithm is available on the internet [28].

```
start s new strips
while there is any triangle in themesh do
  while insert degree 0 candidate do
    try concatenate strips
    create new strips
  end while
  if insert degree 1 candidate then
    try concatenate strips
    create new strips
  else insert degree 2 candidate then
    try concatenate strips
    create new strips
  end if
end while
```

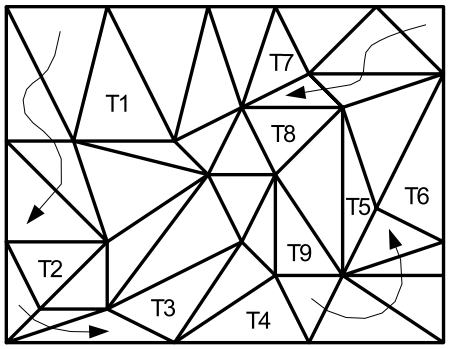
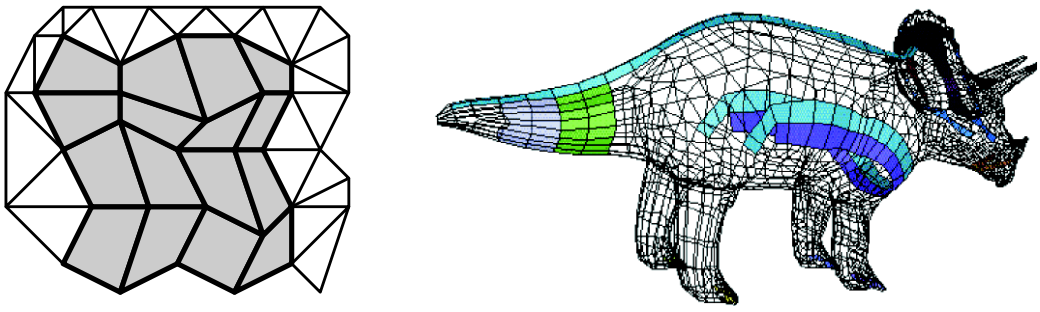


Figure 3.4: Pseudo-code of da Silva's algorithm and an example of simultaneous construction of strips. (From [29])

3.1.4 STRIPE

To improve the quality of stripification, Evans, et al. [16] developed an algorithm that uses a global analysis of the structure of a polygonal model. The algorithm is designed for polygonal (i.e., not fully triangulated) meshes. In such a type of meshes, there are usually many quadrilateral faces, often arranged in large connected regions. The global heuristic attempts to find large rectangular regions consisting only of quadrilaterals – "patches" (see Figure 3.5). These patches are triangulated along each row or column and then stripified.



(a) (b)

Figure 3.5: An example of a rectangular patch (a) and a typical polyhedral mesh with patches (b). (From [16])

To compute the number of polygons in a patch, it is necessary to examine quadrilaterals in both directions (east-west and north-south). It is also necessary to ensure that these quadrilaterals are all adjacent. To avoid generating too many small patches, a minimal patch cutoff size is predefined. This cutoff size defines the smallest patch that is generated.

To stripify the patches, two different approaches were implemented. The first approach – *row/column strips* – partitions the patches into sequential strips along rows/columns depending on the length of patch in the given direction. By generating one strip along each row, the number of swaps is minimized. The second approach – *full-patch strips* – converts each patch into one generalized strip, at a cost of 3 swaps per turn. Furthermore, each strip is then extended from both extremes to neighboring quadrilaterals. Such an approach minimizes the number of strips.

After the global heuristic, an *SGI* based algorithm is used to stripify the remaining polygons. For the triangulation of polygons, three different approaches were suggested. The *static triangulation* triangulates all faces in a preprocessing stage, using the alternate left-right turns. Such a triangulation is more complex than the conventional fan triangulation, but it produces triangles that can be stripified by a sequential strip. The *dynamic whole-face triangulation* triangulates a face when a strip first enters it. Even better results can be achieved by the *dynamic partial-face triangulation*. This approach allows to triangulate only a part of the polygon that the strip is going through. The remaining part of the polygon can be triangulated later on to allow to create a better triangle strip.

The algorithm is designed for static, not fully triangulated meshes with convex polygons. For such meshes the algorithm produces very good triangle strips in nearly linear time. According to authors' tests [16], the best stripification is obtained by the global row/column strips with a cutoff size of 5, using the dynamic whole-face triangulation.

For fully triangulated meshes it produces a higher number of strips but a bit lower number of vertices than the *SGI* method. The time needed for stripification is higher than the *SGI* method. The implementation can be downloaded for free [14] and it is used in many other papers for comparison[2].

[2]There were some bugs in the old implementation of *STRIPE* – it was not very stable and it was not able to stripify large models. Many authors reported similar experience. These troubles were solved in the new version. There are also some inconsistencies in how *STRIPE* count strip vertices, thus some results may differ from other papers.

3.2 Duality based methods

The group of duality based methods uses the dual-graph of triangulation. Usually some existing graph algorithm is applied to obtain a set of paths (paths are dual to strips) from this graph.

3.2.1 Fast Triangle Strip Generator – FTSG

Xiang et al. [39] developed a stripification algorithm based on a spanning tree algorithm and a careful partitioning into a set of paths. The algorithm can generate triangle strips from a polygonal mesh, containing even non-convex polygons. The algorithm tries to minimize the number of vertices, thus mainly sequential strips are used.

The algorithm can be divided into five basic steps that will be explained in a more detailed way (Figure 3.6).

1. Compute a triangulation of non triangle faces
2. Construct a spanning tree in the dual graph of the triangulation
3. Partition the spanning tree into a set of paths
4. Decompose the paths into sequential strips or fans
5. Concatenate short strips into longer strips, using a set of postprocessing heuristics

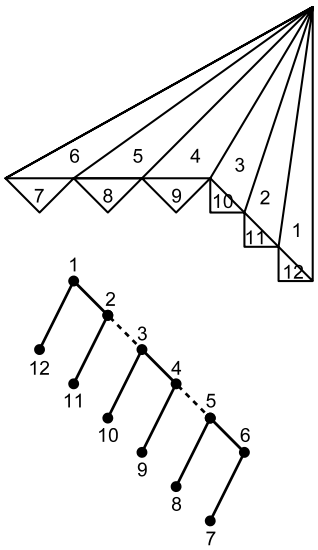


Figure 3.6: FSTG algorithm steps and an example of a triangulation and its corresponding dual graph. (From [39])

To make a fully triangulated model, a modification of a very robust algorithm (*FIST* [19]) is integrated. This algorithm allows to triangulate both convex and non-convex polygons in the mesh, even if the mesh is degenerated or corrupted. The modification of *FIST* used in the stripification outputs triangulation convex faces that are pure tri-strips. This part of algorithm is done in the worst-case $O(n \log n)$ time (for faces with holes). In practice, it takes only a linear time, as most polygons in the real-world meshes tend to be triangles, quadrilaterals, or low-cardinality polygons.

For the construction of a spanning tree, three different approaches were implemented. The standard breadth-first search (BFS, Figure 3.7 (a)), depth-first search (DFS, Figure 3.7 (b)) and a hybrid variant of search that does BFS, but returns to the highest not yet fully explored node (Figure 3.7 (c)). The goal of the Step 2 is to build a spanning tree that has a small number of nodes of degree two, as such nodes lead to a high number of paths. The BFS tends to generate nearly balanced binary trees, therefore the number of nodes of degree two may be large. The hybrid search and DFS both tends to produce more nodes of degree one and the number of generated paths is lower. According to the results, the DFS produces the best spanning tree from all three approaches. While searching for the spanning tree, the algorithm has to decide, which triangle to visit in the next step. As the goal of the algorithm is to minimize the number of vertices, it chooses a triangle that does not produce a swap.

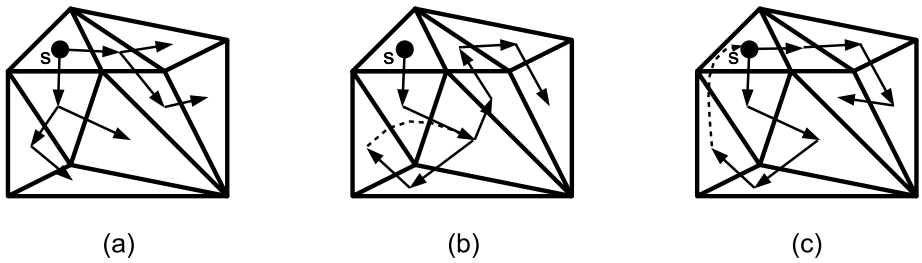


Figure 3.7: Three approaches for the construction of spanning tree. Standard breadth-first search BFS (a), depth-first search BFS (b) and a hybrid search (c). *S* is the root node of the spanning. A dashed line represents returns in the graph search algorithm. (From [39])

For the partitioning of the spanning tree into a set of paths, a dynamic programming optimization is used. For each node of the spanning tree an objective function being the minimal number of sequential strips that can be derived from the subtree is defined. By traversing the tree in a bottom-up fashion and storing the optimal decomposition at every node, one can achieve the optimal decomposition. This computation can be done in linear time, as there are only a constant number of cases per node and each node is visited exactly once.

To minimize the number of vertices, the authors prefer to use sequential strips only. In this case, the decomposition step only converts the list of triangles into a list of vertices. They have also implemented a decomposition using triangle fans (which, in practice, produces the worst results) and a combination of both sequential strips and fans. In the last case, i.e., strips and fans, the fan starts only if the greedy decomposition encounters four consecutive triangles that cannot be encoded with a sequential strip.

In the last step the strip concatenation is performed. For this concatenation the zero-area triangles, i.e., swaps, have to be allowed. There exist several typical configurations in the stripification that can reduce the number of strips by one at a cost of 0,1 or 2 vertices. The concatenation algorithm is looking for such configuration and performs them. Such an approach optimizes the stripification only locally for one pair of neighboring strips. As the strip may have more candidates for the concatenation, the order of the concatenation matters. The global optimization can be achieved by similar multi-passes algorithm, but it will increase the running time and memory usage.

The algorithm is designed for static not-fully triangulated meshes, but the triangulation phase is done before the start of the stripification process itself. The number of vertices produced by the stripification is about 10% lower than by the *SGI* method but it produces significantly more strips than *SGI*.

3.2.2 Easy Triangle Strips For TIN

Speckmann and Snoeyink [30] suggested an original approach of creating triangle strips for triangulated irregular networks (TIN). They use the De Berg's algorithm for traversing a subdivision of a plane [6].

The idea of this algorithm is to define an order of triangles in the triangulation. For each triangle an adjacent predecessor is defined and a directed graph is formed. A stripification can be easily obtained by the depth-first search traversal of this graph. As such an operation is defined for the whole triangulation, no additional data for the stripification are needed.

The predecessor relation is defined as follows. First, an arbitrary (starting) triangle and its inner reference point p is chosen. Then, for each triangle T except for the starting triangle the point of T closest to p under Euclidean distance is computed. If the closest point is an inner point of an edge e of the triangle T, then the predecessor of T is the other triangle T_p adjacent to the edge e (Figure 3.8 (a)). Otherwise the closest point is one of vertices of T with the edge e just before and e' just after (having the counterclockwise orientation). If the edge e is exposed to p (i.e., the directed line induced by $\vec{e}$ has p strictly to the right) the triangle T_p adjacent to e (Figure 3.8 (b)), otherwise the triangle T'_p adjacent to e' is chosen as a predecessor of T (Figure 3.8 (c)).

The graph induced by such a criterion is connected and it is a spanning tree of the dual graph of triangulation. This tree has two nice properties. First, the branches of the tree tend to alternate the left and right turn – the stripification generated from these branches is more or less sequential. Second, the information about the spanning tree does not need to be explicitly stored, as the predecessor relation can be computed in a constant time.

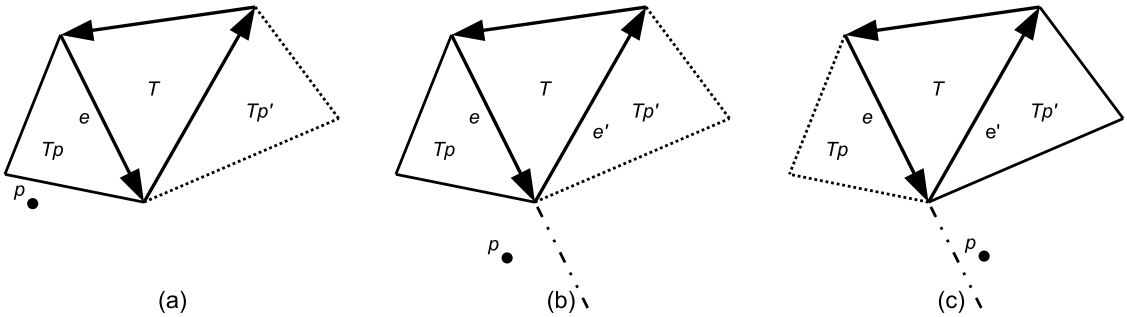


Figure 3.8: If the closest point to point p is an inner point of the edge e of the triangle T, then T_p is the predecessor of T (a). If the edge e is exposed to p, then T_p is the predecessor of T (b) else T'_p is predecessor of t (c). (From [30])

The basic tristrip can be constructed directly from the spanning tree, following the tree in depth-first manner and starting a new strip every time the sequence of left-right turns is disturbed (Figure 3.9 (a)). Sometimes, it is possible to connect a singleton strip to some existing strip, even if they are not connected in the spanning tree, thus reduce the number of singleton strips (Figure 3.9 (a), dotted line). As the TIN algorithm does not store any additional data, the insertion of an unconnected triangle is not done in linear time in the number of triangles.

To decrease the number of triangle strips and vertices, it is also possible to allow the swaps. They can be allowed by a simple modification of the traversal algorithm and no additional data are needed. This modification does not break the strip, but it continues even if the sequence of left-right turns is disturbed. In Figure 3.9 (b), dense dashed lines shows parts of strips that do not full fill the alternating left-right sequences.

It is also possible to look for nodes where the tree is ”wide”, i.e., the node has both children, the left child has its own left child and the right child has its own right child (to prevent too many swaps). In such a situation, it is possible to join the two strips starting at the ”wide” node, saving additional vertices. In Figure 3.9 (c), the root node is ”wide”, thus the strip can be extended to both directions, without swaps. Still, no additional data structures are needed.

The TIN algorithm is designed for static triangulated irregular networks. The main advantage of this algorithm is that it is quite fast and it does not require to store any additional information. On the other side it produces more vertices and much more strips than the *SGI* method.

As far as I know, the authors did not investigate the usage of this algorithm for flyovers of some huge TINs (flight simulators, etc.). In my opinion it can be easily modified to produce a dynamic stripification, by a simple change of the starting triangle.

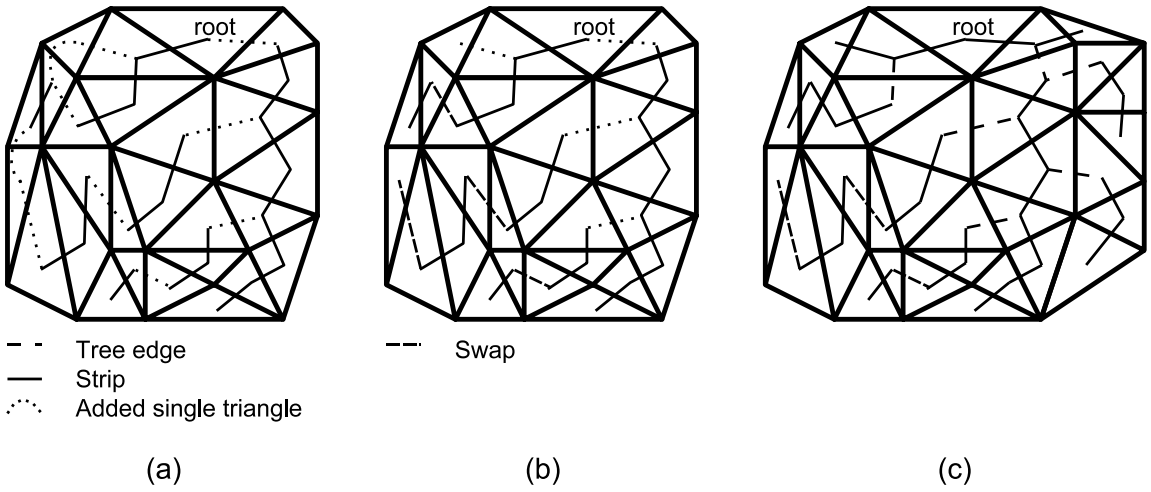


Figure 3.9: The basic traversal strictly maintains the left-right turn sequences. Some singleton strips can be added to existing strips even if there are not connected in the spanning tree (dotted line) (a). Relaxing the left-right turn criterion (allowing swaps – dense dashed line), the number of strips can be lowered (b). The strips can be extended in ”wide” nodes (typically the root node) (c). (From [30])

3.2.3 Tunneling for Triangle Strips in Continuous Level-of-Detail Meshes

Stewart [32] has developed a global algorithm for fully triangulated static and continuous level-of-detail meshes. The method is based on a graph operator called ”tunneling”.

In the dual graph of the stripified triangulation, there can be found two kinds of graph edges (in Figure 3.10 (a), the stripification containing three strips is shown). *Strip edges* join nodes whose corresponding triangles are adjacent in the same strip. All other edges of the dual graph are *nonstrip edges*. A *tunnel* in the dual graph is an alternating sequence of strip and nonstrip edges that starts and ends with nonstrip edges and connects extremities of two strips (Figure 3.10 (b), the tunnel is shown in gray). By complementing the status of each edge in the tunnel, i.e., changing the strip edges to nonstrip and vice versa, the number of strips can be reduced by one (Figure 3.10 (c)).

The algorithm starts by choosing some extreme node of a strip. By breadth-first search the shortest tunnel in the graph is found and the status of the edges of this tunnel is changed. The algorithm is repeated as long as a tunnel can be found and the number of strips reached the local minimum. As the stripification problem is NP-hard, it is not easy to say whether it also reached the global minimum. The number of the strips in the final stripification depends on the order in which the nodes are processed.

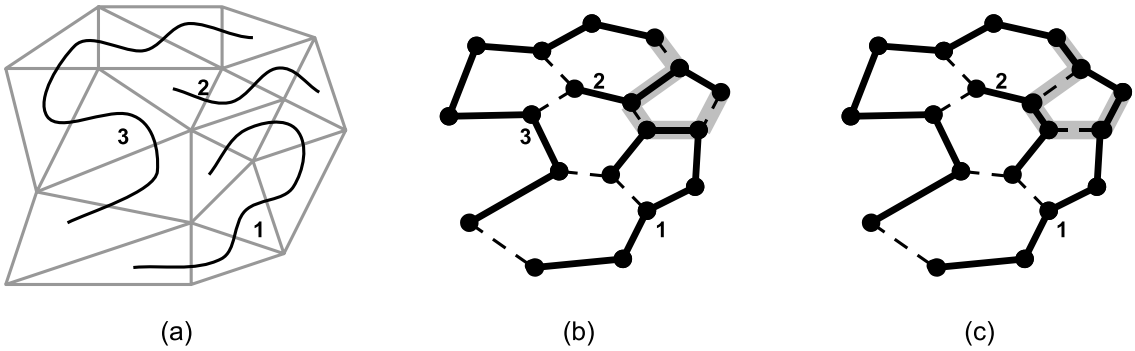


Figure 3.10: A triangle mesh with three strips (a). The tunnel (gray) is an alternating sequence of strip (solid) and nonstrip (dashed) edges (b). Complementing the status of edges in the tunnel, the number of strips is reduced by one (c). (From [32])

There are two limitations for the breadth-first search to produce a valid tunnel. First, the last edge of the tunnel cannot connect two nodes belonging to the same strip (Figure 3.11 (a)). Complementing the status of the edges in such a tunnel does not decrease the number of strips and causes an infinite strip (Figure 3.11 (b)).

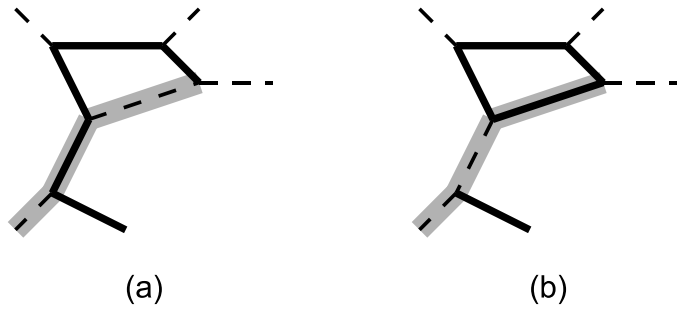


Figure 3.11: The last edge of the tunnel cannot connect two nodes of the same strip (a). In such a situation an infinite strip appears (b). (From [32])

Second, if a nonstrip edge in the tunnel connects two nodes of the same strip, the direction toward the end of strip of the adjacent edges in the tunnel has to be opposite (Figure 3.12 (a)). Again, if the second condition is not fulfilled (Figure 3.12 (b)), the number of strips is not reduced and an infinite strip appears (Figure 3.12 (c)).

To be able to check these conditions, each node has to contain an identifier of the parent strip. When the strip is changed by the tunneling operator, all these identifiers have to be updated, which requires a complete traversal of all affected strips (in the worst case). Such an operation is quite time consuming.

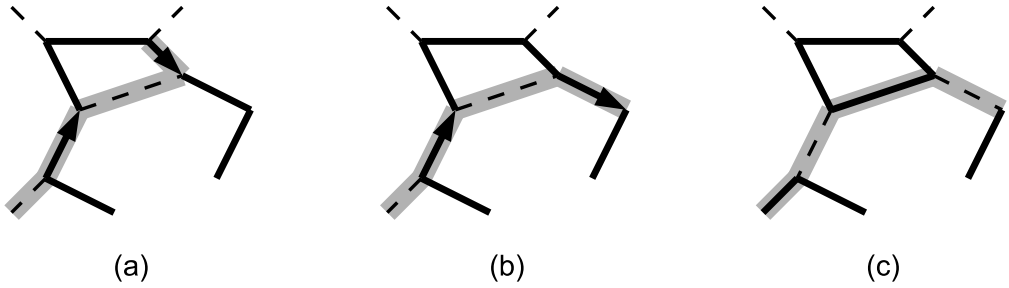


Figure 3.12: If a nonstrip edge connects two strip edges of the same strip, the orientation of those edges toward the end of strip has to be opposite (a). If this condition is not fulfilled (b) an infinite loop appears (c). (From [32])

The tunneling algorithm can be used in several ways. For static meshes the algorithm can simply start from the triangulation. Applying the tunneling algorithm repeatedly, the stripification is obtained. As the tunneling is quite slow, it is also possible to use some other algorithm to obtain an initial stripification and improve it by the tunneling.

The tunneling algorithm can be also used to repair the stripification in CLOD or view-dependent progressive meshes during the simplification process. After an edge split or vertex collapse, the tunneling operator is used, starting only from the triangles from the neighborhood of the split or collapse. Such an approach is called a *local repair*. It is also possible to maintain a list of all endpoints of the strips. With each collapse or split, several triangles from the list are used as starting points for the tunneling. As the number of the chosen starting points is small, the computational overload is negligible. This *global* approach is much better than the local approach, as it maintains the stripification of nearly a constant number of strips.

The tunneling algorithm can handle only fully triangulated meshes. It can be used for both the static triangulation and for the CLOD representation. As far as I know, this algorithm produces the stripification with the lowest number of strips. On the other side, the number of vertices is much higher than the number of vertices produced by the *SGI*. The main disadvantage of the algorithm when used for static meshes is the time complexity that is substantially higher than the *SGI* (while the *SGI* stripification takes several seconds, the tunneling takes several minutes). When using global repairs for the CLOD meshes, the tunneling maintains a good stripification of about a constant number of strips.

3.2.4 Triangle Strips Guided by Simplification Criterion

Belmonte [5] designed an algorithm for stripification of triangle meshes that is guided by a simplification criterion (a minimal quadratic error associated with the contraction of an edge [17]). Triangle strips created with respect to this criterion are preserved as the model is being simplified.

The algorithm uses the edge collapsing for simplification of the model. While collapsing an edge, some error occurs. The calculation of the error is based on the sum of square distances of vertices to an average plane. The algorithm calculates these errors and associates them with corresponding edges. These errors also determine the weight of the edge in the dual graph of the mesh.

Then an algorithm for searching for the maximum spanning tree is applied on the dual graph. Triangle strips are created by simple traversing of this spanning tree. As the tree does not contain the edges with a small error (i.e., edges that are going to be collapsed earlier), the triangle strips are conserved during the simplification process. To preserve the strips, it is necessary to construct strips that do not cross edges with low error. In Figure 3.13 (a), a full resolution triangulation and its stripification is shown. During the simplification step, both strips are preserved (Figure 3.13 (b)).

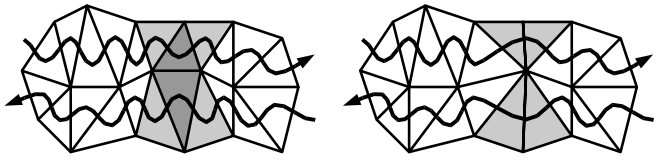


Figure 3.13: A triangulation and its stripification (a). In the case of a good stripification, the strips are preserved during the simplification process (b). (From [5])

The algorithm is designed for progressive triangle meshes. The main goal of the algorithm is to not split the strips during the simplification process. It produces a higher number of strips and higher number of vertices then *SGI* method. On the other side, the number of strips is preserved even if approximately 45% of the edges is collapsed.

The idea of this algorithm could be extended to other duality based stripification methods to produce a stripification that is more or less preserved during the simplification.

3.3 Miscellaneous Approaches

In this subsection methods which produce the stripification from more general or more specific type of data (i.e., point cloud, subdivision surfaces) or which somehow improve the stripification will be introduced.

3.3.1 Hamiltonian Triangulation

Arkin et al. [3] introduced two methods that construct a triangulation of a point set that can be covered with a single strip. The dual graph of this triangulation contains a path that connects all nodes and visits each node exactly once (a Hamiltonian path), thus it is called a Hamiltonian triangulation.

The first method is a simple *insertion* algorithm which produces a triangulation with a Hamiltonian cycle. It requires the points to be in general position (i.e., no three collinear). The second method is more complex, but it does not have the general position limitations.

The *insertion* method starts with a convex hull of a set of points S and a point $v \in S$ that is interior to $conv(S)$. By adding chords from v to each vertex of $conv(S)$ an initial triangulation that contains a Hamiltonian cycle is created (Figure 3.14 (a)). Now all remaining points of S can be added to the triangulation in an arbitrary order. As no three nodes are collinear, an inserted point lies in the interior of some triangle of the current triangulation. The corresponding triangle is split into three new triangles by adding edges from the inserted point to the three vertices of the triangle. The existing triangle strip is not destroyed by this operation, as the three new triangles can always be connected with respect to the entering and exiting edge (Figure 3.14 (b)). The example of the final triangulation is shown on Figure 3.14 (c).

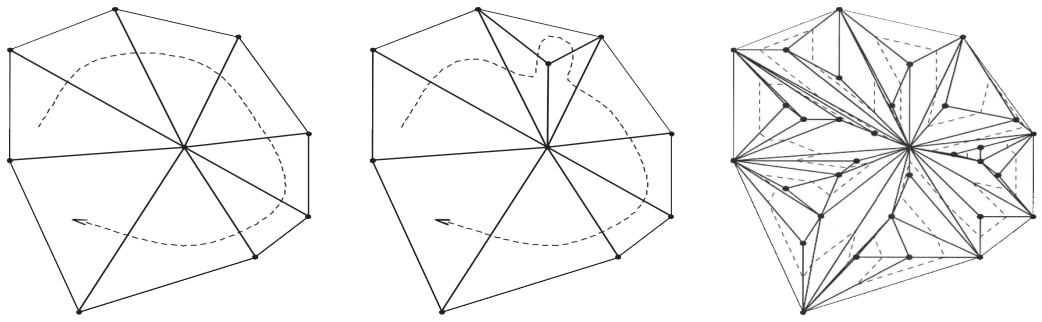


Figure 3.14: The initial triangulation contains the convex hull and one inner point (a). While inserting the remaining vertices, the strip can be preserved (b). The final triangulation contains only one strip (c). (From [3])

The second method – an *onion method* – does not require the points to be in general position. The algorithm computes the convex hull and the convex hull of the remaining points (the points that do not participate in the first convex hull). Then the annulus region bounded by those two convex hulls is triangulated (with $O(n)$ complexity) and a strip connecting all the triangles is created (Figure 3.15 (a)). The algorithm continues following the same scheme for all points from the set. The strips of each annulus are then connected into one triangle strip.

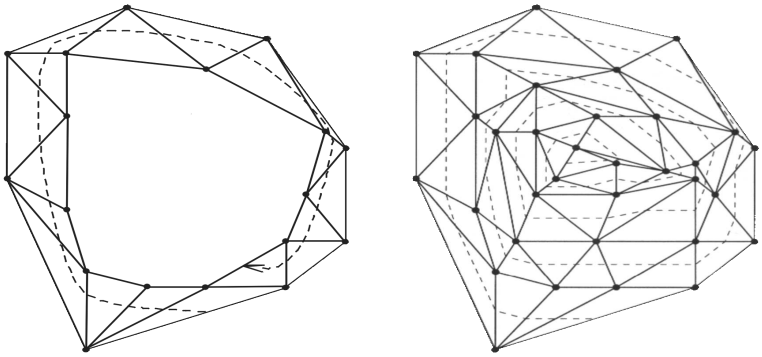


Figure 3.15: The annulus bounded by the first and the second convex hull is triangulated and a strip connecting all triangles is created (a). Following the same construction scheme, the whole triangulation can be created (b). (From [3])

Both these methods can be used only for a 2,5D set of points. Both algorithms have $O(n \log n)$ complexity (as they computes the triangulation whose lower bound complexity is $O(n \log n)$). The insertion method produces a large number of narrow triangles, thus such a triangulation is not suitable for geometry computation. There are also some troubles with rendering of such triangles. The onion method usually produces better triangulation then the insertion method. Still, the triangles are not ideal for further computation.

It would be interesting to try to improve the quality of the triangulation by repairing the ”bad” triangles (to check the quality of triangles, the Delaunay condition can be used). This could be done by swapping the diagonal edge in a quadrilateral consisting of two triangles. In some cases, this swap breaks the triangle strip. The reparation process can stop if a concrete number of strips is achieved, or if the number of bad triangles is lower than some required amount.

3.3.2 Hierarchical Generalized Triangle Strips

Velho et al. [36] introduced a refinement method for computing a triangle sequences of a mesh. This method is applied to construct a triangulation and a single strip that covers a parametric or implicit surface. Furthermore, a hierarchy of triangle strips defined at each refinement level can be obtained.

As the todays graphics hardware usually works on triangles, triangle meshes are often used to approximate smooth surfaces. To be able to render the mesh in appropriate level of detail the mesh refinement is used. Generally, refinement algorithms produce detailed models from a coarse base-mesh by subdividing the original faces. The face is subdivided according to a template, called a *subdivision scheme*.

This algorithm produces a *refinable triangle sequence*, i.e., a triangle sequence whose order can be preserved when its element is subdivided. Such a property depends exclusively on the subdivision scheme. The algorithm has two parts:

1. *Initialization* that creates the base-mesh and the corresponding initial triangle strip.
2. *Refinement* that refine the base-mesh preserving the only triangle strip.

For the refinement it is possible to use both – the uniform or non-uniform subdivision scheme. The uniform scheme recursively subdivides all triangles of a mesh, using the same template, until a desired resolution is obtained. Usually the uniform scheme splits all three edges at the edge midpoint and subdivide the triangle into four sub-triangles. There exists two possible templates – isotropic that subdivide triangle into four identical triangles (Figure 3.16 (a)) and anisotropic (Figure 3.16 (b)). It is obvious that only the use of an anisotropic template produces the triangle sequence (Figure 3.16 (c,d); the shown solution is not the only possible, there exist more configurations in the anisotropic template).

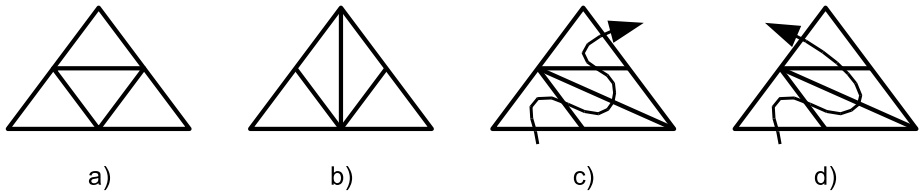


Figure 3.16: Isotropic subdivision template (a). Anisotropic subdivision template (b). Triangle sequences in anisotropic template (c,d). (From [36])

The non-uniform (adaptive) refinement schemes can subdivide only a part of the mesh. They also split the triangle edges into two parts, but not all three edges have to be sub-

divided, therefore at least three different templates must exists. Figure 3.17 shows the subdivision templates: (a) one edge split; (b) two edge split; and (c) three edge split (this template is the same as in the uniform subdivision). For nearly all situations it is possible to find a corresponding triangle sequence that respects the entry and exit edge. In the situation when a non-split exit edge is adjacent to an entry sub-edge, it is not possible to find the sequence, thus a new (so-called Steiner) point s is inserted to the template (Figure 3.17 (d,e)).

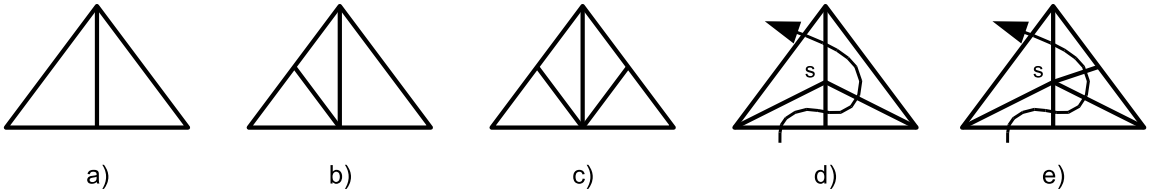


Figure 3.17: For non-uniform subdivision, three templates splitting one (a) two (b) and three (c) edges exists. In two situations it is not possible to produce a triangle sequence, thus the subdivision template has to be modified by inserting a Steiner point s (d,e). (From [36])

The algorithm produces a single triangle strip hierarchy for both uniform and adaptive subdivision schemes. As it uses predefined templates, it is fast and can be used for real-time visualization of progressive meshes, such as NURBS or subdivision surfaces. To manage to get only one strip for the whole mesh, a huge number of swaps is necessary, thus the number of vertices in the final stripification is high.

3.3.3 Skip Strip

An algorithm that efficiently maintains triangle strips during view-dependent simplification was introduced by El-Sana [13]. It is based on hierarchical skip-list-like data structure [27] and it is possible to use it in combination with any stripification algorithm.

To store the view-dependent hierarchy of the mesh, a structure called merge tree [37] is used. The merge tree is constructed in a bottom-up fashion from a high-detail mesh to a low-detail mesh by storing the edge collapsing operations in a hierarchical structure. To build a level of the tree, the maximal set of edge collapses in shortest-edge-first order and with the constraint of no overlapping area is selected. The remaining vertices are promoted to the next level of the tree. The no-overlapping criterion allows to display various details depending upon view-dependent parameters such as light or polygon orientation. As this tree does not change during visualization, it is generated in a preprocessing stage.

As the stripification process does not depend on the merge tree or on the skip list, any stripification algorithm can be used. The stripification is created for the highest resolution model only.

A skip strip is an array of skip strip nodes, where each node contains vertex information (e.g., coordinates, color, etc.), a list of child pointers and a parent pointer. The skip strip node is allocated for each merge tree node (i.e., for each vertex of the base-mesh). The parent pointer and all child pointers are set to copy the merge tree hierarchy. In Figure 3.18 (a), an example of a merge tree is shown. The levels of the tree corresponds to the levels-of-detail of a model. In the figure, the most detailed model consists of four vertices. When simplifying the model, the vertex 2 collapses to the vertex 1 and the the vertex 4 collapses to the vertex 3. The lowest LOD model is represented by a single vertex (the vertex 1). In Figure 3.18 (b) a corresponding skip list structure is presented.

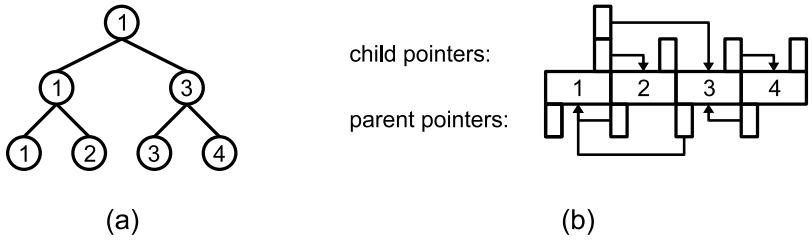


Figure 3.18: An example of a merge tree (a) and its corresponding skip list structure (b). (From [13])

In Figure 3.19 (a) a sample triangle mesh is shown. During the visualization some edges may collapse (Figure 3.19 (b)). The triangle strips can be preserved by replacing the invalid vertices with valid vertices by looking into a skip strip structure. In the figure, the original strip a is defined by a vertex sequence 7,6,4,5,3,2,1. As the vertex 6 collapses to vertex 5 in the lower LOD, the strip is displayed as a sequence 7,**5**,4,5,3,2,1. One can see that while replacing the invalid vertices a group of identical vertices may appear (see the end of the strip b in the sample figure – the strip is displayed as a vertex sequence 1,10,3,9,4,**7,7**). To improve the stripification, the algorithm contains a simple triangle strip scanner that detects and replaces these repeating sequences.

The algorithm does not produce a stripification, but it is designed to maintain the stripification in CLOD meshes. As the skip strip structure is general, it can be used in combination with any stripification method. The speedup while using the skip-strip representation is 30–95% in comparison to triangle representation. For higher simplification (i.e., more decimated meshes), the speedup is lower since the fragmentation of the strip increases.

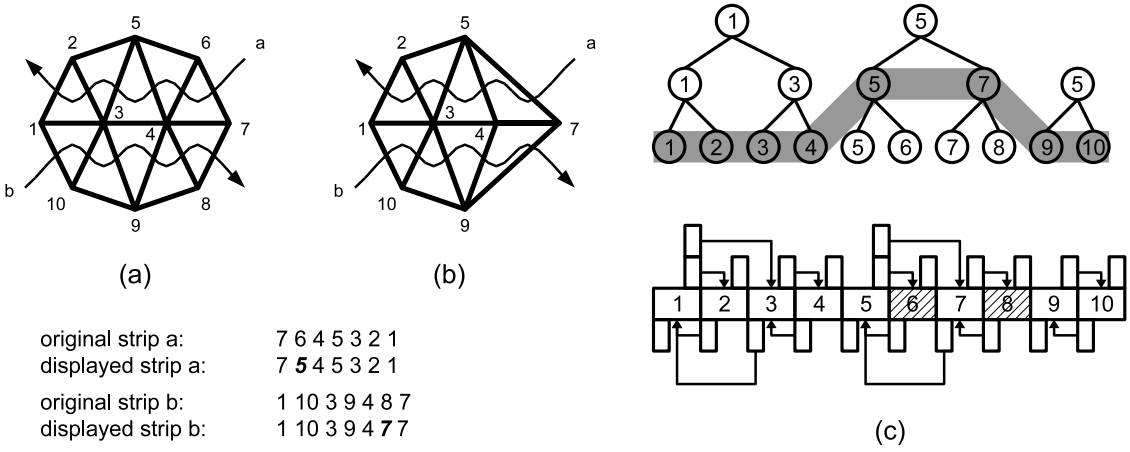


Figure 3.19: An original mesh covered by two strips (a), A decimated mesh (b) and corresponding merge tree and skip list (c). The dark area in the merge tree shows the active LOD for the mesh. (From [13])

3.3.4 Transparent Vertex Caching

Deering [11] proposes the use of a vertex cache of more than two vertices to decrease the amount of vertex transfer from CPU to graphics engine. The idea is to reuse those vertices that are currently buffered in the vertex cache. One year later Bar-Yehuda [4] studied the impact of the buffer size to rendering time (time/space trade off). He has shown that a buffer of size $13.35\sqrt{n}$ is sufficient to render any polygon mesh defined on n vertices in the minimum time $O(n)$.

Hoppe [20] presented an algorithm that optimizes triangle strips for a system of a given memory and transparently reduces the geometry bandwidth. Algorithm is based on a lookahead simulation of the vertex-cache behavior.

The basic strategy of the algorithm is to incrementally grow a triangle strip and to decide at each step whether it is better to add the new triangle to the strip or to start a new strip. To make this decision, a lookahead simulation of the vertex-cache behavior is performed. At the beginning the algorithm marks all triangles of the mesh as unvisited. As a starting triangle for a new strip, it chooses a triangle with the fewest neighbors. If there is only one unvisited neighboring triangle, it is connected to the strip. If there are two faces, the algorithm always continues the strip in a counter-clockwise direction, but it pushes the other neighbor into a queue of possible locations for strip restarts. If there are no unvisited neighboring triangle, the strip cannot continue and a new strip has to be created. As a starting triangle a triangle from the queue is chosen. If there is no triangle

in the queue, the algorithm chooses a triangle whose vertices are already in the cache and which has the fewest number of unvisited neighbors.

As the capacity of the vertex cache is limited, it can happen that the strip overflows it and it will not be possible to re-use the cached vertices. Therefore, a lookahead simulation of the vertex cache is performed, before adding a new face. The simulation performs s simulations of the strip-growing process for the next s triangles. It computes the average number of cache misses per visited triangle. If the lowest cost value corresponds to $s = 0$, the strip is restarted, otherwise a new face is added.

The algorithm is designed to maximize the reuse of cached vertices. According to the results, presented in the paper, the average cache miss per vertex rate is about 1.25 for a vertex cache of size 16, whereas the absolute lower bound is 1.

3.4 Overall Comparison

In this section, an overall comparison of most of the mentioned methods is presented. All experiments were performed on a PC INTEL Pentium 4, 2.8GHz, 2GB of RAM, ATI T32, running on MS Windows XP. Naturally, times of I/O operations have been excluded from measurements if possible.

I have chosen a set of ten models that are often used in other publications and that are available on the internet [31, 18, 8, 10]. Models are shown in Appendix B. All these models are fully triangulated. The demi model consists of 37 disconnected components. The dragon and the blade model contains some inconsistencies, thus the number of components is higher.

#	model	# vertices	# triangles	# components
1	cow	2905	5804	1
2	demi	9138	17506	37
3	bunny	35947	69451	1
4	dinosaur	56194	112384	1
5	balljoint	137062	274120	1
6	club	209779	419554	1
7	hand	327323	654666	1
8	dragon	437645	871414	151
9	happy	543652	1087716	1
10	blade	882954	1765388	295

Table 3.1: Set of testing models.

For the comparison, I have chosen the methods that are able to stripify the full 3D meshes[3]. Table 3.2 presents the algorithm overview. The first and second column shows the short name of the algorithm as it was presented in this work and the chapter with algorithm description. As the programming language and the compiler can influence the speed of the program, the third column (”Compiler”) shows the used compiler. A short name under which will be the algorithm presented in tables is shown in the column ”Label”. For all algorithms I have used the default parameters or parameters that were recommended by the author. The concrete parameter is mentioned in the column ”Parameters”. Very often, the algorithm has implemented both the vertices minimizing function and the strips minimizing function.

[3]I would like to thank to prof. Stewart for providing the source code of Tunneling, to X.Xiang and prof. J.S.B. Mitchell for providing the source code of FTSG and to D.Kornmann for support for his program.

Algorithm	Chapter	Compiler	Label	Parameters	Minimizing
SGI	3.1.1	Delphi 7	SGI SGI-LS	-LNLN -LS	strips vertices
Weighted SGI	3.1.2	N/A	WSGI	keys 7,8,9	strips
SStrip	3.1.3	Cygwin, gcc	SSTRIP SSTRIP-Q	-m 2 -m 2 -q	strips vertices
STRIPE	3.1.4	Cygwin, gcc	STRIPE-L STRIPE-Q	-l -q	strips vertices
FTSG	3.2.1	Cygwin, gcc	FTSG-SGI FTSG-ALT	-dfs -concat -sgi -dfs -concat -alt	strips vertices
Tunneling	3.2.3	Cygwin, gcc	TUNNEL		strips

Table 3.2: Algorithms overview

- *SGI(-LS)*: Although the original *tomesh.c* code is available on the internet [1], I was not able to make it work due to lack of documentation. I have implemented and tested several methods based on the *SGI* algorithm in [34]. I have used the standard SGI method (*SGI*) and vertex minimizing heuristic (*SGI-LS*) for the tests.

- *WSGI*: There is only a binary version of the algorithm available on the internet [23]. As the algorithm output depends on some randomized decisions, ten measurements were taken and the average value is presented. All three heuristics were enabled for the measurement. The program only visualizes the result but it does not export the stripified mesh, thus I was not able to make some tests (FPS, memory, time).

- *SSTRIP(-Q)*: The source code of the program is available on the internet [28] under the GNU General Public License. The number of the simultaneous strips was set to two. The measurement for both heuristic (strips minimizing, vertices minimizing (-q)) was performed.

- *STRIPE(-L/-Q)*: The source code of the program is freely available on the internet [14] for non commercial use. The mesh is exported during the stripification process, thus it is not possible to exclude the time of I/O operation. The tests are performed with two heuristic functions: ”Look ahead one level in choosing the next polygon” (-l) and ”Choose the polygon which does not produce a swap” (-q). The tests are performed with STRIPE version 2, which is much faster.

- *FTSG(-SGI/-ALT)*: The program is free for non-commercial purposes only and it can be obtained via e-mail [38]. The tests were performed with the depth-first search

heuristic (-dfs) and enabled concatenation of strips (-concat). The next triangle decision was based on the *SGI* criterion (-sgi – strips minimizing) and on alternating the left-right turns (-alt – vertices minimizing).

- *TUNNEL*: The program is not available on the internet, but it can be obtained via e-mail. The tests were performed with the default settings. The memory usage and computation time is very high!

3.4.1 Vertices

In the Table 3.3 (Figure 3.20 (top)[4]), a comparison of number of vertices in strips is presented. The number of vertices determines the size of data needed for the model – i.e. the amount of data sent to the rendering engine. The difference in the number of vertices does not vary too much for different algorithms, because there are two theoretical boundaries. The number of vertices could not be lower than $number\ of\ triangles + 2$ (for a sequential strip, covering the whole triangulation, which is quite impossible for a real-life model) and it could not be higher than $3 \cdot number\ of\ triangles$ for a set of isolated triangles or $2 \cdot number\ of\ triangles$ for a connected set of triangles. The Table 3.4 (Figure 3.20 (bottom)) shows a comparison of vertices per triangle, i.e., the ratio of number of vertices to the number of triangles.

The vertices minimizing algorithms (*STRIPE-Q*, *SSTRIP-Q* and *FSTG-ALT*) produces nearly the same number of vertices. The average V/T for these algorithms is about 1.25. The SGI-LS algorithm produces stripifications with the lowest number of vertices (1.23 V/T in average). As this algorithm strictly chooses the triangles which do not cause a swap, the low V/T is compensate by a huge number of strips.

The *EVANS-L* algorithm produces a stripification with an average V/T about 1.47. In my opinion there is some bug in the code, as this algorithm produces a high number of vertices and also a high number of strips (although it should minimize the number of strips).

It is quite interesting that nearly all algorithms (except *TUNNEL* and *STRIPE-L*) have the same behavior. For the bunny model (which is topologically very simple), the average V/T is very low, on the other side, the average V/T for the dragon and for the happy buddha is more than 10% higher. Similar behavior is also noticeable in the average length of strips (3.21 (bottom)).

[4]The left graph shows the dependency on number of triangles. As the stripification process does not depend only on the number of triangles but also on the topology of the model, the right graph shows the dependency on concrete model.

3.4.2 Strips

Number of strips produced by tested algorithms are presented in the Table 3.5 (3.21 (top)). The number of strips as well as the number of vertices is crucial for the rendering speed. As starting a new strip takes some extra time, a huge number of triangle strips slows down the rendering. On the other side, minimization of the number of strips often leads to higher number of vertices (swaps). For better comparison, the average length of triangle strips is presented in the Table 3.6 (3.21 (bottom)).

The *TUNNEL*ing algorithm produces more than three times lower number of triangle strips than all other algorithms. On the other side, to obtain such a long triangle strips, it is necessary to use swaps (thus increase the number of vertices).

The differences in the number of strips are very high. The *SGI-LS* algorithm produces stripification with more than 20 times higher number of strips than the *TUNNEL*ing.

3.4.3 Rendering Speed

As the triangle strips are mainly used to speed up the visualization, I have also tested the rendering speed of models stripified by different techniques (Table 3.7, Figure 3.22 (top)). To maximally use the graphics hardware, the OpenGL vertex buffer objects are used [26]. Note that the rendering speed depends on the GPU architecture and can vary for other graphics cards.

According to the tests, the speed of rendering depends on the number of vertices (as these vertices has to be transmitted) and on the number of strips (as the creation of a new strips cost some additional time). For this reason, the fastest rendering is neither achieved by the *TUNNEL*ing algorithm (that produces the lowest number of strips) nor by the SGI-LS (that produces the lowest number of vertices). The best rendering performance was achieved with models produced by *STRIPE-Q* and *FSTG-SGI*. The differences in FPS are less than 20% in the worst case but less than 10% in average.

In Figure 3.22 (bottom) a rendering time of a single frame is shown. The rendering speed more or less linearly depends on the number of vertices of the model. Unfortunately, it is not possible to find out the dependency of the rendering speed on the number of vertices in strip – Figure 3.23 (top,left); or on the number of strips – Figure 3.23 (top,right) (both figures shows the rendering speed for three models stripified by all possible methods).

As the *WSGI* program does not produce any output file, it was not possible to measure the rendering speed.

3.4.4 Execution Time

The time of stripification process is actually not very crucial, as the stripification is usually used in a preprocessing stage. The execution time presented in Table 3.8 (Figure 3.24) does not include the I/O operation (except the *STRIPE* algorithm, as the output operation runs during the stripification process).

All algorithms except *STRIPE* and *TUNNEL* produce the stripification in about the same time. *STRIPE* is slower as the saving process is included in the measurement. As *TUNNEL*ing searches for a tunnel with a breadth first search method from each strip endpoint, the complexity of the algorithm is higher than $O(n)$ and the execution time is not comparable to other algorithms.

The SGI-LS algorithm is the fastest one, as it uses a very simple criterion and it does not make the lookahead search.

The execution time of *SSTRIP* algorithms is not published for all models, as the algorithm did not worked well on the Windows platform[5]. Although the *WSGI* program shows the time needed for stripification, it is not included in the table, as it does not show the time for temporary structures such as adjacency tables, etc.

3.4.5 Memory Usage

As different algorithms use different data structures, the amount of allocated memory can differ (Table 3.9, Figure 3.23 (bottom)). To measure the memory usage, a program that scans the running processes (using win32 API CreateToolhelp32Snapshot function) and stores the memory usage peak for a process is used. As the scanning is not continuous, some inaccuracy may appear.

The *TUNNEL*ing is the most memory consuming stripification program of the tested programs. This is not very surprising as the algorithm needs a special data structure to maintain the information about the tunnels. The memory usage of *STRIPE* is also very high, but I do not know the reason. As far as I know, it does not need any special structures (it works on the same principle as the SGI algorithm) furthermore, the strips are being saved during the stripification process.

As the *WSGI* program provides also the visualization of the model (thus it needs some additional memory), I did not include the memory usage.

[5]For high resolution models, the program crashed

3.4.6 Conclusion

According to all these tests, the algorithm presented by Xiang et al. (*FTSG-SGI*), provides a stripification that is most suitable for the rendering. Furthermore, this algorithm is very fast and it does not need too much memory. Following the rendering speed criterion the *STRIPE-Q* algorithm produces a stripification of the same quality.

Although *SSTRIP-Q* and *SGI-LS* (vertex minimizing heuristic) produces the lowest number of vertices, the rendering speed is lower. Optimizing only the number of strips (*TUNNEL*ing) leads to lower rendering speed, too.

model	SGI	SGI-LS	WSGI	SSTRIP	SSTRIP-Q	STRIPE-L	STRIPE-Q	FSTG-SGI	FSTG-ALT	TUNNEL
cow	7618	7087	7681	7585	7079	8568	7190	7400	7186	8295
demi	23441	21838	23457	22802	21861	25443	21939	22631	22169	24255
bunny	86982	81730	87034	85909	81890	101806	82760	85362	82213	98503
dinosaur	148437	138857	149516	147477	139984	164430	141267	144788	140152	159361
balljoint	358070	337585	360172	355481	340518	400951	343022	345862	339738	385909
club	532253	505036	536496	527017	507086	613411	511751	521652	508143	580683
hand	875690	812042	884921	866341	817614	965939	825434	855683	824224	938693
dragon	1237019	1129993	1251062	1217292	1141854	1289540	1153301	1195550	1156027	1260651
happy	1545562	1409575	1563465	1519534	1424174	1609065	1438639	1492285	1442542	1574172
blade	2293726	2134682	2313250	2265291	2142271	2609281	2159526	2248981	2166404	2542184

Table 3.3: Number of vertices in strips.

model	SGI	SGI-LS	WSGI	SSTRIP	SSTRIP-Q	STRIPE-L	STRIPE-Q	FSTG-SGI	FSTG-ALT	TUNNEL
cow	1.31	1.22	1.32	1.31	1.22	1.48	1.24	1.27	1.24	1.43
demi	1.34	1.25	1.34	1.30	1.25	1.45	1.25	1.29	1.27	1.39
bunny	1.25	1.18	1.25	1.24	1.18	1.47	1.19	1.23	1.18	1.42
dinosaur	1.32	1.24	1.33	1.31	1.25	1.46	1.26	1.29	1.25	1.42
balljoint	1.31	1.23	1.31	1.30	1.24	1.46	1.25	1.26	1.24	1.41
club	1.27	1.20	1.28	1.26	1.21	1.46	1.22	1.24	1.21	1.38
hand	1.34	1.24	1.35	1.32	1.25	1.48	1.26	1.31	1.26	1.43
dragon	1.42	1.30	1.44	1.40	1.31	1.48	1.32	1.37	1.33	1.45
happy	1.42	1.30	1.44	1.40	1.31	1.48	1.32	1.37	1.33	1.45
blade	1.30	1.21	1.31	1.28	1.21	1.48	1.22	1.27	1.23	1.44

Table 3.4: Number of vertices per triangle (V/T).

model	SGI	SGI-LS	WSGI	SSTRIP	SSTRIP-Q	STRIPE-L	STRIPE-Q	FSTG-SGI	FSTG-ALT	TUNNEL
cow	98	352	87	78	127	141	136	105	312	19
demi	335	1183	315	293	419	456	418	286	1020	139
bunny	648	3560	622	575	1174	1531	1229	618	3238	166
dinosaur	1177	7276	1355	1271	2422	2470	2498	1346	6411	260
balljoint	2279	17454	2910	2519	5746	6145	5820	2446	15371	536
club	2658	23966	3875	3111	7782	9210	8184	3054	21148	750
hand	8997	44710	9279	8318	14674	15309	15422	10394	38779	1590
dragon	17399	71182	17112	16402	23564	22928	25356	20571	58377	3331
happy	21578	88143	21250	20119	29150	28563	31550	25576	72271	3710
blade	23125	115568	23468	21829	35101	41128	35952	26779	99890	4606

Table 3.5: Number of strips in a model.

model	SGI	SGI-LS	WSGI	SSTRIP	SSTRIP-Q	STRIPE-L	STRIPE-Q	FSTG-SGI	FSTG-ALT	TUNNEL
cow	29.64	8.25	33.58	37.24	22.87	20.60	21.36	27.67	9.31	152.89
demi	27.28	7.72	29.02	31.19	21.81	20.04	21.86	31.95	8.96	65.74
bunny	55.47	10.10	57.84	62.52	30.62	23.48	29.25	58.17	11.10	216.55
dinosaur	47.74	7.72	41.46	44.21	23.20	22.75	22.50	41.75	8.77	216.13
balljoint	60.14	7.85	47.10	54.41	23.85	22.30	23.55	56.04	8.92	255.71
club	78.92	8.75	54.14	67.43	26.96	22.78	25.63	68.69	9.92	279.71
hand	36.38	7.32	35.28	39.35	22.31	21.38	21.22	31.49	8.44	205.86
dragon	25.15	6.15	25.58	26.68	18.57	19.09	17.26	21.27	7.50	131.39
happy	25.19	6.17	25.58	27.02	18.65	19.03	17.23	21.26	7.52	146.54
blade	38.18	7.64	37.62	40.45	25.15	21.47	24.56	32.97	8.84	191.70

Table 3.6: Average length of strips.

model	SGI	SGI-LS	WSGI	SSTRIP	SSTRIP-Q	STRIPE-L	STRIPE-Q	FSTG-SGI	FSTG-ALT	TUNNEL
cow	421.24	411.98		414.77	416.08	416.15	420.17	417.23	407.27	419.88
demi	334.02	332.50		336.53	335.28	322.88	337.27	338.99	327.86	329.57
bunny	137.76	136.43		135.96	133.78	125.40	138.06	136.38	133.50	130.23
dinosaur	107.51	106.71		106.47	108.27	97.87	109.05	108.37	105.25	101.52
balljoint	49.08	48.94		49.55	49.88	44.45	50.58	50.74	49.04	46.34
club	34.38	34.08		34.43	34.51	29.84	34.89	34.74	33.78	31.88
hand	21.86	22.14		22.18	22.23	19.70	22.79	22.25	21.95	20.59
dragon	15.43	16.38		16.09	16.13	14.72	16.26	15.86	16.09	15.28
happy	13.07	12.89		13.58	13.72	12.84	13.51	14.28	13.27	13.78
blade	8.89	8.74		9.05	9.07	8.70	9.09	9.35	8.97	8.39

Table 3.7: The average FPS.

model	SGI	SGI-LS	WSGI	SSTRIP	SSTRIP-Q	STRIPE-L	STRIPE-Q	FSTG-SGI	FSTG-ALT	TUNNEL
cow	0.028	0.016		0.012	0.013	0.258	0.268	0.015	0.016	0.515
demi	0.088	0.031		0.042	0.042	0.849	0.829	0.047	0.062	0.703
bunny	0.387	0.125		0.218	0.217	3.150	3.195	0.250	0.250	101.375
dinosaur	0.645	0.235		0.386	0.389	4.847	4.888	0.422	0.437	44.719
balljoint	1.669	0.672		1.128	1.145	12.055	12.087	1.125	1.109	107.500
club	2.545	0.953		1.710	1.725	18.651	18.695	1.688	1.687	366.076
hand	3.509	1.235				34.340	34.195	2.031	2.578	338.359
dragon	3.863	1.938				37.426	37.342	3.016	3.204	662.406
happy	6.260	2.609				46.359	45.992	3.797	4.063	889.938
blade	9.684	4.125				112.472	112.375	5.735	5.984	3488.190

Table 3.8: The computation time in seconds.

model	SGI	SGI-LS	WSGI	SSTRIP	SSTRIP-Q	STRIPE-L	STRIPE-Q	FSTG-SGI	FSTG-ALT	TUNNEL
cow	1.7	1.6		4.7	4.7	4.2	4.2	2.7	2.3	5.1
demi	2.6	2.4		6.1	6.1	8.4	8.4	4.9	4.9	11.2
bunny	6.4	6.0		12.5	12.5	27.2	27.2	13.6	13.6	37.2
dinosaur	10.4	9.8		17.7	17.7	42.5	42.6	17.5	17.5	57.8
balljoint	23.3	22.2		37.5	37.5	100.7	100.7	38.8	38.8	137.6
club	35.0	33.3		55.3	55.3	152.9	153.0	58.0	58.0	209.3
hand	54.0	51.3				237.4	237.4	118.2	118.2	325.0
dragon	71.6	68.0				315.3	315.4	122.1	122.1	435.1
happy	89.0	84.6				393.0	393.0	190.2	190.2	540.8
blade	143.5	136.3				636.5	636.7	298.8	298.8	873.7

Table 3.9: The amount of allocated memory in MB.

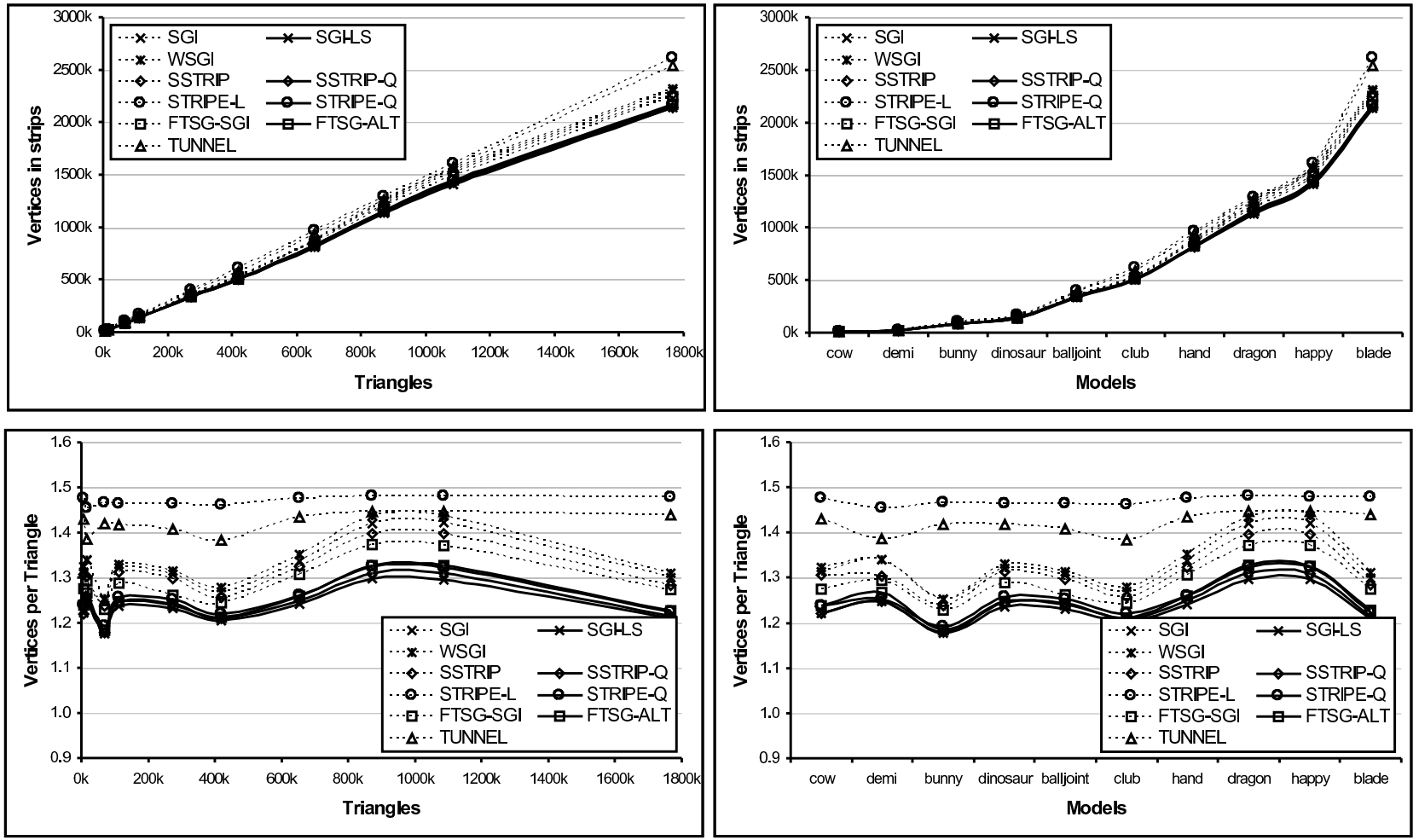


Figure 3.20: Graphs: Number of vertices in strips (top). Number of vertices per triangle (bottom).

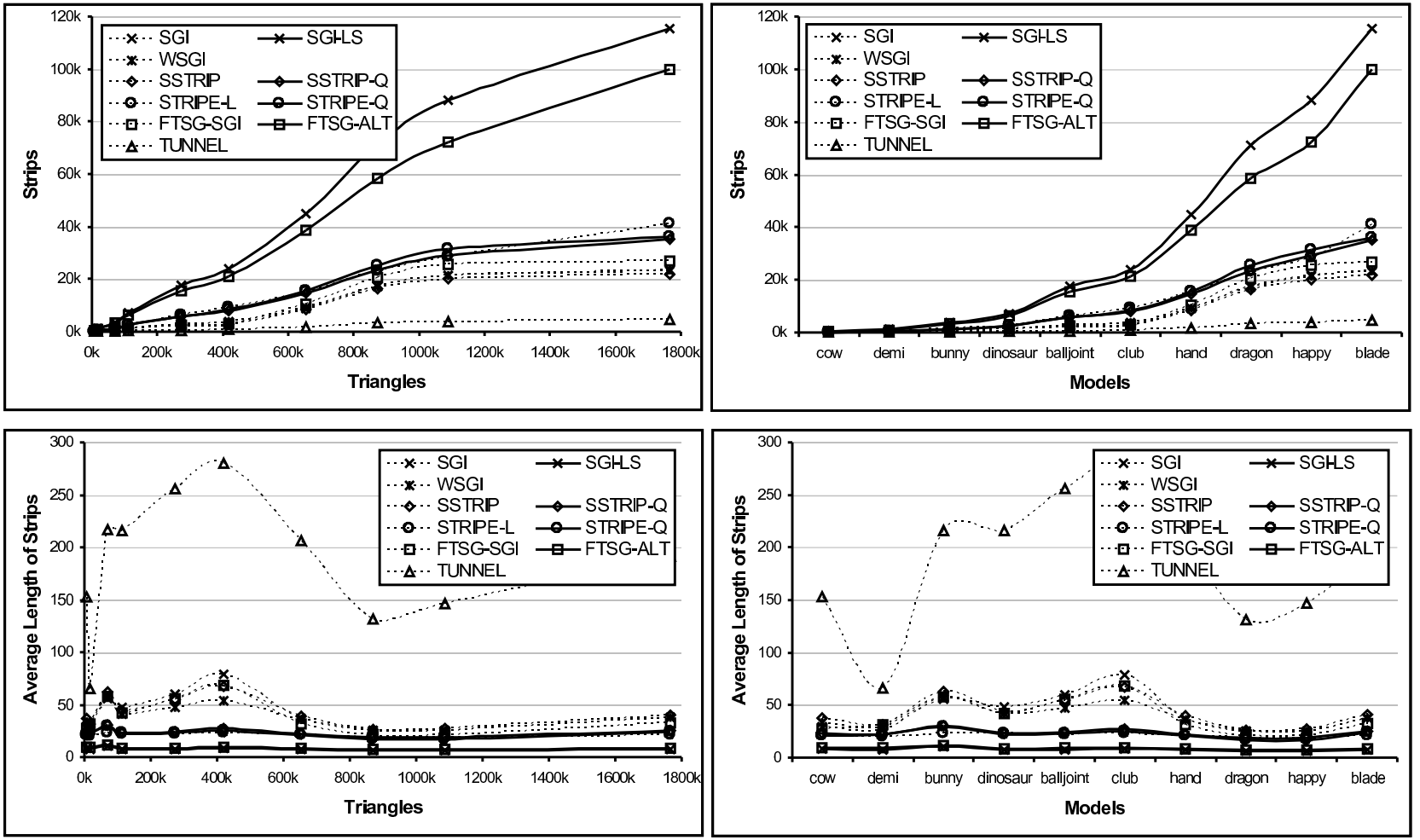


Figure 3.21: Graphs: Number of strips (top). Average length of strips (bottom).

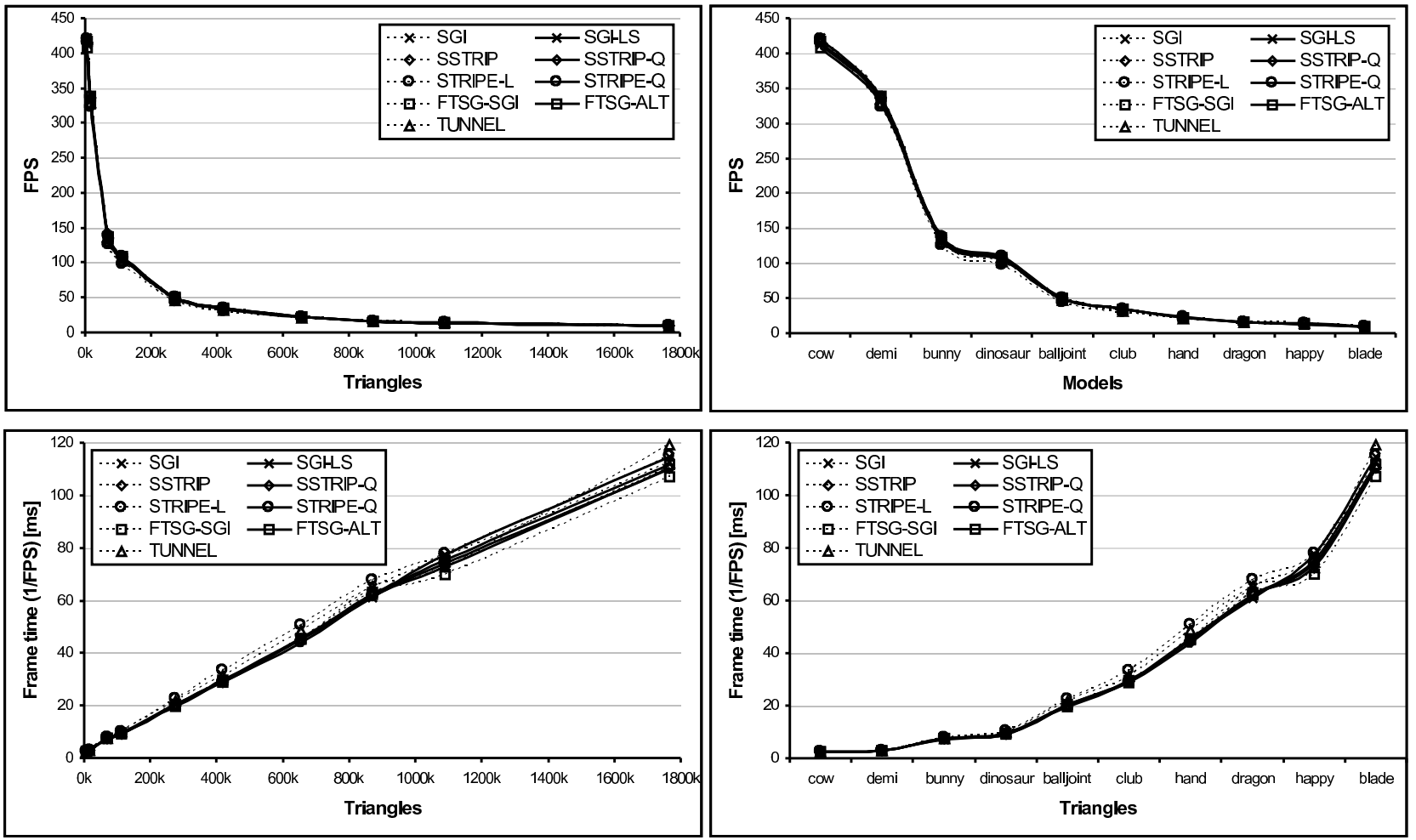


Figure 3.22: Graphs: Average FPS (top). Rendering time for a single frame (bottom).

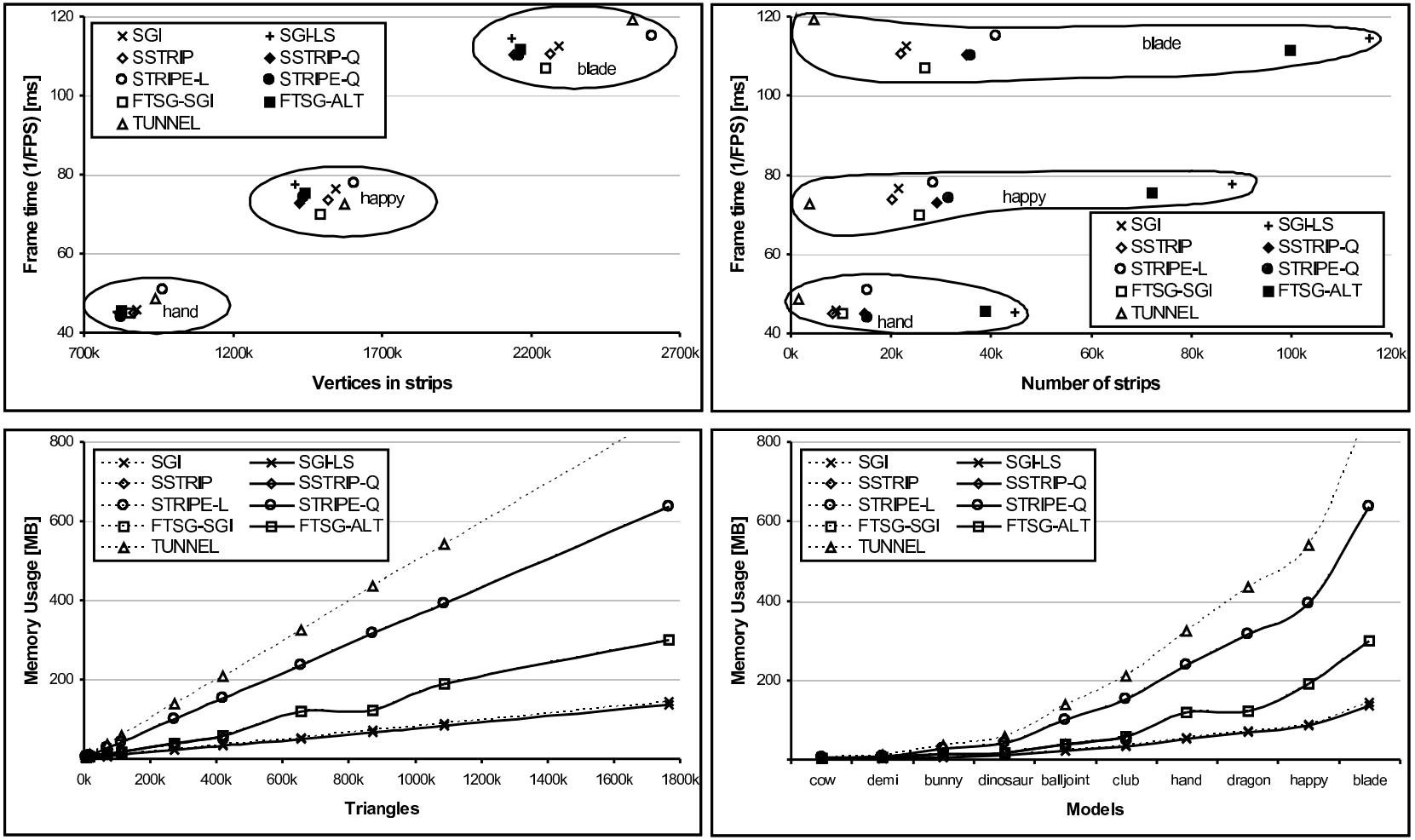


Figure 3.23: Graphs: Rendering time for a single frame (top). Memory usage (bottom).

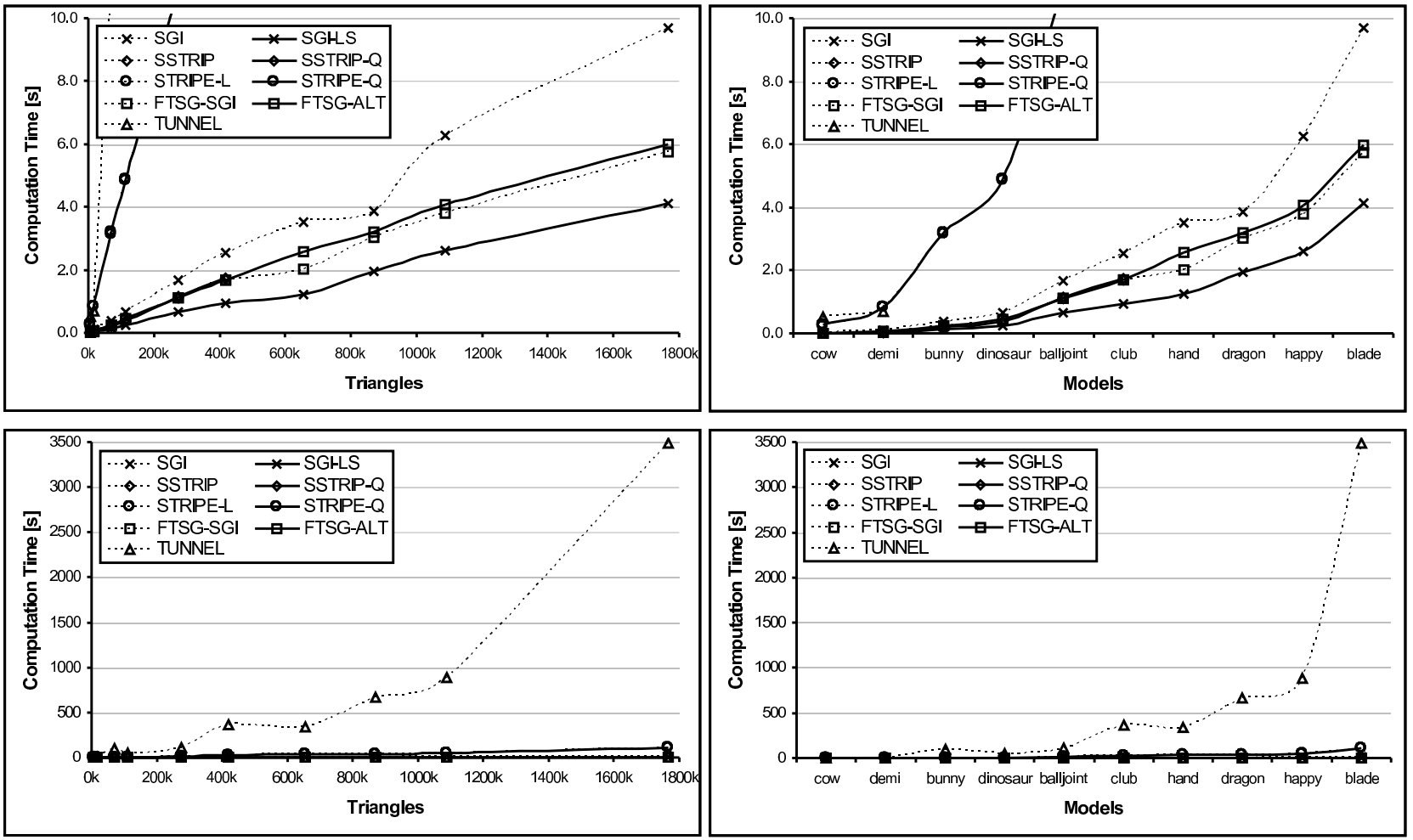


Figure 3.24: Graphs: Execution time.

4 Delaunay Stripification

In this section I will concentrate on 2D and 2.5D triangulations, which are often used for terrain modeling. The terrain models are often given as a point set and it is necessary to make a triangulation of this point set first. One of the most common triangulations is the Delaunay triangulation. This triangulation is very popular especially because of two facts: (1) it produces the most equiangular triangles of all possible methods (it maximizes the minimum angles); (2) it can be computed in $O(n \log n)$ time in the worst case and in $O(n)$ time in the expected case. It is also possible to create several levels of detail while using an incremental insertion algorithm for the Delaunay triangulation.

4.1 Delaunay Triangulation

In this section I will describe the Delaunay triangulation and structures that I use. More details about the Delaunay triangulation are e.g. in [12].

Definition 1 *A triangulation $T(P)$ of a set of points P in the Euclidean plane is a set of edges E such that*

1. *no two edges in E intersect at a point not in P,*
2. *the edges in E divide the convex hull of P into triangles.*

Definition 2 *The triangulation $DT(P)$ of a set of points P in the Euclidean plane is a Delaunay triangulation of P if and only if the circumcircle of any triangle of $DT(P)$ does not contain any other point of P in its interior.*

There exist several approaches of constructing a Delaunay triangulation, e.g.:

- divide & conquer [12],
- incremental insertion [25, 22],
- high-dimensional embedding [7].

Although the fastest method is divide & conquer [12] (according to [33]), I decided to use the incremental insertion for several reasons: divide & conquer methods are often too sensitive to numerical inaccuracy, another reason is that the insertion method allows us to insert points in a specific order (e.g., according to the importance of the point) to obtain different levels of details. Also the implementation of incremental insertion is easier than

the divide and conquer. While using randomized incremental insertion, the algorithm is insensitive to input data configurations. Last but not least – the incremental insertion has been already implemented in our computer graphics group [22].

The incremental insertion algorithm is described in Figure 4.1.

Input: the set of points P in E^2
Output: *DT(P)*

1. Create a temporary triangle, such that all points of P are enclosed in it;
2. For each p from P do
   (a) Find the triangle t or edge e that contains the point p;
   (b) If the p point lies on an edge e, find the triangles sharing this edge and subdivide them into four new triangles
   else subdivide the triangle t into three new triangles;
   (c) If new triangles do not fulfill the Delaunay condition, flip the edges (thus create new triangles) and repeat this step.

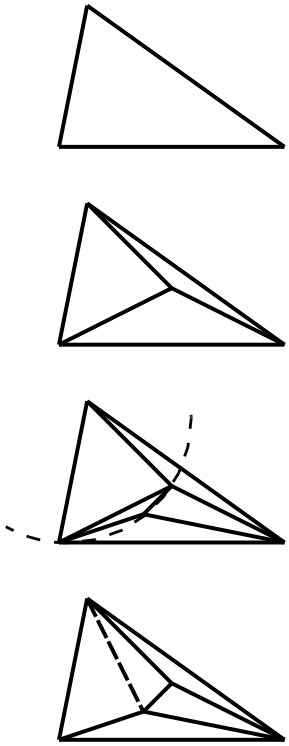


Figure 4.1: Algorithm steps for the incremental insertion of DT and an example of the triangulation construction.

The most time consuming part of the algorithm is step 2a – a quick location of the triangle containing the inserted point. In our approach, triangles are kept in a directed acyclic graph (DAG) – a graph where the history of insertion and flipping is stored.

An example of vertex insertion and edge flipping is shown in Figure 4.2. In the first step, a new vertex is inserted. Then the corresponding triangle is divided into three new triangles (4,5,6). Because the new triangles do not fulfill the Delaunay condition, edge flips are performed in steps three and four.

4.2 Delaunay Stripification

To speed up the visualization of different levels of detail of the triangulation, it is possible to use triangle strips. In Figure 4.2 (d), one can see that it is possible to obtain a stripification

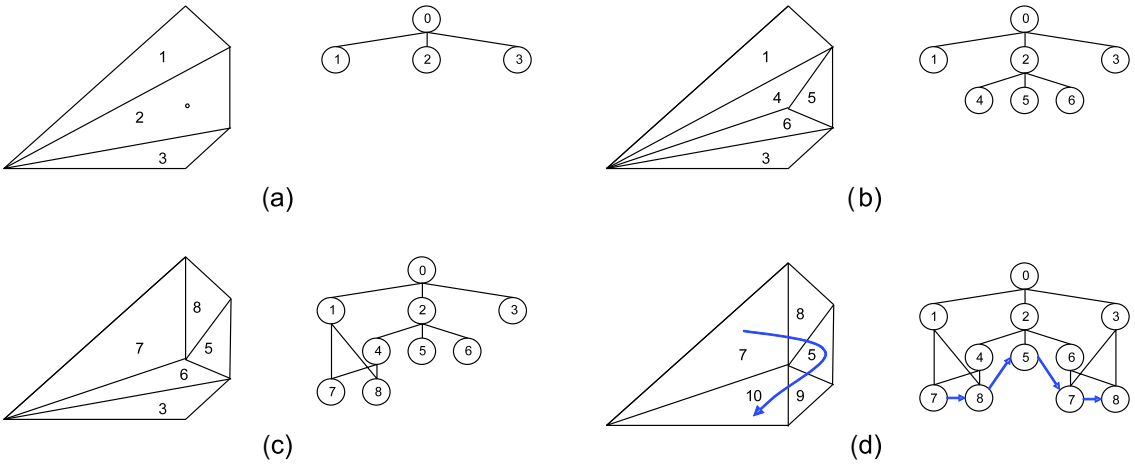


Figure 4.2: Example of DAG. A new point is inserted into a triangulation (a). The corresponding triangle is subdivided into three new triangles (b). The triangles checked for the Delaunay condition (c) and (d).

for each step of the triangulation process by traversing the leaves of the DAG structure very quickly. This algorithm was published in [35].

To improve the quality of stripification, it is necessary to modify the existing algorithm [22] to avoid breaking strips. There are two steps in the algorithm where the strip could be broken: (a) insertion of a new vertex, and (b) flipping edges to fulfill the Delaunay condition.

While inserting a new vertex two situations can appear. If the inserted vertex lies inside a triangle, three new triangles are created. To preserve the strip, we need only to keep the right order of sons in the DAG (see Figure 4.3).

If we don't care about the Delaunay condition (do not perform flips), we obtain a Hamiltonian triangulation (as described in [3] – we get one strip for the whole triangulation, penalized by worse quality of triangles).

In Figure 4.3 (left) an old triangulation with a strip is shown. In the middle, there is a new triangulation and a new triangle strip after a vertex insertion. On the right side, there is the corresponding DAG.

In the other situation the inserted vertex lies on an edge. In such a situation several cases may appear. In the first case, the incoming edge (i.e., the edge on which the strip enters the triangle) of the first triangle and the outgoing edge (i.e., the edge on which the strip leaves the triangle) of the second triangle have a common vertex. It is possible to connect all four new triangles into one strip and continue (see Figure 4.4).

The second case, where the incoming edge of the first triangle do not share any vertex

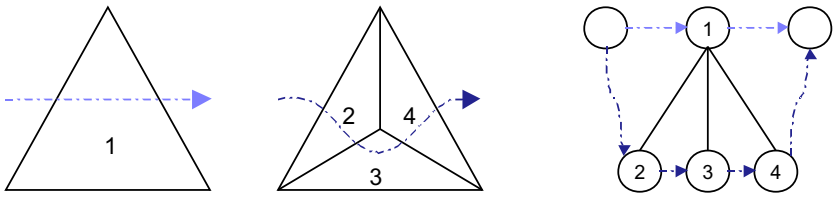


Figure 4.3: Insertion of a vertex into a triangle

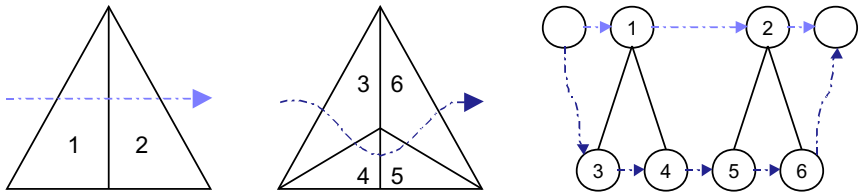


Figure 4.4: Insertion of a vertex on an edge (case 1)

with the outgoing edge of the second triangle, is the most problematic. In this case it is not possible to insert all four new triangles into a strip and a new strip has to be created.

There are two possibilities: (1) Insert three new triangles to the existing strip and create one new single-triangle strip (in Figure 4.5 triangle 4); or (2) to avoid the single-triangle strip it is possible to divide the strip and insert triangles 3 and 4 to the first strip and triangles 5 and 6 to the second strip.

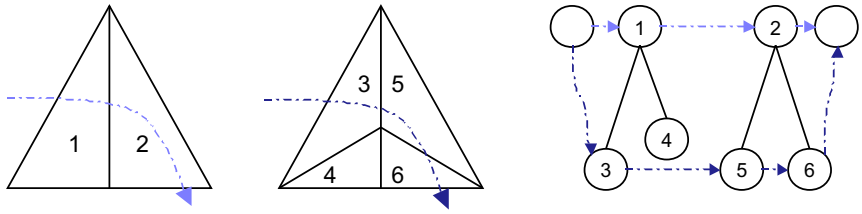


Figure 4.5: Insertion of a vertex on an edge (case 2)

In the last case, the first triangle lies in another strip than the second one. The new triangles are simply inserted into the existing strips (see Figure 4.6).

To make the Delaunay triangulation, each new triangle has to be checked and if it does

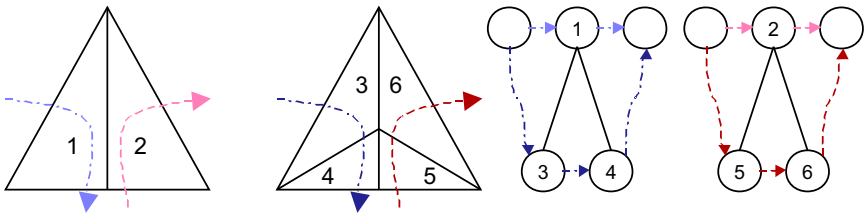


Figure 4.6: Insertion of a vertex on an edge (case 3)

not fulfill the condition, it is necessary to flip the edge. Again, several cases may appear. When the incoming edge of the first triangle does not share a vertex with the outgoing edge of the second triangle, it is possible to connect both new triangles into a strip (Figure 4.7).

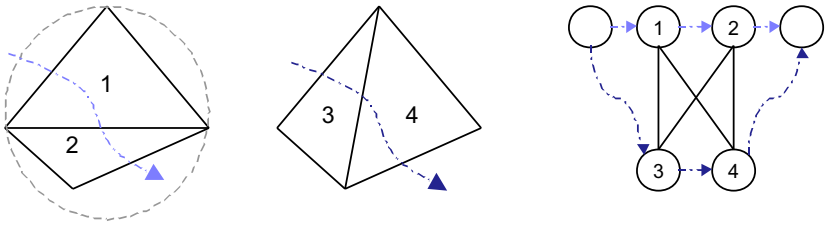


Figure 4.7: Edge flipping (case 1)

If the incoming and outgoing edges share a vertex, a new single-triangle strip has to be created (Figure 4.8).

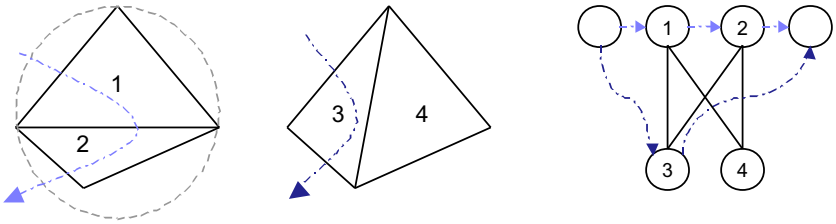


Figure 4.8: Edge flipping (case 2)

If the two flipped triangles lie in the same strip but do not share a common edge in the strip, the existing strip is divided into two strips (Figure 4.9).

In the last case the two triangles do not belong to the same strip. After the edge is

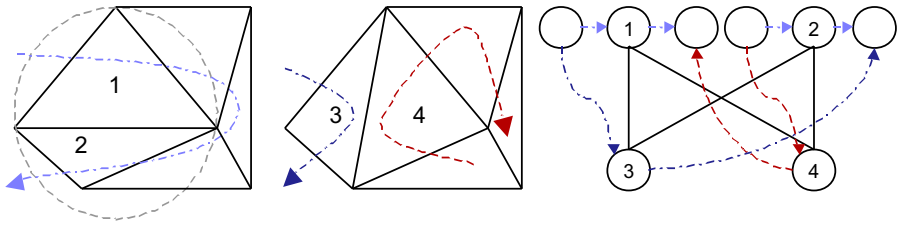


Figure 4.9: Edge flipping (case 3)

flipped, the beginning of the first strip is connected to the end of the second strip and vice versa (Figure 4.10).

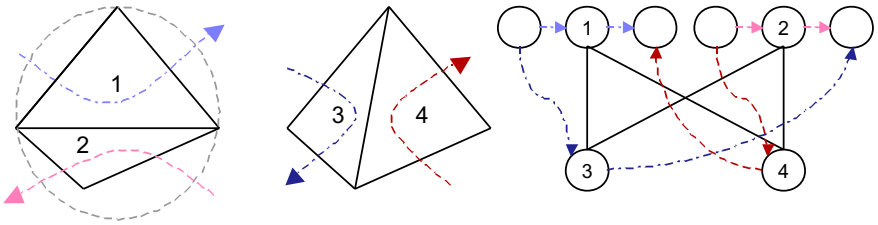


Figure 4.10: Edge flipping (case 4)

When the insertion and flipping step is finished, it is possible to extract the stripification. It can be performed in three steps:

- In the first step, the algorithm is traversing the leaves of the DAG (triangles of the final triangulation). If it is possible, it connects the triangle to an existing strip, if not, a new strip containing this triangle is created.
- In the second step the algorithm goes through the list of strips and tries to concatenate strips into longer ones. To detect whether two strips could be connected or not, each strip has a pointer to its terminal triangles and each terminal triangle points to the corresponding strip.
- To speed up the visualization, we can use the OpenGL vertex arrays or vertex buffers. To be able to use this extension, the algorithm has to extract vertices of each strip into a continuous block of memory in the last step.

4.3 Test and Results

This algorithm was implemented in Borland Delphi 6.0. It has been tested on a set of 16 randomly generated and 8 real terrains. Experiments have been performed on a PC AMD Duron 850MHz with 256MB of RAM, running on MS Windows 2000 system. The implementation was compared to *STRIPE 1.0* [14] with default settings (compiled with gcc, I/O operations excluded from time measurement) and to my own implementation of *SGI* algorithm [34]. This comparison is not completely fair, because unlike this algorithm, both *STRIPE* and *SGI* algorithms are more general and work also for fully 3D models. But as far as I know, there are no public free methods for our class of models. Naturally, times of I/O operations have been excluded from measurements.

In Table 4.1 the name and description of all methods is printed. These names are used in the following tables. In Table 4.2 the number of triangles and vertices in models is shown.

DT	Delaunay triangulation only
DTS	Delaunay stripification
DTS(O)	DTS time minus DT time (only the time of stripification)
SGI	Our implementation of SGI method
STRIPE	STRIPE (default settings)

Table 4.1: Methods

model	# of vertices	# of triangles
1	4,897	9,774
2	13,829	27,642
3	15,820	31,617
4	20,014	40,016
5	41,853	83,678
6	60,244	120,465
7	70,433	140,841
8	100,000	199,114

Table 4.2: Models

Next tables show comparison of the *DTS* to *STRIPE* and to *SGI*. Table 4.3 shows the time needed for stripification. The time for the Delaunay stripification is only 2–5% higher than the Delaunay triangulation without stripification (except of the model 1, which is too

small to give reliable results). In comparison to *STRIPE*, the *DTS* is about 8–15 times faster. It is also more than five times faster than the *SGI* algorithm. This speedup is caused by several things. Nearly all temporary structures are accesible directly in *DTS* while in other algorithms we need to create them. The order of insertion of triangles into strips is done simply by traversing the DAG leaves. The concatenation of triangle strips is done via a greedy algorithm which is very fast.

model	DT	DTS	DTS(O)	STRIPE	SGI
1	190	210	20	128	70
2	701	721	20	356	201
3	832	851	19	402	230
4	1072	1132	60	514	290
5	2634	2714	80	1163	591
6	4086	4197	111	1690	872
7	4917	5108	191	1941	1091
8	6349	6599	250	2432	1261

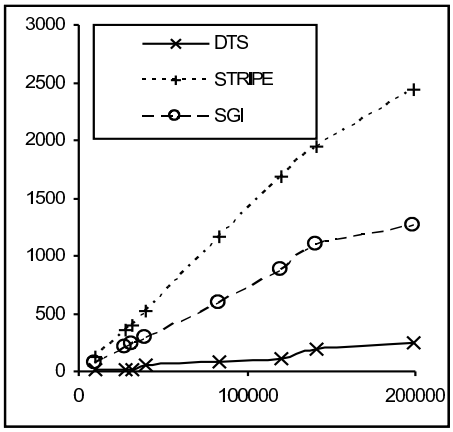


Table 4.3: Runtime in milliseconds

Table 4.4 shows the number of strips needed for a model. We can see that both *SGI* and *STRIPE* creates approximately three times less triangle strips than *DTS*. This is quite surprising because I have expected an algorithm that creates a low number of strips. This problem is caused by a big amount of flips during the triangulation process (6 flips per vertex on average).

model	DTS	STRIPE	SGI
1	638	252	242
2	1785	697	672
3	2030	795	769
4	2625	946	929
5	5457	2052	1895
6	7753	2759	2627
7	9074	3288	3144
8	12048	3445	3363

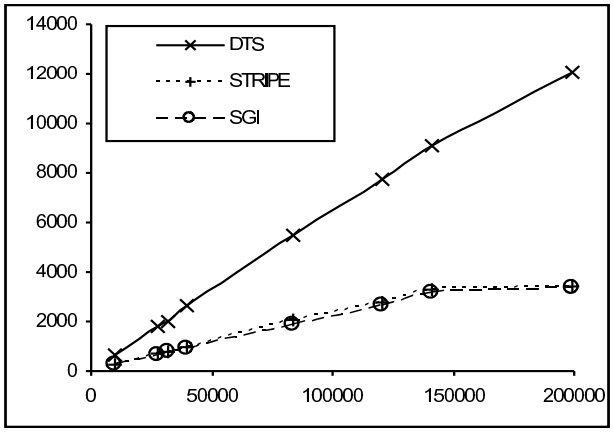


Table 4.4: Number of strips in a model

Table 4.5 lists the number of vertices in strips for all algorithms. The *DTS* algorithm produces 5–6% more vertices than the *STRIPE* and 8–11% more vertices than the *SGI*.

model	DTS	STRIPE	SGI
1	15,390	14,589	14,175
2	43,651	41,489	40,243
3	49,874	47,429	45,936
4	63,012	59,805	57,707
5	131,921	125,090	120,887
6	189,790	179,222	172,991
7	222,116	209,703	202,607
8	261,001	294,706	280,387

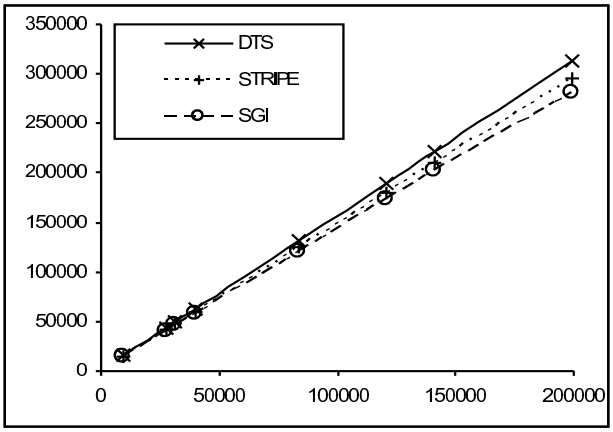


Table 4.5: Number of vertices in strips

There could be two reasons why is my algorithm worse than *SGI* or *STRIPE*. First, the number of strips is higher. Second, in the stripification there exists a lot of fan-like strips caused by the flips (see Figure 4.11). Therefore a combination of triangle strips and triangle fans could bring some additional reduction.

In Figure 4.11 (left) a new vertex is inserted into a triangulation. After the insertion, flips are performed and the order of triangles in the strip is changed. The color intensity marks out the order of triangles.

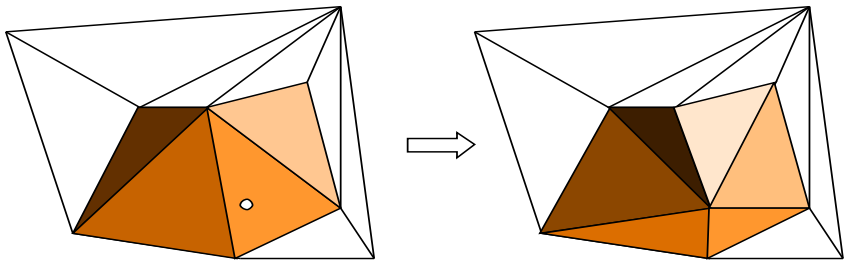


Figure 4.11: Insertion of a vertex changes the order of triangles (the color intensity marks out the order of triangles).

Although my algorithm produces higher number of strips, the speedup is sufficient for the previews. There is probably still a place for reducing the number of strips by some improvements in the insertion and flipping stage.

5 Ideas and Future Work

In this section, my current and future work is presented. There are also several ideas that may or may not be realized in the future.

Several last months I have been working on a new stripification algorithm that uses the duality approach. This algorithm is inspired by the algorithm for searching the Hamiltonian circle designed by Christophides [9] and improved by Kocay [21]. Their algorithm is based on an exhaustive search of paths in a graph. The algorithm starts with an arbitrary node and any incident edge. While recursively extending the path, edges that are incident to the node, which is in the middle of the path, are removed, because there is no possibility to use them (Hamiltonian path visits each node only once). In some cases, this edge removal leads to starting of a new path. The algorithm stops in the case that a Hamiltonian path was found. The algorithm works well for Hamiltonian graphs (i.e., graphs that contain a Hamiltonian path).

From that algorithm, I have taken the basic idea – to make a path containing a node of degree of two and one of its adjacent nodes (such nodes has to be inside a strip, otherwise the strip is broken here). As it is not necessary to produce a single strip, but a small group of strips, the exhaustive search part can be eliminated and the complexity of the algorithm can be reduced to $O(n)$.

My new algorithm does not build one strip at a time, but it creates a strip for each suitable group of triangles and concatenates these strips if possible. Such an approach produces triangle strips of about the same length and it avoids short or singleton strips (i.e., strips containing one triangle only).

According to the first tests, the algorithm is able to produce a stripification with very low number of triangle strips (comparable to *tunneling* [32], which produces the lowest number of strips from all algorithms – as far as I know). On the other side, it seems to produce a huge number of vertices.

For the future, there are several challenges related to this algorithm:

- One of the most import task for the future is to find a way how to reduce the number of swaps (number of vertices) in triangle strips produced by this algorithm and create even better stripification. This improvement should increase the rendering speed.

- There is probably still a place to create even less triangle strips by using the loops, which occasionally appear in the stripification. In the presented algorithm, the edges that could lead to such a loop are removed to speed up the algorithm. On the other

side, such loops could be very useful, because they can be disconnected on any segment and concatenated with some other strip, which is starting/ending in the neighborhood of this loop.

- Because the algorithm is based on the dual graph, it is also possible to make a modification which uses a weighted graph. Changing the weights of edges could help to control better the stripification process. It could be used to avoid swaps, make more local strips (i.e., strips that are located on a small area and could be removed by some clipping algorithm), etc.

- It could be also possible to combine this new algorithm with Belmonte's algorithm [5] and produce triangle strips, which are preserved during the simplification process. If the stripification will be constructed from a weighted graph, this can be easily archived.

At present, I am also cooperating with Radek Sviták on a FRVŠ/G1 on the data reconstruction from orthogonal slices. Our goal is to use as much information as possible from the reconstruction process for the stripification. We hope that such an approach will increase the quality of the stripification.

As the stripification problem is NP and there is no exact criterion for the "optimal stripification", I would like to perform a set of tests to get some better definition of this problem. According to the tests presented in Section 3.4, one can see that the optimum stripification is neither the one with the lowest number of strips nor the one with lowest number of vertices. As the speed of visualization of triangle strips is hardware dependent there is probably no exact answer. On the other side, I hope it is possible to get at least some guidelines.

I was quite disappointed by the results of Delaunay stripification. Still, I am convinced that there exists a better way how to produce a good stripification for 2D data. I think that one possible way is to make a combination of Delaunay and Hamiltonian triangulation to obtain a good triangulation with a low number of strips. The main idea is to create the Hamiltonian triangulation – using the onion method. This triangulation can be improved by repairing "bad" triangles. The repairing process can stop after the triangulation achieved some requested quality or the number of strips reaches some predefined maximum. The main problem of this idea is that the quality of the triangulation decreases very rapidly with each triangle that does not fulfill the Delaunay condition.

The algorithm described in [30] or some modification of this algorithm could be probably used to generate a dynamic stripification for flyovers of some huge terrain data by

moving the reference point to the viewers position. Such an approach can be used to generate a stripification of some limited part of the triangulation. The restripification process should be started e.g. after a new block of data is loaded into the memory.

I would like to establish a cooperation with Sebastian Krivograd on compression of triangular meshes. During the decompression phase of the algorithm presented in [40], there is a possibility to construct the stripification directly, without decompressing the triangle mesh first and stripifying it afterward. This could significantly speedup the preprocessing stage of rendering.

References

[1] K. Akeley, P. Haeberli, and D. Burns. tomesh.c. http:// research.microsoft.com/ ∼hollasch/ cgindex/ geometry/ tomesh.c.

[2] K. Akeley, P. Haeberli, and D. Burns. tomesh.c. C Program on SGI Develope's Toolbox CD, 1990.

[3] E.M. Arkin, M. Held, J.S.B. Mitchell, and S.S. Skiena. Hamiltonian Triangulations for Fast Rendering. *Visual Computer*, vol. 12, no. 9, pp. 429–444, 1996.

[4] R. Bar-Yehuda and C. Gotsman. Time/Space Tradeoffs for Polygon Mesh Rendering. *ACM Transactions on Graphics*, vol. 15, no. 2, pp. 141–152, 1996.

[5] O. Belmonte, J. Ribelles, I. Remolar, and M. Chover. Searching Triangle Strips Guided by Simplification Criterion. In V. Skala, editor, *WSCG 2001 Conference Proceedings*, 2001.

[6] M.de Berg. Simple Traversal of a Subdivision without Extra Storage. *International Journal of GIS*, 1997.

[7] K.Q. Brown. Voronoi Diagrams from Convex Hulls. In *Information Processing Letters*, pp. 223–228, 1979.

[8] CCGDV, University of West Bohemia. Data archive. http:// herakles.zcu.cz/ research/mve/ download.php.

[9] N. Christophides. *Graph Theory, an Algorithmic Approach*. Academic Press, New York, 1975.

[10] CYBERWARE. Sample models. http:// www. cyberware. com/ samples/.

[11] M. Deering. Geometry compression. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, pp. 13–20. ACM Press, 1995.

[12] R.A. Dwyer. A Simple Divide-and-Conquer Algorithm for Computing Delaunay Triangulations in O(n log log n) Expected Time. In *Proceedings of the Second Annual Symposium on Computational Geometry*, pp. 276–284. ACM Press, 1986.

[13] J. El-Sana, E. Azanli, and A. Varshney. Skip Strips: Maintaining Triangle Strips for View-Dependent Rendering. In *Proceedings of the Conference on Visualization '99*, pp. 131–138. IEEE Computer Society Press, 1999.

[14] F. Evans. STRIPE, 1998. http://www.cs.sunysb.edu/ ˜stripe/.

[15] F. Evans, S. Skiena, and A. Varshney. Completing Sequential Triangulations is Hard. Technical report, Department of Computer Science, State University of New York at Stony Brook, 1996.

[16] F. Evans, S. Skiena, and A. Varshney. Optimizing Triangle Strips for Fast Rendering. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pp. 319–326, 1996.

[17] M. Garland and P.S. Heckbert. Surface Simplification Using Quadric Error Metrics. *Computer Graphics*, vol. 31, pp. 209–216, 1997.

[18] Georgia Institute of Technology. Large Geometric Models Archive. http:// www.cc.gatech.edu/ projects/ large_models/.

[19] M. Held. Efficient And Reliable Triangulation Of Polygons. In *Proceedings of Computer Graphics International*, pp. 633–643, 1998.

[20] H. Hoppe. Optimization of Mesh Locality for Transparent Vertex Caching. In Alyn Rockwood, editor, *Siggraph 1999, Computer Graphics Proceedings*, pp. 269–276, Los Angeles, 1999. Addison Wesley Longman.

[21] W. Kocay. An Extension of the Multi-Path Algorithm for Finding Hamilton cycles. *Discrete Mathematics 101*, pp. 171–188, 1992.

[22] I. Kolingerová and B. Žalik. Improvements to Randomized Incremental Delaunay Insertion. *Computers & Graphics*, vol. 26, pp. 477–490, 2002.

[23] D. Kornmann. Fast and Simple Triangle Strip Generation. http:// www.dlc.fi/ dkpa/ strip/ strip.html.

[24] D. Kornmann. Fast and Simple Triangle Strip Generation. Technical report, VMS Finland, Espoo, Finland, 1999.

[25] D.E. Knuth L.J. Guibas and M. Sharir. Randomized Incremental Construction of Delaunay and Voronoi Diagrams. In M. S. Paterson, editor, *Automata, Languages and Programming: Proc. of the 17th International Colloquium*, pp. 414–431. Springer, New York, 1990.

[26] NVIDIA Corporation. Using Vertex Buffer Objects. White Paper: http:// developer.nvidia.com/ object/ using_VBOs.html, 2003.

[27] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. In *Workshop on Algorithms and Data Structures*, pp. 437–449, 1989.

[28] M.V.G.da Silva, O.M.van Kaick, and H. Pedrini. Fast Mesh Rendering through Efficient Triangle Strip Generation. http:// pet.inf.ufpr.br/ ∼om/ software.php.

[29] M.V.G.da Silva, O.M.van Kaick, and H. Pedrini. Fast Mesh Rendering through Efficient Triangle Strip Generation. In *WSCG'2002*, pp. 127–134, 2002.

[30] B. Speckmann and J. Snoeyink. Easy Triangle Strips for TIN Terrain Models. In *Canadian Conference on Computational Geometry*, pp. 239–244, 1997.

[31] Stanford Computer Graphics Laboratory. The Stanford 3D Scanning Repository. http:// graphics.stanford.edu/ data/ 3Dscanrep/.

[32] J. Stewart. Tunneling for Triangle Strips in Continuous Level-of-Detail Meshes. In *Graphics Interface*, pp. 91–100, 2001.

[33] P. Su and R.L.(Scot) Drysdale. A Comparison of Sequential Delaunay Triangulation Algorithms. In *Symposium on Computational Geometry*, pp. 61–70, 1995.

[34] P. Vaněček. Comparison of Stripification Techniques. In *6-th Central European Seminar on Computer Graphics CESCG'02*, pp. 65–74, 2002.

[35] P. Vaněček and I. Kolingerová. Fast Delaunay Stripification. In *Proceedings of the 19th Spring Conference on Computer graphics*, 2003.

[36] L. Velho, L.H.de Figueiredo, and J. Gomes. Hierarchical Generalized Triangle Strips. *The Visual Computer*, vol. 15, no. 1, pp. 21–35, 1999.

[37] J.C. Xia, J. El-Sana, and A. Varshney. Adaptive Real-Time Level-of-Detail-Based Rendering for Polygonal Models. *IEEE Transactions on Visualization and Computer Graphics*, vol. 3, no. 2, pp. 171–183, 1997.

[38] X. Xiang. Fast Triangle Strip Generator. http:// www.ams.sunysb.edu/ ∼xxiang/ strip.html.

[39] X. Xiang, M. Held, and P. Mitchell. Fast and Effective Stripification of Polygonal Surface Models (short). In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1999.

[40] B. Žalik and S. Krivograd. Compression of Triangular Meshes Processing Two Triangles at the Same Time. *Contributions to Geometric Modelling and Multimedia*, vol. 2, no. 8, pp. 1–23, 2002.

A Activities

Publications

- Vaněček P. and Kolingerová I. Weighted Multi-Path Algorithm for Triangle Strips, *Electronic Computers and Informatics 2004*, Herlany, Slovakia, 2004 (waiting for review).
- Vaněček P. and Kolingerová I. Multi-Path Algorithm for Triangle Strips, In *Computer Graphics International (CGI) 2004*, Crete, Greece, 2004 (accepted as full paper).
- Vaněček P. and Kolingerová I. Fast Delaunay Stripification, In *Spring Conference on Computer Graphics (SCCG) 2003*, Budmerice, Slovakia, 2003 (also published in ACM ISBN 1-58113-861-X).
- Vaněček P. Comparison of Stripification Techniques. In *6-th Central European Seminar on Computer Graphics (CESCG) 2002*, pages 65–74, Budmerice, Slovakia, 2002.

Related Talks

- Vaněček P. Teorie grafů a její aplikace v počítačové grafice, Center of Computer Graphics and Data Visualization, University of West Bohemia, Pilsen, Czech Republic, April 2004.
- Vaněček P. Trojúhelníkové stripy, Center of Computer Graphics and Data Visualization, University of West Bohemia, Pilsen, Czech Republic, November 2003.
- Vaněček P. Triangle Strips For Fast Rendering, Technical University of Graz, Austria, October 2003.
- Vaněček P. Triangle Strips For Fast Rendering, University of Maribor, Slovenia, September 2003.

Stays Abroad

- Technical University of Graz, Austria, October 2003.
- University of Maribor, Slovenia, September 2003.
- University of Ioannina, Greece, February – August 2001.

B Models

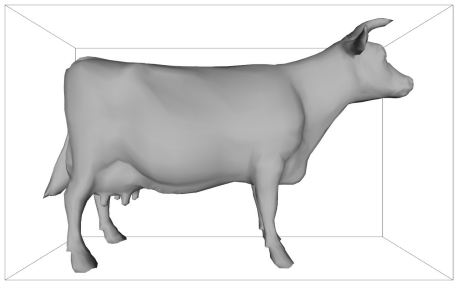


Figure B.1: Cow. 2905 vertices, 5804 triangles.

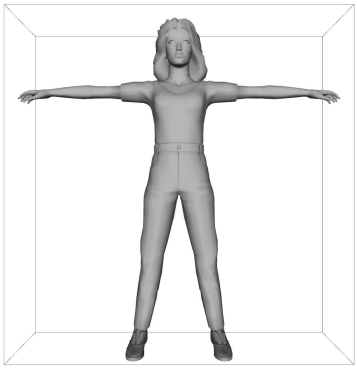


Figure B.2: Demi. 9138 vertices, 17506 triangles.

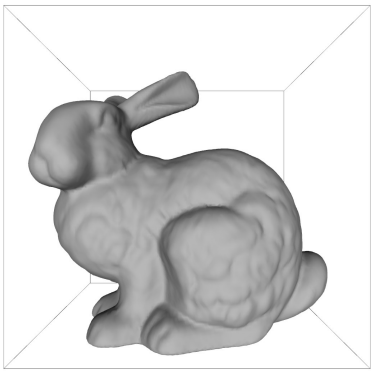


Figure B.3: Bunny. 35947 vertices, 69451 triangles.

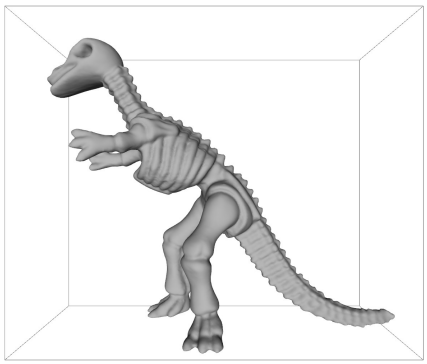


Figure B.4: Dinosaur. 56194 vertices, 112384 triangles.

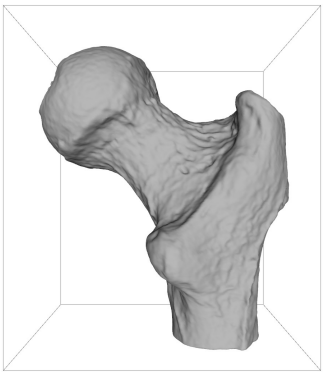


Figure B.5: Balljoint. 137062 vertices, 274120 triangles.

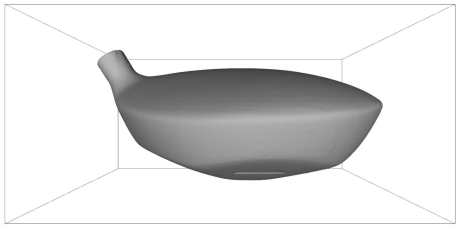


Figure B.6: Club. 209779 vertices, 419554 triangles.

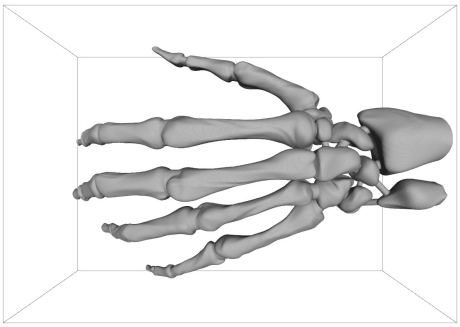


Figure B.7: Hand. 327323 vertices, 654666 triangles.

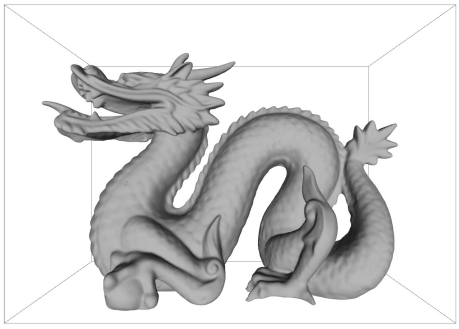


Figure B.8: Dragon. 437645 vertices, 871414 triangles.

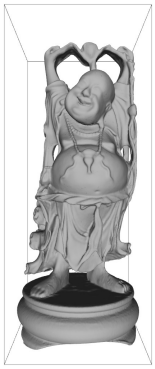


Figure B.9: Happy. 543652 vertices, 1087716 triangles.

Figure B.10: Blade. 882954 vertices, 1765388 triangles.

C Output Examples

The bunny model consists of 35947 vertices and 69451 triangles. The *SGI* method (LS heuristic) produces the highest number of triangle strips (Figure C.1). As these strips do not contain a high number of swaps, they are narrow straight. On the other side, *TUNNEL*ing produces more than 20 times lower number of triangle strips, but the number of vertices (i.e., swaps) is more than 20% higher (Figure C.4). These strips cover huge regions.

Figure C.1: SGI-LS. 3560 strips, 81730 strip vertices.

Figure C.2: STRIPE-Q. 1229 strips, 82760 strip vertices.

Figure C.3: FTSG-SGI. 618 strips, 85362 strip vertices.

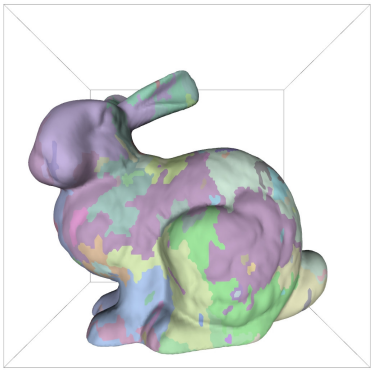


Figure C.4: TUNNEL. 166 strips, 98503 strip vertices.