University of West Bohemia

Faculty of Applied Sciences

Doctoral Thesis

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in specialization

Computer Science and Engineering

Ing. Petr Vaněček

Triangle Strips for Fast Rendering

Supervisor: Doc. Dr. Ing. Ivana Kolingerová

Date of state doctoral exam: June 30, 2004

Date of thesis consignation: June 28, 2005

Pilsen, 2005

Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Disertační práce

k získání akademického titulu

doktor

v oboru

Informatika a výpočetní technika

Ing. Petr Vaněček

Triangle Strips for Fast Rendering

Školitel: Doc. Dr. Ing. Ivana Kolingerová

Datum státní závěrečné zkoušky: 30. června 2004

Datum odevzdání práce: 28. června 2005

V Plzni, 2005

Prohlášení

Předkládám tímto k posouzení a obhajobě disertační práci zpracovanou na závěr doktorského studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni.
Prohlašuji tímto, ze tuto práci jsem vypracoval samostatně, s použitím odborné literatury a dostupných pramenů uvedených v seznamu, jenž je součástí této práce.

V Plzni dne 28. června 2005 Ing. Petr Vaněček

Abstract

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.

We introduce our algorithm for stripification of Delaunay triangulated irregular networks (TIN). The method does not produce stunning results, but it is fast enough to create previews for different levels of detail of Delaunay triangulation during the incremental construction.

For triangle meshes we have designed a new stripification algorithm based on Hamiltonian path search in a dual graph of triangulation. As far as we know, this algorithm produces the lowest number of strips in linear time. We have also proposed a modification of this algorithm that deals with the weights in the dual graph to allow a better control of stripification process.

As tetrahedral meshes are becoming very important data representation in many graphic and volume computation applications, we present some state of the art of tetrahedral strips. We also show, how to extend our triangle stripping algorithm for tetrahedral meshes. Some early tests and results on the field of tetrahedral stripification are included.

We have suggested a new specialized stripification algorithm for purely quadrilateral meshes. For these type of meshes, the algorithm produces high quality stripifications.

Finally, we present a comparison of some of the most important stripification algorithms on a set of reallife and artifical objects. We also show, how the topology can influence the stripification process.

Abstrakt

V dnešním světě počítačové grafiky patří trojúhelníkové sítě k často používaným reprezentacím. Častým úkolem počítačové grafiky je rychlé zobrazování takovýchto sítí. V mnoha případech je rychlost zobrazování limitována propustností sběrnice. Jednou z možností, jak snížit množství dat, je komprese topologické informace pro přenos po sběrnici a její dekomprese až v GPU. Protože sousedící trojúhelníky sdílejí vrcholy na společné hraně, je možné snížit množství dat tím, že tyto vrcholy pošleme pro každou dvojici sousedících trojúhelníků pouze jednou. K tomu je třeba sousedící trojúhelníky pospojovat do souvislých pásů (stripů).

V této práci je navrženo několik algoritmů pro hledání trojúhelníkových pásů pro různá vstupní data. V některých případech (například terénní modely) máme k dispozici pouze množinu vrcholů. Je tedy nutné nejprve vytvořit trojúhelníkovou síť, která je následně převedena do pásů. Přestože námi navržená metoda neprodukuje příliš kvalitní stripifikace, urychlení při vykreslování je dostatečné. Tuto metodu lze navíc použít opakovaně během vytváření sítě a poskytnout tak uživateli rychlý náhled už pro několik bodů a s přibývajícími body tento náhled zpřesňovat.

Nejčastější úlohou je uspořádání již existující trojúhelníkové sítě. Námi navržená metoda vychází z hledání hamiltonovské cesty v duálním grafu trojúhelníkové sítě. Dosažené výsledky jsou velmi kvalitní a z nám známých algoritmů s lineární časovou složitostí vytváří navržená metoda nejmenší počet pásů. Přidáním vah do duálního grafu je navíc možné proces vytváření pásů lépe kontrolovat a vytvářet pásy požadovaných vlastností.

U objemových dat se v posledních několika letech zvětšuje význam čtyřstěnných sítí. Proto uvádíme lehký úvod do problematiky vytváření čtyřstěnných pásů (tetrahedral strips). Zároveň uvádíme rozšíření našeho algoritmu pro čtyřstěnné sítě a porovnání našich výsledků s existujícími metodami.

V některých případech nemusí být model definován trojúhelníkovou sítí, ale obecně n-úhelníkovou. Pro případ čtyřúhelníkové sítě jsme navrhli algoritmus, který vytváří velmi kvalitní stripifikaci (z nám známých algoritmů nejlepší).

V neposlední řadě uvádíme porovnání několik významných metod pro vytváření pásů na reálných i umělých datech a ukážeme vliv pravidelnosti topologie na výsledek stripifikace.

Contents

Acknowledgement

First of all, I would like to thank to my adviser Doc. Ivana Kolingerová, for her patient supervision of my PhD. study. All the time, she was the provider of great ideas and she was pushing me in the right direction. I also would like to thank her for helping me writing up the thesis and all other papers. I would like to thank to Prof. Skala for providing great working conditions and for his critical opinions.

My thanks also go to all my colleagues, especially to Michal, Honza, Jindra and Pepa for many scientific and non-scientific discussions and for all the small things and ideas that makes life easier.

I also would like to thank to all the people that are responsible for the free data archives such as Georgia Large Geometric Models Archive and The Stanford 3D Scanning Repository for all the stunning-looking models that I have used for all my tests. I would like to apologize to Stanford Bunny for all the violation he had to suffer.

Last, I would like to thank to all nice stripification people for being a good inspiration for my work. My thanks goes especially to Prof. James Stewart, Dr. Xinyu Xiang, Dr. David Kornmann, Davis King and Manfred Weiler.

My special thanks go to the gcc, which is more than a good compiler . . .

Introduction

Triangulated surface models are nowadays the 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.

In this chapter, we make a very short introduction to the problem of triangle strips. The overview of the thesis is included in the end of this chapter.

Triangle surface models (often called *meshes*) are nowadays the most often types of geometric objects description in computer graphics. These models are often used in various kind of applications such as CAD/CAM, VR, medical data or computer games. To increase the visual realism, the number of triangles that represents the object is increasing, while the rendering should be performed in the real-time. 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., Continuous Level-of-Detail – (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.

1.1 Triangle Strips

Using a traditional way of encoding of triangle meshes we need three vertices to specify one triangle. As neighboring triangles share an edge and the vertices of this edge, the vertices are sent to the rendering pipeline multiple times. In a typical mesh, the number of vertices is about twice higher than the number of triangles, thus each vertex is specified six times on average.

A *sequential tristrip* is a sequence of $n+2$ vertices that represents n triangles: in Figure 1.1 (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).

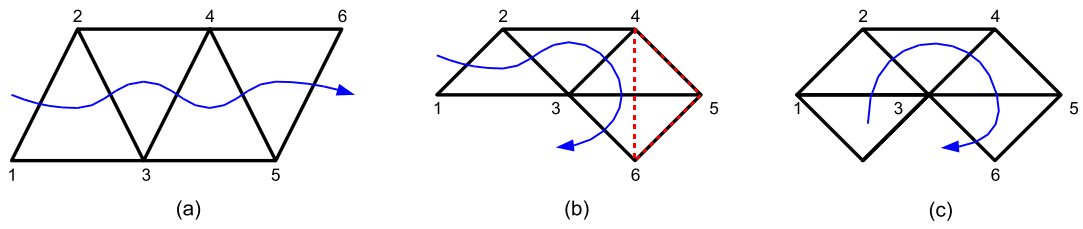


Figure 1.1: *An example of a sequential triangle strip (a), a generalized triangle strip (b) and a triangle fan (c).*

There also exist situations where the triangle adjacency does not allow a sequential encoding. In Figure 1.1 (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 1.1 (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.

As triangle strips can potentially reduce the amount of data needed for rendering, they are widely supported by the graphic hardware and graphic libraries (OpenGL, DirectX, etc.).

To increase the speed of rendering, modern GPUs contain a small FIFO vertex cache (of size of tens of vertices) that prevents the re-processing of already cached vertex. To maximize the benefit of vertex cache, the mesh triangles have to be rendered in an order which is somehow local – to minimize the average cache miss rate (*ACMR*), which is defined as a ratio of cache misses to total number of triangles and it depends on the size k of the cache:

$$ACMR(k) = \frac{number\ of\ cache\ misses}{number\ of\ triangles}$$

Containing the last two vertices, triangle strips behave very well on systems with vertex cache of size two. Although for systems with larger caches, triangle strips are not necessarily the fastest way of rendering, still, they provide high performance rendering on many low-end and mid-end GPUs.

1.2 Thesis Overview

We start with possible classifications of stripfication algorithms and with a short overview of existing algorithms in Chapter 2.

In Chapter 3, we introduce our algorithm for stripification of Delaunay triangulate irregular networks (TIN). The method does not produce stunning results, but it is fast enough to create previews for different levels of detail of Delaunay triangulation during the incremental construction.

We have developed a stripification algorithm based on an algorithm for searching a Hamiltonian path. As far as we know, this algorithm produces the lowest number of strips of all linear time algorithms. The description of our new algorithm is presented in Chapter 4.

Next (Chapter 5), we show how to modify this algorithm to produce even better results. The modification is based on a weighted dual graph of triangulation. We also demonstrate the possibilities of this modification using a very simple weight criterion. Surprisingly, using this criterion, the algorithm produces stripification of very good quality.

The importance of computation and visualization of tetrahedral meshes is growing in last few years. In Chapter 6, we present a short introduction to tetrahedral strips, and we show that our algorithm can be modified for tetrahedra strips.

In Chapter 7, we describe a new specialized stripification algorithm for purely quadrilateral meshes. For these type of meshes, the algorithm produces high quality stripifications.

We also include a set of tests of several important methods to give the reader a better possibility to compare stripification methods. This comparison is done on a set of reallife models and on a set of artifical objects. We have also studied the influence of topology on the quality of stripification. Some of our experiments and results are discussed in Chapter 8.

Finally, we conclude this work in Chapter 9 and we suggest several possible direcions for our future work.

State of the Art

In this chapter we present a list of nearly thirty existing stripification algorithms. As the number of stripification algorithms is quite high, first we propose several ways how to classify them.

2.1 Classification

Triangle stripification algorithms can be categorized in several different ways. Here we enumerate five classifications that can be used:

1. According to the type of input data (isolated vertices, triangles, etc.).
2. According to the type of meshes (static meshes, CLOD, etc.).
3. According to the type of optimization (minimization of number of strips, minimization of number of vertices).
4. According to the type of heuristic function (local heuristic, global heuristic).
5. According to the hardware support (optimization for vertex caches).

One of the possible classifications is based on the type of input data. The first category of algorithms takes only the geometrical information (i.e., only the vertices) as an input [AHMS96, VK03]. Typically, these algorithms work only with data sets on a plane or with a height field. The second category takes triangles of the model and tries to build triangle strips, not necessarily a single strip, without changes in topology [AHB90, Kor99, Ste01, SKP02, VK04a]. The third category is more general as it takes polygons that are triangulated with respect to the stripification [ESV96b, XHM99, CC99, Tau02]. The last category takes either triangles or polygons and inserts some extra vertices (Steiner points) to achieve a single triangle strip [AHMS96, VFG99, EG04]. In this paper, we will focus on category two and three according to this classification.

The majority of stripification algorithms is designed for static meshes (i.e. meshes without changes in topology). As the complexity of some industrial models is very high, the need of visualization of view-dependent meshes is growing. There are two approaches to use triangle strips in LOD meshes. First, special stripification methods that produce a stripification with some properties [BRRC01, Ste01, VFG99] and second, special data structures and algorithms that can manage the strips during the view-dependent visualization [ESAV99, SP03, RCBR04, DBPM05]. In this work, we use only static meshes for our comparison.

Furthermore, the term ’optimal stripification’ is not uniquely determined. One can optimize the stripification algorithm to produce a 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 [Ste01, PS03, VK04a]. Usually, it is not possible to minimize both these parameters at once – decreasing the number of triangle strips often leads to increase in 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. In our comparison we use both types of heuristic functions if possible, to show the influence of vertex/strip trade off on the rendering speed.

We can also classify the stripification algorithms according to the type of the heuristic function. 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 [ESV96a, Ste01, EMX02, EG04].

Todays GPUs contain large vertex caches and their use can significantly reduce the bandwidth. This criterion was taken into account and several algorithms that respects the vertex cache were developed [Hop99, BD00].

In the next section we describe most of the published stripification algorithms classified according to the type of input data. As the number of stripification algorithms is quite high, the list of algorithms is probably not complete.

2.2 Algorithms

Stripfication of Set of Points

Arkin et al. [AHMS96] suggested two algorithms that can create a Hamiltonian triangulation (i.e., the triangulation that can be covered by a single strip) from a set of points in 2.5D. The first algorithm (the Insertion algorithm) is based on the fact that splitting a triangle into three new triangles by a new vertex insertion does not break the strip. As the triangulation created by this algorithm contains a lot of narrow triangles, they proposed another algorithm (the Onion algorithm). This algorithm computes a set of nested convex hulls. These convex hulls partition the set of points to a set of convex annuli that are triangulated and stripified each with a single strip. Strips from neighboring annuli are concatenated to a single strip covering the whole mesh. Still the quality of the resulting triangulation is not very high.

In [VK03] an algorithm for fast Delaunay stripification is suggested. The algorithm is based on an incremental insertion algorithm for DT [KŽ02]. It simply traverses the DAG structure (Directed Acyclic Graph - it is used for a fast location of vertices in a mesh) and concatenates strips if possible. The method is fast but the quality of stripification is not very good.

Stripfication of Triangle Meshes

One of the first algorithms for stripification of a mesh without changes in topology was developed in SGI [AHB90]. It is a greedy algorithm that constructs the strips by adding adjacent triangles with the lowest degree, which tends to avoid short strips. As this algorithm is easy and fast, its modification has been used in many other algorithms. Kornmann [Kor99] extended this algorithm by adding some other deciding conditions. In [SKP02] an algorithm that creates multiple strips concurrently using the SGI algorithm is suggested. Behr [BA02] improved the speed of the SGI algorithm and suggested to execute a randomized stripification process for several times to obtain a better stripification. Vaněček [Van02] have made a comparison of SGI methods using different heuristic techniques.

Stewart [Ste01] proposed a method that works on the dual graph of the triangulation (i.e., the graph where each node corresponds to a triangle and adjacent triangles are connected with an edge in the graph). He presented a new graph operator – tunneling which can reduce the number of strips by one. This operator can be used repeatedly to improve the quality of stripification. Furthermore this operator can be used during the mesh simplification. The number of strips produced by this algorithm is very low but it takes a long time to produce a stripification as the algorithm uses a breadth-first search. Some improvements of this method were suggested in [PS03, PS04]. Porcu [PSS05] also suggested an algorithm that maintains the stripification in progressive meshes. This algorithm uses lookup tables to repair a stripification after a vertex split operation. When the quality of stripification falls bellow some critical threshold, the tunneling operator is used.

Another algorithm that produces very low number of triangle strips is presented in [VK04a]. It is based on the idea that triangle strips has to go through triangles with only two neighbors, otherwise the strip would be broken. This algorithm starts with many triangle strips that are being concatenated as the stripification process continues. This algorithm was extended in [VK04b]. This extension uses a weighted graph to create a stripification that is preserved during a simplification process.

For the progressive meshes, Belmonte [BRRC01] suggested the algorithm that uses a weighted spanning tree of the dual graph to construct the stripification. The assignment of weights is guided by the simplification criterion (i.e., edges that will be collapsed first has lower weight and vice versa). The algorithm can handle 3D meshes.

Speckmann [SS97] introduced an algorithm designed only for TIN (triangulated irregular networks). The algorithm constructs a spanning tree that is based on the euclidean distance of the current triangle to an arbitrary point. Such a spanning tree has two nice properties: first, the branches of the tree typically alternate from left to right (i.e., strips contains only a low number of swaps); and second, there is no need to store the spanning tree explicitly as it depends on the geometry of the TIN. The algorithm is very fast but the stripification is not very good. On the other side, as there is no need to store any information about the strips, it can be used for large data sets.

Estkowski [EMX02] suggested a more theoretical algorithm that uses integer programming to obtain an optimal decomposition into triangle strips. As the complexity of this algorithm is quite high ($O(n^2)$), it is not possible to use it for real models.

Šíma [Ším04] proposed a new stripification method based on a minimum energy problem in Hopfield nets. Similar approach was presented by Pospíšil [PZ04]. Although these algorithms are very slow and produce stripifications of average qualities, the main idea is very original and interesting.

Stripification of Polygonal Meshes

STRIPE [ESV96a] is one of the best known algorithms for stripification. It has average results for fully triangulated models (as it uses the SGI algorithm), but it has very good results for models with quads (quadrilateral faces). The algorithm uses a global heuristic to find quadrilateral patches and stripify them with a single strip.

Another algorithm for not fully triangulated models is *FTSG* (Fast Triangle Strip Generator) [XHM99]. The algorithm constructs a spanning tree of the dual graph of triangulation. This tree is partitioned to a set of strips by dynamical programming and these strips are concatenated if possible.

Cheng [CC99] introduced a dynamic programming based algorithm for such a triangulation of a simple polygon that can be decomposed into a minimum number of triangle strips. The complexity of the algorithm is $O(n^3)$, thus it is not suitable for a practical purpose.

In the last years, quadrilateral meshes became a popular representation in visualization and computer animation. Taubin [Tau02] suggested an algorithm that can cover any connected manifold quadrilateral mesh without boundaries with a single strip. First, the algorithm finds an Eulerian circuit, which is partitioned to a set of Hamiltonian cycles. Then, these cycles are concatenated by flipping a diagonal of the corresponding quad.

A new algorithm for fully quadrilateral meshes was presented in [VSKS05]. It is a modification of greedy *SGI* algorithm. First, the algorithm creates sequences of neighboring quads. Then, these sequences are decomposed into triangle strips. As it follows the heuristic criterion that tends to not produce a swap, the resulting stripification contains only a low number of vertices. Creating the sequence of quads first and producing the triangle strips afterward (i.e., splitting quads to triangles respecting the sequence) significantly reduces the number of strips.

Stripification with Inserted Steiner Points

To obtain a single strip during the stripification process, it is possible to insert special vertices (Steiner points) that usually change only the topology but not the geometry of the mesh. An algorithm that produces a single strip triangulation of manifolds is proposed in [EG04]. It is based on a perfect matching algorithm. By removing all matching edges from the dual graph of the triangulation, a set of disjoint cycles arises. These cycles can be connected at a cost of two new triangles per connection. The algorithm complexity is higher as it uses a perfect matching algorithm (which is $O(n)$ for planar graphs and $O(n \log^4 n)$ in general). According to the presented results, the number of new triangles is less than 2% of the input number of triangles.

Velho [VFG99] suggested a subdivision scheme for progressive triangular meshes. In several cases it is necessary to insert Steiner vertices to preserve a constant number of triangle strips during the refinement process.

Vertex Cache

The problem of vertex caches and rendering sequences is closely related to the problem of stripfication, thus we present several important works from this topic.

Deering [Dee95] 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.

Bar-Yehuda [BYG96] studied the impact of the GPU's buffer size to rendering time (time/space tradeoff). 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 [Hop99] 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.

Bogomjakov [BG01] suggested an algorithm that produces a rendering sequences that are not dependent on the vertex cache size. He also proposed an update algorithm that automatically reorders the rendering sequence in progressive meshes.

A detailed description of some of the most important algorithms as well as their comparison can be found in [Van04]. In this thesis, we present a comparison of some of the existing methods and our new algorithms. This comparison can be found in Chapter 8.

Delaunay Stripification

In this chapter we 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.

3.1 Delaunay Triangulation

At the beginning we will describe the Delaunay triangulation and structures that we use. More details about the Delaunay triangulation are e.g. in [Dwy86].

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 [Dwy86],
- incremental insertion [LGS90, KŽ02],
- high-dimensional embedding [Bro79].

Although the fastest method is divide & conquer [Dwy86] (according to [SD95]), we 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 [KŽ02].

The incremental insertion algorithm is described in Figure 3.1.

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 3.2. In the first step, a new vertex is inserted. Then the corresponding triangle is divided into three new triangles (4,5,6). As the new triangles do not fulfill the Delaunay condition, edge flips are performed in steps three and four.

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

1. Create a temporary triangle (with points pt_1, pt_2, pt_3), 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 point p 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.
3. Remove all triangles that are incident to pt_1, pt_2 or pt_3.

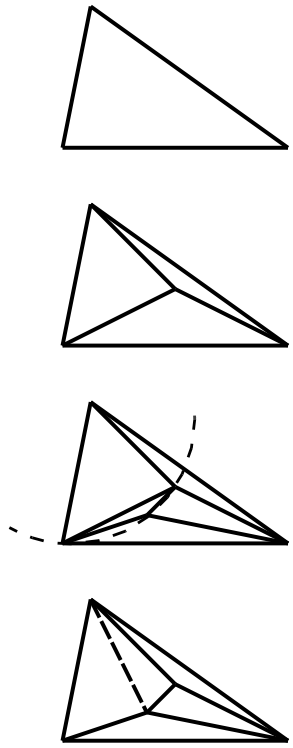


Figure 3.1: *Algorithm steps for the incremental insertion of DT and an example of the triangulation construction.*

3.2 Delaunay Stripification

To speed up the visualization of different levels of detail of the triangulation, it is possible to use triangle strips. In Figure 3.2 (d), one can see that it is possible to obtain a stripification for each step of the triangulation process by traversing the leaves of the DAG structure very quickly. This algorithm was published in [VK03].

To improve the quality of stripification, it is necessary to modify the existing algorithm [KŽ02] 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 3.3).

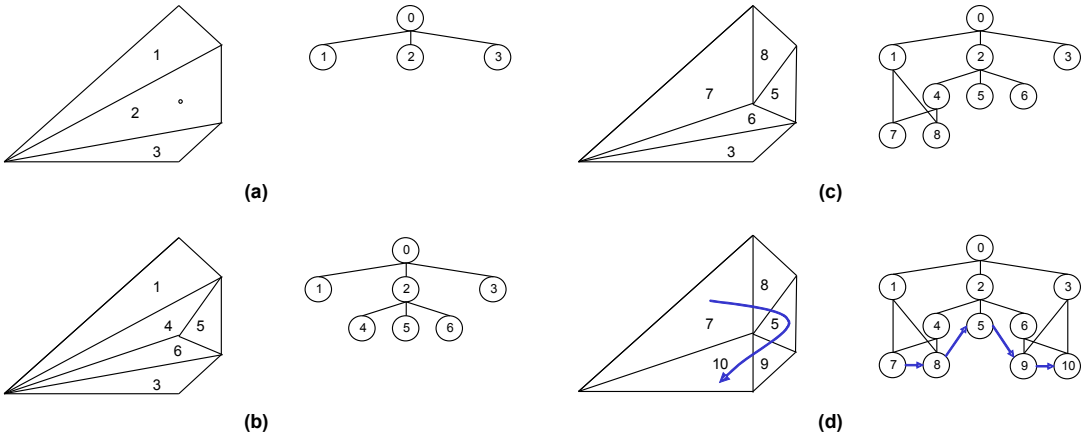


Figure 3.2: *An example of DAG. A new point is inserted into a triangulation (a). The corresponding triangle is subdivided into three new triangles (b). The triangles are checked for the Delaunay condition (c) and (d).*

If we don't care about the Delaunay condition (do not perform flips), we obtain a Hamiltonian triangulation (as described in [AHMS96] – we get one strip for the whole triangulation, penalized by worse quality of triangles).

In Figure 3.3 (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.

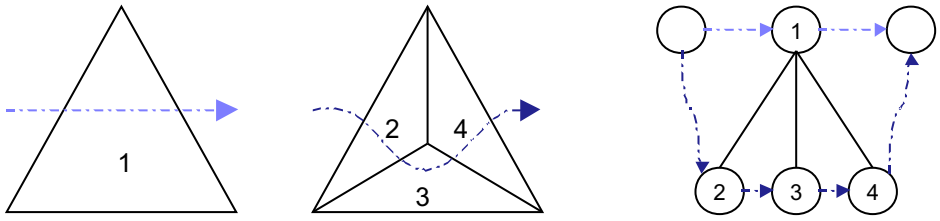


Figure 3.3: *Insertion of a vertex into a triangle.*

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 3.4).

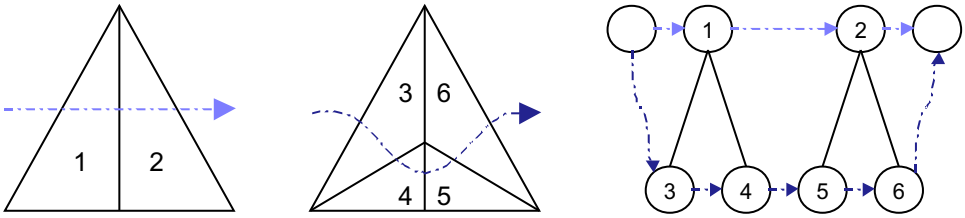


Figure 3.4: *Insertion of a vertex on an edge (case 1).*

The second case, where the incoming edge of the first triangle does not share any vertex 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 3.5 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.

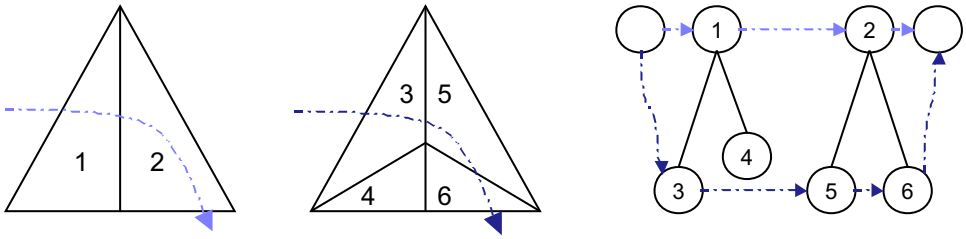


Figure 3.5: *Insertion of a vertex on an edge (case 2).*

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 3.6).

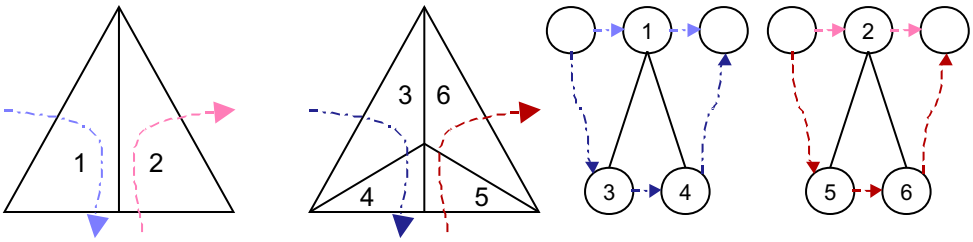


Figure 3.6: *Insertion of a vertex on an edge (case 3).*

To make the Delaunay triangulation, each new triangle has to be checked and if it does 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 3.7).

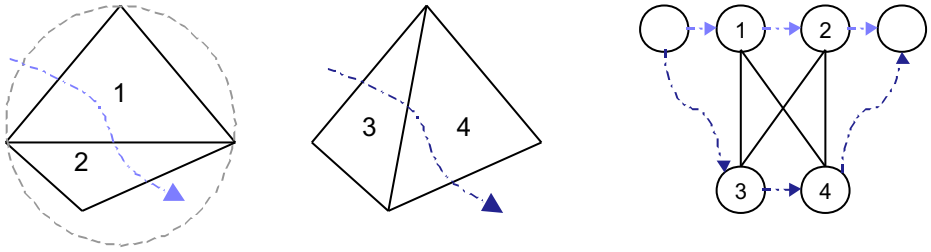


Figure 3.7: *Edge flipping (case 1).*

If the incoming and outgoing edges share a vertex, a new single-triangle strip has to be created (Figure 3.8).

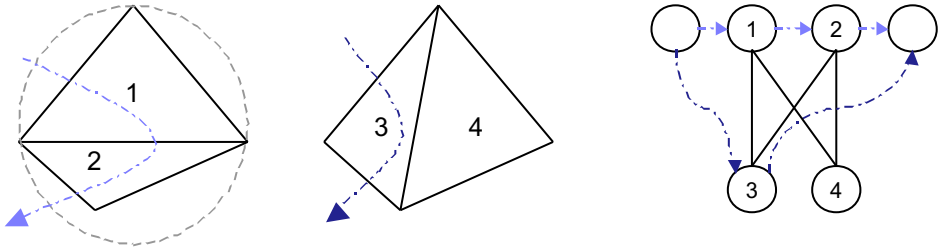


Figure 3.8: *Edge flipping (case 2).*

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 3.9).

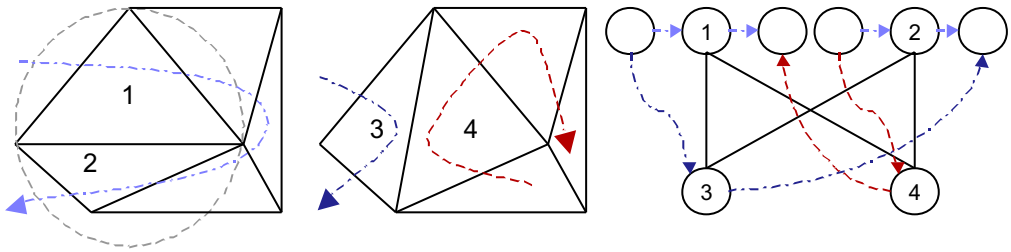


Figure 3.9: *Edge flipping (case 3).*

In the last case the two triangles do not belong to the same strip. After the edge is flipped, the beginning of the first strip is connected to the end of the second strip and vice versa (Figure 3.10).

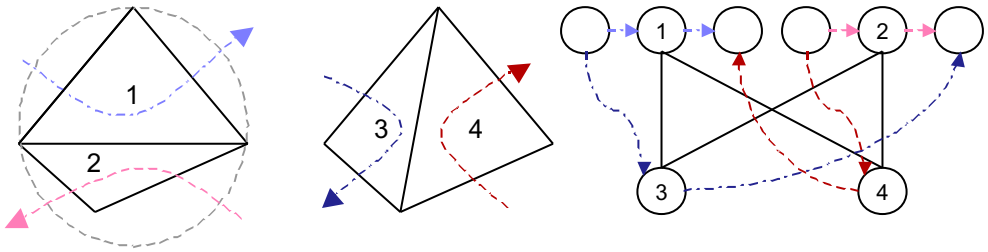


Figure 3.10: *Edge flipping (case 4).*

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.

3.3 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* [Eva98] with default settings (compiled with gcc, I/O operations excluded from time measurement) and to my own implementation of *SGI*

algorithm [Van02]. 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 we 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 3.1 the name and description of all methods is printed. These names are used in the following tables. In Table 3.2 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 3.1: *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 3.2: *Models.*

Next tables show comparison of the *DTS* to *STRIPE* and to *SGI*. Table 3.3 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

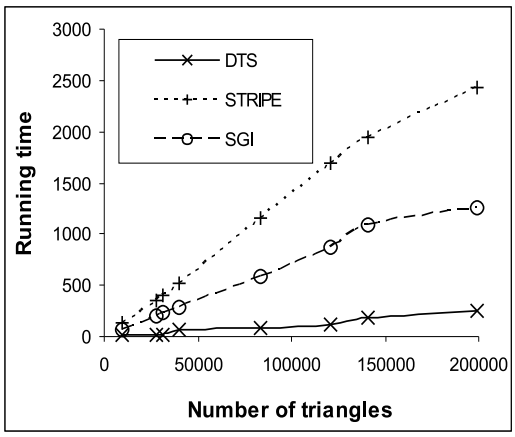


Table 3.3: *Runtime in milliseconds (grey cells emphasize the best values, black cells emphasize the worst values).*

Table 3.4 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 we 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

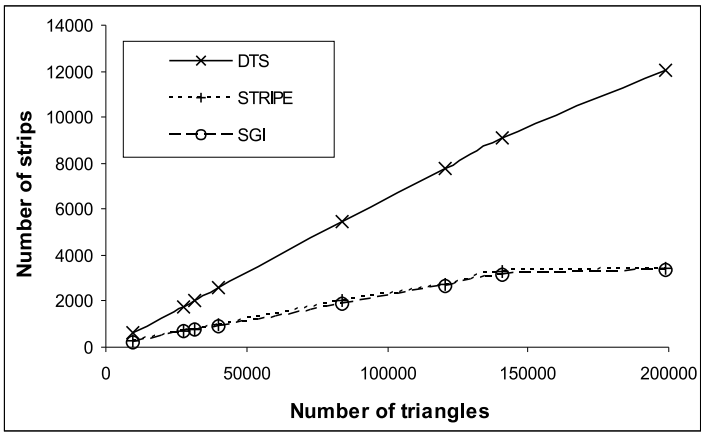


Table 3.4: *Number of strips in a model.*

Table 3.5 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*.

There could be two reasons why is our algorithm worse in the number of vertices 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 3.11). Therefore a combination of triangle strips and triangle fans could bring some additional reduction.

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

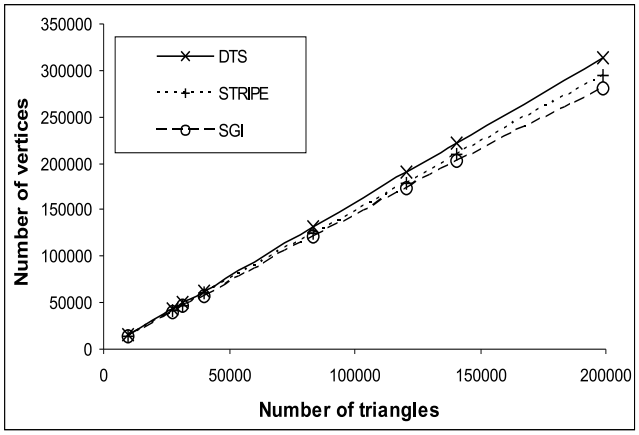


Table 3.5: *Number of vertices in strips.*

In Figure 3.11 (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.

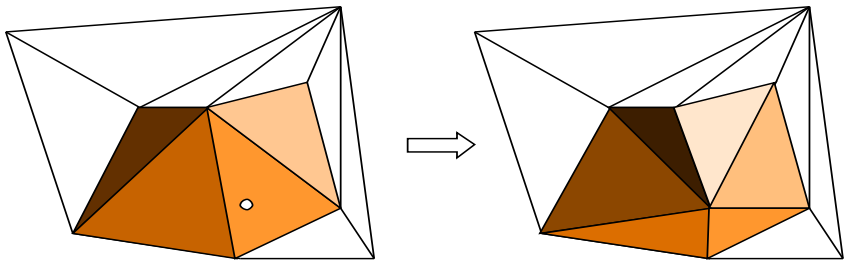


Figure 3.11: *Insertion of a vertex changes the order of triangles (the color intensity marks out the order of triangles).*

The main goal of stripification is the speedup of rendering. In Table 3.6, we present the average number of frames per second for non-stripified model and for models stripified with *DTS*, *STRIPE* and *SGI*. To use the power of the graphic card, we have used the OpenGL display lists.

As shown in the table, using the triangle strips significantly increases the speed of rendering. This speed-up increases with the increase of model complexity, as the data bandwidth is becoming more critical. For most cases, the stripification produced by *SGI* algorithm is rendered most quickly. Although the stripification produced by our new algorithm is the slowest, the differences in rendering speed are not very high.

model	DT	DTS	STRIPE	SGI
1	45.6	49.2	50.2	50.7
2	38.1	50.1	50.4	50.1
3	22.4	38.0	38.7	38.6
4	18.6	34.6	36.3	35.7
5	13.1	26.2	26.9	27.3
6	9.5	20.6	21.3	22.2
7	8.7	19.8	19.9	20.4
8	7.1	13.7	15.0	15.5

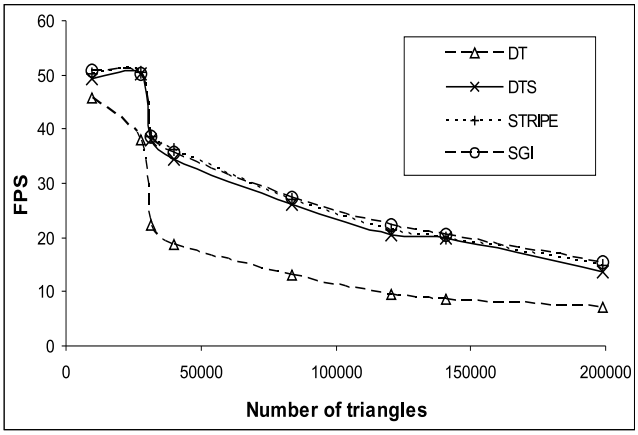


Table 3.6: *Number of frames per second.*

3.4 Summary

We have developed and implemented a new algorithm for triangulation and stripification of models based on Delaunay triangulation using incremental insertion algorithm with DAG. As far as we know, this is the first algorithm which is able to create triangle strips together with the construction of triangulation. Our algorithm is fast enough to create previews for different levels of detail of Delaunay triangulation. Due to greater number of triangle strips, it is better to use some other algorithm for the final stripification.

Although our 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.

Multi-Path Algorithm

In this chapter we describe a new algorithm for stripification of static, fully triangulated meshes and some of its extensions. This algorithm is based on a dual graph of triangulation and it produces a stripification with very low number of triangle strips.

4.1 Multi-Path Algorithm for Hamiltonian Cycles

The stripification problem is related to the problem of searching of the Hamiltonian cycles in the dual graph, i.e., a path connecting all nodes of a graph, visiting each node exactly once.

Christophides [Chr75] and Kocay [Koc92] introduced a Multi-Path algorithm for finding Hamiltonian cycles. This 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 cycle was found. The algorithm works well for Hamiltonian graphs (i.e., graphs that contain a Hamiltonian cycle). For non-Hamiltonian graphs, it is necessary to explore all possibilities, thus it can take a long time.

4.2 Stripification

From the Multi-Path algorithm, we have taken the basic idea – to make a path containing a node of degree of two and one of its adjacent nodes – and we have modified it to better suitability for stripification problem.

Our 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).

According to the degree and status of a corresponding node in the dual graph all triangles are classified into sets.

- U_i – the set of unconnected nodes of degree $i; i \in \{0, 1, 2, 3\}$. Such a node represents a triangle that is not connected in a strip.
- C_i – the set of connected nodes of degree $i; i \in \{1, 2\}$. A triangle represented by such a node is connected to a strip over one edge (i.e., it is a boundary triangle of some strip). As there is no possibility to extend a strip from a connected node of degree zero, such nodes are classified as T.
- T – the set of fully connected nodes. It represents triangles that are inside strips.

The algorithm begins by adding the neighboring information into the dual graph of the triangulation. Considering the number of neighbors, all nodes are classified into the U_i sets (all nodes are unconnected at the beginning).

After this initialization part, the main loop of the stripification process can start. The algorithm chooses one node from the graph following this priority order:

1. an unconnected node from U_0 – a triangle without neighbors (it is a singleton triangle and there is no possibility to connect it to a strip)
2. an unconnected node from U_1 – a triangle with one neighbor (we have to connect it to the neighbor to avoid the singleton strips)
3. a connected node from C_1 – a triangle that is an endpoint of a strip and that has one neighbor (it is good to connect it to its neighbor, to avoid the strip breaking)
4. an unconnected node from U_2 – a triangle with two neighbors (a strip should go through such a triangle)
5. a connected node from C_2 – a connected triangle with two neighbors
6. an unconnected node from U_3 – an unconnected triangle with three neighbors

An edge incident to the selected node and the other node of the edge are chosen. A new strip containing triangles corresponding to the selected nodes is created and the edge is removed from the dual graph. If one (or both) of selected nodes is already a connected node (i.e., it is an endpoint of some strip), the new strip has to be concatenated.

Both selected nodes are moved to a corresponding set: an unconnected node of degree i is moved to connected $i-1$ set, a connected node of any degree is moved to fully connected set T. If a node is moved to the T set, it is also necessary to remove all its remaining edges and to update the status and degree of its neighbors.

To avoid an infinite triangle strip (loop), a simple test which checks the endpoints of the current strip is performed. If there is an edge connecting these two endpoints, it is removed and the status of neighboring nodes is updated.

The main loop is performed as long as there are some not fully connected nodes in the dual graph. In the end, a simple method is used to decompose the list of triangles in strips into a list of vertices of strips (including swaps). We suppose that the orientation of all triangles of the input mesh is consistent. The stripification process is running in linear time.

In Figure 4.1 a pseudocode for the Multi-Path stripping algorithm is shown.

```
input:  list of triangles
output:  list of triangle strips

begin
  Create neighbors;
  Classify nodes;

  while there is any node in the graph do
    Choose starting node t1;
    Choose neighboring node t2 to node t1;
    Add edge (t1,t2) to the list of strips;
    Try to concatenate the new edge with some existing strip;
    Remove edge (t1,t2) from the dual graph;
    Check loop in strips;
  end while;

  ExtractStripsVertices;
end;
```

Figure 4.1: *Pseudo code for the Multi-Path stripping algorithm.*

4.3 An Example

Now we will present an example. Figure 4.2 (a) shows the triangulation and its corresponding dual graph. In the beginning all nodes are classified into U_i sets.

In the first step (Figure 4.2 (b)), a $U2$ node and one incident edge is chosen. This edge is removed from the graph, and a strip of length two is created. The node is moved from U_2 to C_1 and its neighbor is moved from U_3 to C_2.

In the next step (Figure 4.2 (c)), a node $C1$ is available, so the algorithm processes it. The remaining edge of the node $C1$ is removed from the graph and a new strip of length two is created. As the node is already a part of a strip, these two strips are concatenated. The C_1 node is now moved to T (a fully connected node). A similar situation appears in the next two steps (Figure 4.2 (d),(e)).

Now (Figure 4.2 (f)), an edge connecting $C1$ and $C2$ node was removed from the graph and the strip was created. As both nodes are already connected in different strips, all three strips are concatenated into one. The status of both nodes is changed to T and the unprocessed incident edge of the $U3$ node is removed. By this step, the incident $U3$ changed its status to $U2$.

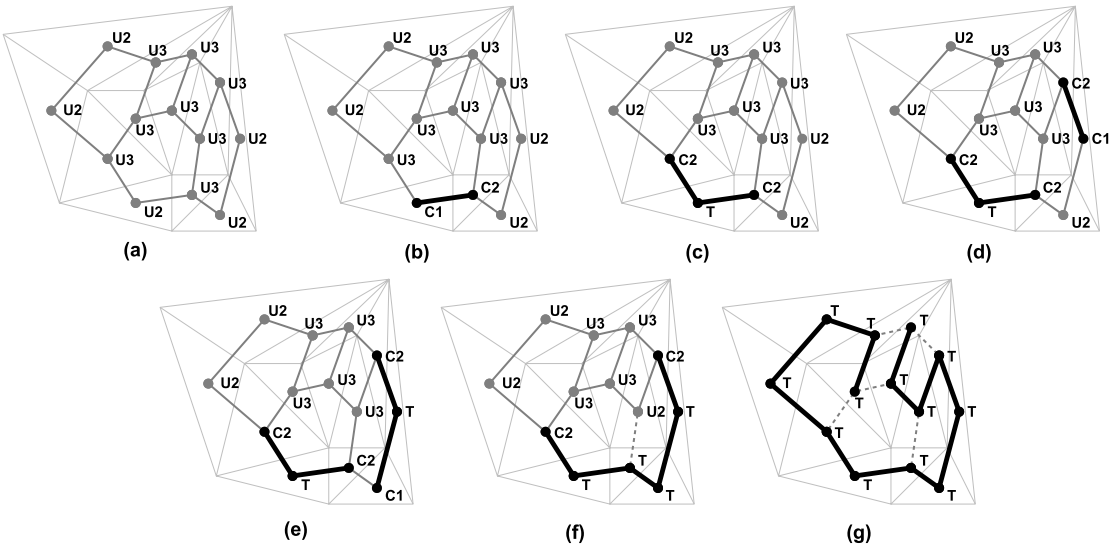


Figure 4.2: *Algorithm steps. The original triangulation and its dual graph (a). In (b) a new strip starting from a* $\mathbf{U_2}$ *node is created and it is extended in the next step (c). As the highest possible priority set is* $\mathbf{U_2}$*, another strip is created (d) and following the rules, it is extended (e). In figure (f), the two already existing strips are concatenated. This leads to an edge removal (a dashed line). The final stripification contains one triangle strip (g).*

When there are only T nodes in the graph, the stripification is done (Figure 4.2 (g)). Finally, the algorithm converts the strips (which are lists of neighboring triangles) to lists of vertices of the strips.

4.4 Experiments and Results

Our new algorithm has been implemented in Borland Delphi 6.0. It has been tested on a set of well known models (Table 4.1). The experiments were performed on a PC INTEL Pentium 4, 1.8GHz, 512MB of RAM, running on MS Windows XP. Naturally, times of I/O operations have been excluded from measurements.

We have chosen models that are often used in other papers and are available on the internet [Sta, Geo, CYB]. All our models are fully triangulated, so we decided to compare our new algorithm (*Multi-Path*) to SGI-based method [Van02]. As our new algorithm produces very low number of triangle strips, we also decided to compare it to *Tunneling* algorithm [Ste01], with the default settings. As far as we know, the tunneling algorithm produces the lowest number of triangle strips.

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

Table 4.1: *Set of testing models.*

Stripification

A comparison of number of strips created by various methods is shown in Table 4.2.

We can see that the number of strips produced by *Multi-Path* algorithm is nearly four times lower than the number of strips produced by *SGI*. For lower resolution models ($<$ 300k of triangles), our new algorithm produces even less vertices than tunneling. For large models, our algorithm produces approximately 40% more strips than tunneling.

On the other side, our new method produces at least three times lower number of strips than other stripification methods (Table 4.3).

In the next table (Table 4.4), a comparison of number of vertices in strips is presented (number of vertices is in thousands). The difference in the number of vertices could not be as big as the difference in the number of strips, because there are two theoretical boundaries. The number of vertices could not be lower than $2 + number\ of\ triangles$ (for a sequential strip, covering the whole triangulation) 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 *Multi-Path* algorithm produces less vertices than the *Tunneling* algorithm, but it produces about 5% more vertices than the *SGI*. Although this difference is not so big, it could lead to a lower frame-rate. In the next chapter, we show a possible way, how to improve the algorithm.

#	model	SGI	Tunneling	Multi-Path
1	cow	98	19	17
2	demi	335	137	97
3	bunny	601	188	156
4	dinosaur	1177	267	308
5	ball joint	2279	707	690
6	club	2658	909	978
7	hand	8997	1944	2227
8	dragon	17399	3672	4876
9	happy buddha	21578	4219	5809
10	blade	23125	5537	5863

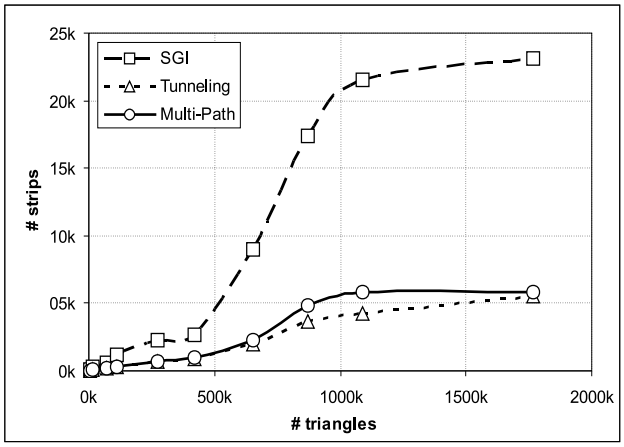


Table 4.2: *Comparison of number of triangle strips in a model (grey cells emphasize the best values, black cells emphasize the worst values).*

We have also tested the distribution of length of triangle strips in the mesh. In Figure 4.3, the distribution of length for the 'happy buddha' dataset is shown (the number of strips of the current length is divided by the total number of strips).

As our new algorithm uses a global criterion, it does not produce a big number of extremely short triangle strips. If we compare *Multi-Path* to the *SGI* method, which uses a local criterion, the difference is apparent. We were surprised by the strip length distribution of *Tunneling*, which also uses a global criterion, but produces a lot of shorter strips. This is probably caused by the SGI-based algorithm, which is used to give a quick initial stripification for *Tunneling*.

method	bunny	dragon
Multi-Path	156	4876
Tunneling [Ste01]	188	3672
Silva [SKP02]	599	16222
SGI [Van02]	601	17399
SGI [AHB90]	705	17653
STRIPE [ESV96b]	917	19935
FTSG [XHM99]	618	20571

Table 4.3: *Comparison of number of triangle strips in a model (for more methods).*

#	model	SGI	Tunneling	Multi-Path
1	cow	8	8	8
2	demi	23	24	24
3	bunny	87	97	95
4	dinosaur	148	159	158
5	ball joint	358	387	387
6	club	532	585	582
7	hand	876	941	921
8	dragon	1237	1261	1254
9	happy buddha	1546	1574	1564
10	blade	2294	2561	2425

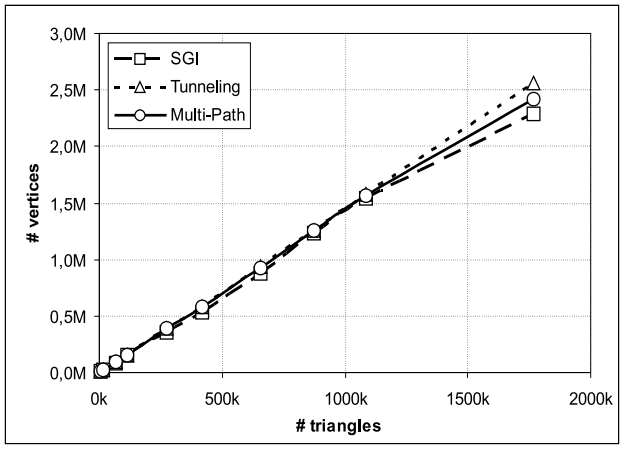


Table 4.4: *Comparison of number of vertices in strips. The number of vertices is in thousands.*

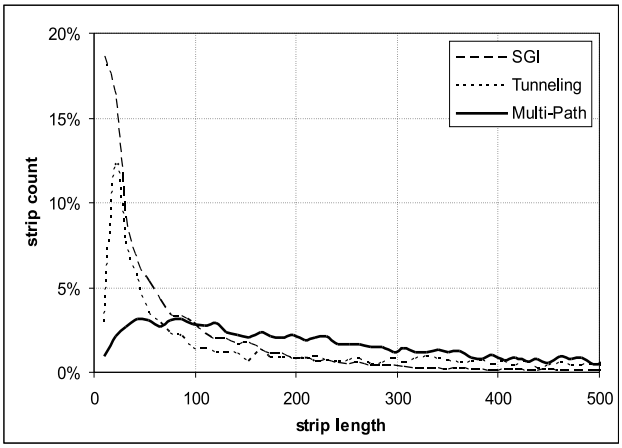


Figure 4.3: *Comparison of strip length distribution for 'happy buddha' (the number of strips of the current length is normalized by the total number of strips).*

Performance

Finally, we have compared runtime of algorithms (Table 4.5). The time includes the allocation of all necessary memory (excluding memory for model itself, i.e., array of vertices and array of indices), construction of all data structures (triangle neighbors, etc.) and the stripification process itself. For the *Tunneling* algorithm, the time for *SGI* initial stripification is also included.

Both *SGI* and *Multi-Path* are linear time algorithm and their running time is significantly lower than *Tunneling*. To create a stripification (including all data structures) of 'blade', which consists of nearly 1.8 millions of triangles, we need less than 18 seconds, on Intel Pentium IV 1.8Mhz.

The *Tunneling* algorithm is very slow, and the speed is not comparable to other algorithms. Although the stripification process is usually a preprocessing, the tunneling algorithm with the default settings is not usable for large models. We also were not able to create a stripification of 'blade' by tunneling on our testing machine, due to a lack of memory, so we have used a different machine (it took more than 1 hours on dual Pentium 4 XEON 2GHz, 1GB RAM, running on Linux).

#	model	SGI	Tunneling	Multi-Path
1	cow	0.1	1.2	0.1
2	demi	0.4	1.6	0.4
3	bunny	0.9	77.4	0.8
4	dinosaur	1.5	128.3	1.2
5	ball joint	3.5	372.5	3.8
6	club	6.2	804.0	4.9
7	hand	7.6	881.0	6.4
8	dragon	9.5	1345.0	8.0
9	happy buddha	11.7	1793.7	10.2
10	blade	26.0	N/A	17.1

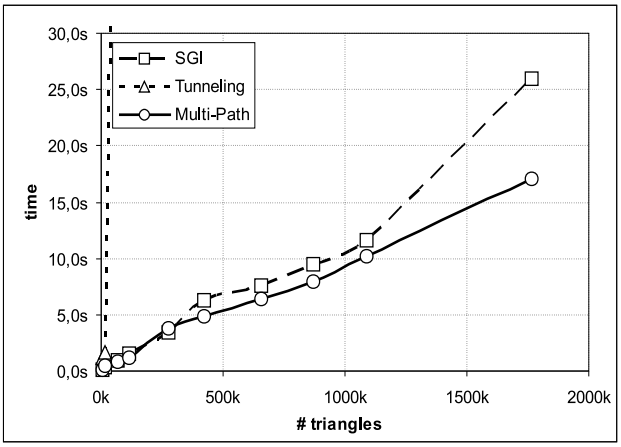


Table 4.5: *Comparison of runtime. Times are in seconds.*

Output examples

Figure 4.4 shows a visual comparison of the ’cow’ and the ’bunny’ model. It is obvious that the *SGI* algorithm (top) produces more triangle strips than the *Tunneling* algorithm (middle) or *Multi-Path* (bottom). As the *SGI* strips are covered by less vertices (i.e., there is a small number of swaps), they are straight and narrow. By contrast, *Tunneling* and *Multi-Path* produce triangle strips that cover huge compact areas of the mesh. Such a behavior can be useful for some additional data reduction, e.g., view-dependent culling of triangle strips.

4.5 Summary

We have designed and implemented a fast and effective method for stripification of static meshes. This method is based on dual-graph and it uses a global criterion for strip creation.

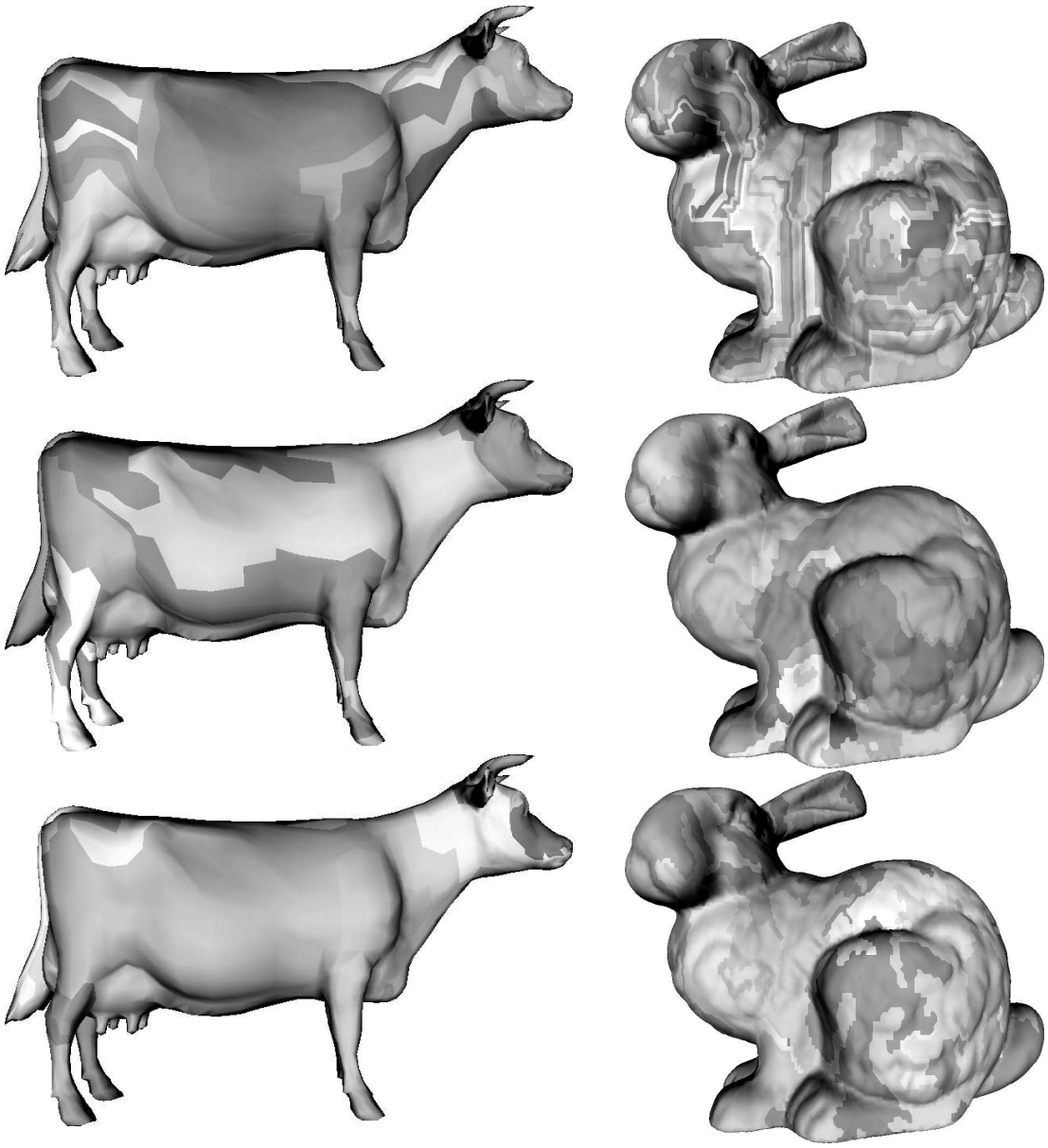


Figure 4.4: *Output examples. The 'cow' has 5804 triangles. It is stripified with 98 strips using SGI (top), 19 strips using Tunneling (middle) and 17 strips using Multi-Path (bottom). The 'bunny' has 69451 triangles. It is stripified with 601 strips using SGI (top), 188 strips using Tunneling (middle) and 156 strips using Multi-Path (bottom).*

It produces a stripification with very low number of triangle strips, but it contains higher number of swaps (i.e., the number of vertices is higher). The algorithm itself is easy to understand and easy to implement.

Probably there is still a place to create even less triangle strips by using the loops, which occasionally appear in the stripification. In the presented algorithm, we remove edges that could lead to such a loop 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.

Extended Multi-Path Stripification

In the previous chapter, we have described a new stripification algorithm based on a dual graph of a triangulation. In this chapter we show, how to extend the algorithm by giving weights to nodes and edges of the graph. We also demonstrate how the extension works on a simple weight function.

5.1 The Extension

The number of strips produced by the Multi-Path algorithm is very low (as far as we know, there is no other linear time algorithm producing such a low number of strips). On the other side, the number of vertices is higher than the number of vertices produced by other algorithms. In this section we propose an extension of the Multi-Path stripification algorithm that allows better control of stripification process and among others it can decrease the number of vertices.

As the Multi-Path stripification is based on a graph algorithm, there is a possibility to influence the stripification process by adding weights to the dual graph. No matter what these weights represent, they can be handled in two ways: the weights are given to the nodes of the graph and the nodes of highest/lowest degree are processed first; or the weights are given to the edges of the graph and the edges of highest/lowest degree are processed first. In both cases, the weights can be either static or dynamic. In our method, we use a combination of all above mentioned cases. We assign the weights to all nodes of triangulation and we dynamically compute the weight of edge as a weight difference of nodes that are connected by this edge.

To include the weight criterion into the stripification process, it is necessary to slightly modify the algorithm described on Figure 4.1. The original algorithm finds the highest possible priority list of nodes and it chooses the first node from the set (see Section 4). To choose the best candidate, we have to traverse the whole set of candidates and find the node that has the lowest weight difference from its neighbor(s). As we have to traverse the whole set of possible candidates in each stripification step, the algorithm complexity is higher than linear. Luckily, the number of candidates is not very high in real life situations. It is also possible to terminate the searching, when the weight difference is equal to zero.

Choosing the Weights

The weight function can be chosen in many different ways and it depends on the feature of stripification we want to improve (CLOD, visibility culling, etc.). This makes our new algorithm very flexible.

To demonstrate how the weights influence the final stripification, we decided to use weights according to the X-coordinates of vertices of triangles (there is no special reason to choose X axis and, as we show in the next section, the direction of the weight function does not influence the quality of stripification too much). Such a criterion should lead

to straight strips coplanar to YZ plane. We supposed that this criterion could decrease the number of vertices in the stripification as the strips are more or less straight (i.e., less swaps is needed). Surprisingly, this criterion also decreases the number of strips as we show in the next section.

Although the above described function produces very good results, the real challenge is to find some topologically based functions for better stripification and functions for stripfication of progressive meshes.

5.2 Tests and Results

The original Multi-Path algorithm, as well as our new extension have been implemented in Borland Delphi 7.0. All experiments were performed on a PC AMD Athlon XP 2800+, 2.1GHz, 512MB RAM, ATI Radeon 9600 with 64MB memory, running on MS Windows XP. Naturally, times of I/O operations have been excluded from measurements. The algorithm was tested on the same set of models as *Multi-Path* algorithm (Table 4.1)

In the next sections, we compare several important factors of stripification. The number of strips and number of vertices in strips shows the compression level. Although the stripification process is usually done in preprocessing stage, the running time is still important for large data sets. To be able to stripify large datasets, low memory requirements are important. The speed of rendering shows the speedup that was achieved. Finally, we show how the direction of weight function axis influences the stripification.

Stripification

As the creation of a new strip is quite a time consuming operation, the number of strips is an important factor of stripfication quality. A comparison of number of strips created by various methods is shown in Table 5.1.

Our new algorithm produces very low number of triangle strips and for many models, it produces the lowest number of strips from all algorithms. For models of hand, happy buddha and dragon, the number of strips is higher than the number of strips produced by tunneling. This fact is caused by the high topological and geometrical irregularity of these models (these models contain large number of vertices of high degree, for visual comparison see the dinosaur Figure 5.1 (a) and the happy buddha Figure 5.1 (b) models).

model	SGI	Tunneling	MPath	EMPath
cow	98	19	17	16
demi	335	137	97	94
bunny	648	188	156	86
dinosaur	1177	267	308	197
balljoint	2279	707	690	381
club	2658	909	978	454
hand	8997	1944	2227	1646
dragon	17399	3672	4876	4380
buddha	21578	4219	5809	5250
blade	23125	5537	5863	4281

Table 5.1: *Number of strips in models (grey cells emphasize the best values, black cells emphasize the worst values).*

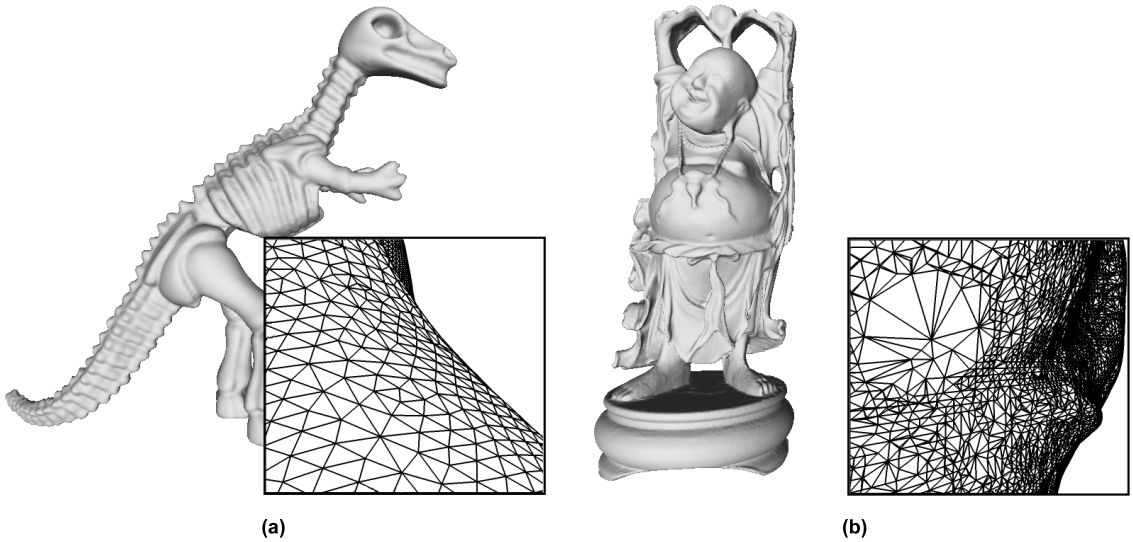


Figure 5.1: *Visual comparison of topological regularity of objects. (a) shows the model of dinosaur which is quite regular, contains only a low number of vertices with a high degree and most of the vertices has degree six, unlike the model of happy buddha in (b) which is highly irregular.*

Vertices

The number of strips is not the only factor that influences the rendering speed. The number of vertices that are beeing sent through the bus is also very important. A comparison of number of vertices is presented in Table 5.2.

model	SGI	Tunneling	MPath	EMPath
cow	8K	8K	8K	8K
demi	23K	24K	24K	24K
bunny	87K	97K	95K	86K
dinosaur	148K	159K	158K	150K
balljoint	358K	387K	387K	355K
club	532K	585K	582K	537K
hand	876K	941K	921K	890K
dragon	1237K	1261K	1254K	1243K
buddha	1546K	1574K	1564K	1552K
blade	2294K	2561K	2425K	2364K

Table 5.2: *Number of vertices in strips.*

Usually, the algorithms that produce a low numbers of strips produce high numbers of vertices, as maintaining a long strip costs some additional swaps. Although our new algorithm produces a low number of strips, it also produces much lower number of vertices than the *Tunneling* or *Multi-Path* algorithm. In the case of very regular models, the number of vertices is close to or even lower than *SGI*.

Running Time

The stripification is usually done in preprocessing stage of the visualization, thus the running time of stripification process is not crucial. On the other hand it should not take too much time. The running times are shown in Table 5.3.

The time includes the allocation of all necessary memory (excluding memory for model itself, i.e., array of vertices and array of indices), construction of all data structures (triangle neighbors, etc.) and the stripification process itself.

As our new algorithms searches for the best possible candidate in each stripification step, the algorithm complexity is higher than the complexity of *Multi-Path*. In comparison to the *Tunneling* the running time of our algorithm is still low and it can be used even for large data. Furthermore, this weak point can be reduced by using some additional data structures such as priority queues or buckets to speedup the best possible candidate choice.

model	SGI	Tunneling	MPath	EMPath
cow	0.16	0.49	0.14	0.37
demi	0.65	0.97	0.50	1.02
bunny	0.69	176.66	0.72	1.08
dinosaur	1.03	72.12	1.81	1.62
balljoint	3.75	176.39	3.46	28.36
club	5.76	629.46	5.96	63.97
hand	5.30	586.63	6.14	20.50
dragon	7.96	1177.27	10.98	276.26
happy	10.31	1580.71	13.37	338.98
blade	17.76	N/A	20.61	151.52

Table 5.3: *The computation time in seconds.*

Rendering Speedup

To measure the rendering speedup, we use the OpenGL display lists. Each vertex is defined by its position (12 Bytes) and normal vector (12 Bytes). To show the benefit of vertex strips, we also present the speed of rendering while drawing unordered triangles and while drawing triangles ordered by Bogomjakov's reordering method [BG01] (for this method, we present the rendering speed achieved by vertex buffer objects that are much faster than display list for this method).

model	Triangles	Bogomjakov*	SGI	Tunneling	MPath	EMPath
cow	542.8	654.7	644.4	651.1	650.1	645.2
demi	456.6	639.6	614.3	618.3	618.0	618.0
bunny	157.0	291.5	336.3	352.2	348.8	341.6
dinosaur	121.1	217.9	269.9	278.2	276.6	276.1
balljoint	54.2	98.7	136.3	139.5	138.8	138.6
club	36.0	95.2	94.1	95.5	95.7	95.1
hand	23.6	48.2	58.6	64.0	64.5	64.7
dragon	3.0	47.2	37.7	48.4	50.1	50.0
happy	2.4	45.4	30.0	38.7	40.5	40.4
blade	1.5	13.1	19.8	24.4	24.4	24.6

Table 5.4: *The average FPS (display lists; *vertex buffer objects).*

While using triangles, for most of the models the framerate is about 2 – 2.5 times smaller than the framerate of stripified objects. This ratio corresponds to the theoretical assumption (strips can reduce the amount of data by a factor of three in the best case). The effect of topology compression is also visible in high resolution models (dragon, buddha and blade); when using the triangle representation, the size of these models exceeds the amount of GPU memory and makes the rendering unbearably slow.

Memory Requirements

As different algorithms use different data structures, the amount of allocated memory can differ (Table 5.5). To measure the memory usage, we wrote a very simple program that scans the running processes and stores the memory usage peak of the stripification process. As the scanning is not continuous, some inaccuracy may appear.

We also include the average bytes per processed triangle on a 32-bit computer. This ratio can be used to compute the maximal size of model that can be stripified on a machine with a given size of memory.

model	SGI	Tunneling	MPath	EMPath
cow	0.5	5.1	2.0	2.0
demi	2.1	11.2	3.4	3.5
bunny	6.8	37.2	9.4	9.7
dinosaur	9.3	57.8	14.4	14.9
balljoint	21.0	137.6	33.3	34.3
club	34.9	209.3	50.0	51.6
hand	53.9	325.0	78.0	80.5
dragon	71.5	435.1	104.4	107.7
happy	88.9	540.8	129.8	133.9
blade	143.3	879.0	207.1	213.8
bytes/tri	86	524	125	129

Table 5.5: *The amount of allocated memory in MB.*

The *SGI* algorithm does not use too many additional data structures and the memory requirements are very low. As our new algorithm does not use any special structures, the memory requirements are nearly the same as the original *Multi-Path* and they are about four times lower than the memory requirements of the *Tunneling* algorithm.

Impact of rotation

As the weight function that we use for the stripification depends on the orientation of the model (or on the orientation of axis of weight function), we have also included a test of impact of rotation transformation. We have rotated the model of bunny and the model of dragon around the Z axis and we have made the stripification for all orientations with five degree step.

The behavior of our new algorithm for regular meshes is shown in Figure 5.2 (the bunny model).

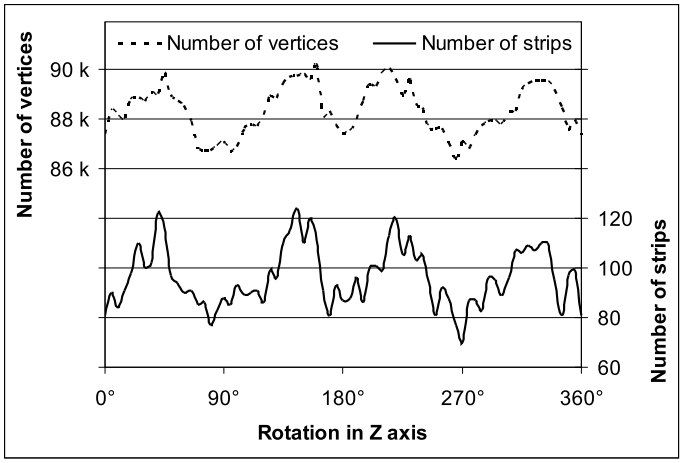


Figure 5.2: *Impact of rotation to the stripification quality. Rotating the model of bunny around Z axis with weight function according to the X coordinate.*

As the vertex distribution in many objects is more or less axis aligned (see Figure 5.3 (a)) the rotation of the object (or of the direction of weight function) by 45 degrees decreases the quality of stripification (the number of vertices as well as the number of strips increases). This fact is well visible on the graph. We also include a visual comparison of two extreme cases; the rotation by 145 degrees produces a stripification with 124 strips and nearly 90k of vertices (Figure 5.3 (b)) and the rotation by 270 degrees produces a stripification with 70 strips and 87k of vertices (Figure 5.3 (c)). Although the differences are quite high, our new method produces better stripification than *Tunneling* or *Multi-Path* even for the worst case.

The behavior of Extended Multi-Path for irregular meshes is shown in Figure 5.4 (dragon model).

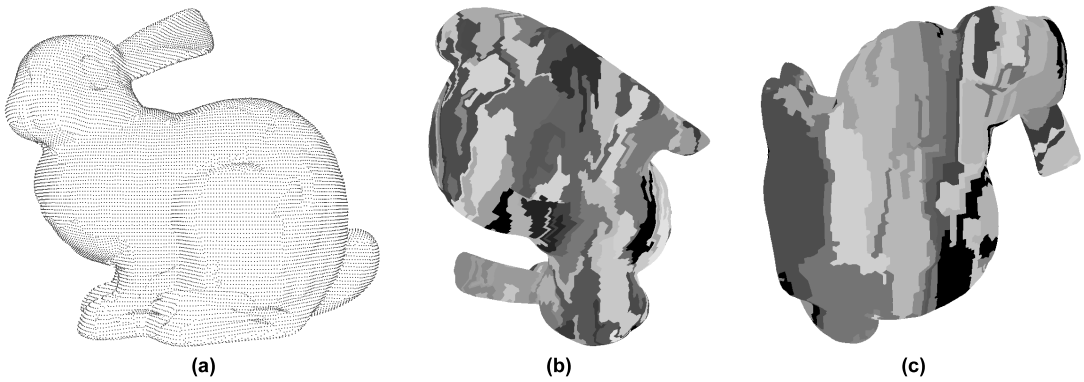


Figure 5.3: *Impact of rotation to stripification quality. Rotating the model of bunny around Z axis with weight function according to the X coordinates cause changes of the result of stripification process. Figure (a) shows the structure of vertices in the model. Figure (b) presents the stripification of model rotated by 145 degrees, which is the worst case, and Figure (c) presents the stripification of model rotated by 270 degrees, which is the best case.*

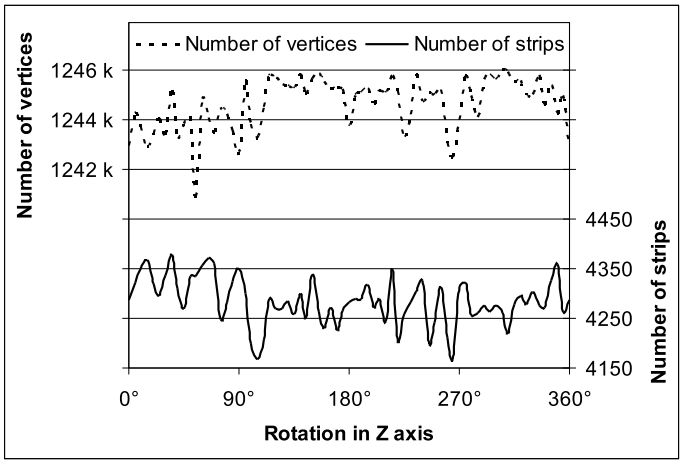


Figure 5.4: *Impact of rotation to stripification quality. Rotating the model of dragon around Z axis with weight function according to the X coordinate.*

In the case of highly irregular meshes, the rotation of the object (or of the direction of weight function) does not influence the quality of stripification too much. The stripification has higher number of strips than *Tunneling*, but lower than *Multi-Path*.

The above tests show that choosing the axis aligned direction of weight function is a good choice in most of the cases, as most of the objects is regular. To get better results, it is important to suggest some function that is based on a topological criterion, or to suggest a specialized function for a concrete type of input data.

5.3 Summary

We have designed and tested a new stripification method based on a Multi-Path Stripification algorithm [VK04a]. Using weights in a dual graph of triangulation, this method allows the user to influence the final stripification.

We have also suggested one possible weight function that can highly improve the quality of stripification, especially for topologically regular triangle meshes. Using this function, our new method produces stripification with very low number of triangle strips (in many cases even lower than *Tunneling* algorithm, which, as far as we know, produces the lowest number of strips from all existing methods), that are covered with lower number of vertices. The algorithm complexity is close to $O(n)$ and it is possible to process large datasets.

Our new algorithm offers a wide area of possibilities for stripification. In the future work, we would like to explore some other weight functions that can produce even better stripfication and that are not so sensitive to topological and geometrical irregularity.

Multi-Path for Tetrahedral Meshes

The importance of computation and visualization of tetrahedral meshes is getting very important in the last years. In this chapter we present a short introduction to tetrahedral strips and we show the modification of Multi-Path *algorithm for tetrahedral strips.*

6.1 Tetrahedral Meshes

For many 3D applications, the real-time rendering is necessary. In many of these applications, only the surface of the visualized scene is necessary. To take the full advantage of modern graphic hardware, the high level primitives such as NURBS or subdivision surfaces are converted to a set of triangles (or triangle strips).

However, there exists large area of applications, where the surface visualization is not sufficient and the complete volume rendering is necessary (medical application, hydro/aerodynamic computation, etc.). Similarly to surfaces, there are two possible ways of volume representation – regular grid, which can be rendered quite easily with the programmable GPUs, but which lacks some properties such as locally adaptive resolution; and tetrahedral meshes, which are more complicated to render, but which are more flexible providing locally adaptive resolution, integration with polygons or fitting to complex boundaries.

Using the full advantage of triangle rendering hardware and optimized algorithms for visibility sorting, the tetrahedral renderers are recently achieving an interactive frame rates using the Projected Tetrahedra algorithm. While tetrahedral meshes are nowadays used mainly for simulation and visualization of vector fields and medical research, the possibility of real-time volume rendering will bring up a huge area of applications such as highly realistic atmospheric effects, high resolution volume sculpturing or visual feedback of simulation of deformations.

6.2 Tetrahedral Strips

Increasing the complexity of scenes, the same problem of insufficient data bandwidth as in triangle meshes arises. King [KCW01] et al. suggested an architecture for tetrahedral volume rendering and an OpenGL API extension to support tetrahedral strips that can decrease the transmission cost of topology. Furthermore, the tetrahedral strip primitive can improve the vertex cache management when rendering tetrahedral meshes on GPUs that support vertex caching and updatable vertex arrays.

Unfortunately, the description of the tetrahedral connectivity is more difficult than in triangle meshes for two reasons:

- It is impossible to use the term 'left-right alternation', as there are no obvious notions of 'left', 'right', or 'alternating' sequence.
- It is not possible to orient tetrahedra incident to a given vertex in a simple way.

To deal with the added dimension of tetrahedral meshes, it is possible to consider simplicial complexes in general. Let's suppose that k-dimensional subcomplexes of n dimensional simplicial complex are equivalent for varying k and n, if $n-k$ is constant. Using this assumption, a vertex of a triangle ($k = 0, n = 2$) is equivalent to an edge of a tetrahedron ($k = 1, n = 3$). This is also true for edges and faces – as every edge in triangle mesh is incident to two triangles at most, each face in tetrahedral mesh is incident to two tetrahedra at most. Now, using an edge as a basic element, it is possible to define a tetrahedral fan and tetrahedral strip:

Tetrahedral Fan

A *tetrahedral fan* is a sequence of tetrahedra which share a common edge. Using the tetrahedral fan, the transmit cost of n tetrahedra can be reduced by the factor of four (from $4 \cdot n$ to $n + 3$ vertices). The important fact is that generally it is not possible to include the entire neighborhood of any vertex, while using a tetrahedral fan.

Although the tetrahedral fan is quite a simple primitive, it is necessary to carefully define the syntax to make it clear which vertices define the common edge. In his suggested extension, King used a consistent notation of the fan as in the OpenGL triangle fan, i.e., the first two vertices define the edge that is shared.

The suggested OpenGL extension adds a primitive GL_TET_FAN_EXT, which can be used in a very similar way as a triangle fan primitive. If passing vertices $v_0, v_1, v_2, ..., v_5$, three tetrahedra ($v_0v_1v_2v_3$, $v_0v_1v_3v_4$ and $v_0v_1v_4v_5$) will be rendered.

```
glBegin(GL_TET_FAN_EXT)
  glVertex(v0)
  glVertex(v1)
  glVertex(v2)
  glVertex(v3) // tetra 0123
  glVertex(v4) // tetra 0134
  glVertex(v5) // tetra 0145
glEnd()
```

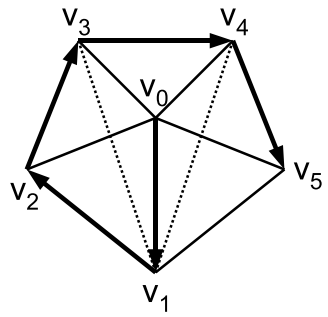


Figure 6.1: *An example of OpenGL API extension and a corresponding tetrahedral fan [KCW01].*

Tetrahedral Strip

A *tetrahedral strip* is a sequence of tetrahedra that are connected by shared faces, but not all of them necessarily share one common edge. Similarly to triangle strips, for the tetrahedral strip, the application has to send four vertices of the first tetrahedron, and then a single vertex for each tetrahedron in the strip.

Like in the case of triangle strips, there exist two categories of tetrahedral strips. In the case of a sequential strip, each four consecutive vertices represent a tetrahedron. To be able to draw a generalized tetrahedral strip, it is necessary to introduce a *swap* like operation. Using the zero area tetrahedron to perform the swap can be very expensive, as it would be necessary to send three vertices. For this reason, King et al. [KCW01] suggested to send a flag indicating which of the four vertices that were processed in the previous step should be replaced.

To have a maximal benefit from the tetrahedral strips, two primitives are suggested to distinguish between the sequential and generalized strip. The simpler primitive is the GL_SEQUENTIAL_TET_STRIP_EXT. The use of this primitive is very easy and intuitive – the first four vertices specify the first tetrahedron and each following vertex replaces the first vertex of the previous tetrahedron. The primitive GL_GENERAL_TET_STRIP_EXT is more complicated, as the programmer has to send a flag to choose which vertex should be replaced (GL_REPLACE_VERTEX_EXT_1,2,3,4) by calling a glReplaceVertexEXT function. This extension makes the general tetrahedral strip primitive very flexible.

```
glBegin(GL_SEQUENTIAL_STRIP_EXT)
  glVertex(v0)
  glVertex(v1)
  glVertex(v2)
  glVertex(v3) // tetra 0123
  glVertex(v4) // tetra 1234
  glVertex(v5) // tetra 2345
glEnd()
```

```
glBegin(GL_GENERAL_STRIP_EXT)
  glVertex(v0)
  glVertex(v1)
  glVertex(v2)
  glVertex(v3) // draws tetra 0123
  glReplaceVertexEXT(
    GL_REPLACE_VERTEX_EXT_1)
  glVertex(v4) // draws tetra 1234
  glReplaceVertexEXT(
    GL_REPLACE_VERTEX_EXT_3)
  glVertex(v5) // draws tetra 1245
glEnd()
```

Figure 6.2: *An example of OpenGL API extension and a corresponding tetrahedral sequential strip and generalized strip [KCW01].*

6.3 Existing Stripification Methods

As far as we know, King et al. [KCW01] proposed the first tetrahedral stripification algorithm. It is based on a simple greedy heuristic algorithm which is equivalent to the *SGI* stripification for triangles. They choose the first unvisited tetrahedron as the start of the strip. Then, they choose one of its neighbors as the next member of the strip and repeat this step. If there is no other neighbor, the algorithm returns back to the first tetrahedron of the strip, and extends the strip by choosing another neighbor. This process is repeated as long as there are any unvisited tetrahedra in the mesh. This basic algorithm produces a stripification with the mean strip size of 9-14 tetrahedra. To improve the greedy algorithm, they also used several heuristics:

1. Choosing randomly.
2. Choosing the tetrahedron with the fewest unvisited neighbors.
3. Choosing a sequential order first.
4. Attempt to create a fan first, then switch to another variant

According to the results, method 2 produces the best stripification, which is in fact an extension of the *SGI* algorithm for tetrahedral meshes. The average length of the strips increases up to 49 tetrahedra.

Similar stripification method is presented by Weiler et al. [WMKE04]. For unknown reason, they report an average strips length of 10 tetrahedra for fewest unvisited neighbor strategy and length of 5 for sequential strips.

6.4 Modification of Multi-Path Algorithm

Similarly to triangle meshes, we can have a dual graph of tetrahedral mesh. Using the simplicial complex generalization, we obtain a graph, where each node corresponds to a tetrahedron and tetrahedra that share a face are connected by an edge in the graph. The main difference is that each node can have four neighbors instead of three.

To modify the Multi-Path algorithm for tetrahedral meshes, we have to add two new classification sets – U_4 for unconnected tetrahedra with four neighbors; and C_3 for connected tetrahedra with three neighbors (see Chapter 4). These two sets have the lowest

priority and in fact, the tetrahedra from these groups are used very rarely, as each tetrahedral mesh has a boundary in E^3.

6.5 Test and Results

We compared our method to both known tetrahedral stripification algorithms [KCW01, WMKE04]. King [KCW01] suggested four different stripification methods (we denote them as *m1–m4*) and Weiler [WMKE04] suggested two methods: for sequential strips and for generalized strips (we denote them as *seq* and *gen*). As we did not obtain the data sets from King and Weiler, we present the results of stripification of random datasets with uniform distribution of about the same size (for models presented in [KCW01], we have only an estimation of number of tetrahedra from the number of strips and the average length of strips).

In the Table 6.1 we present the comparison of number of strips in a model. Similarly to triangle stripification, our method produces a stripification with much lower number of strips than other methods.

models			[KCW01]				[WMKE04]		M-Path
name	tetrahedra	vertices	m1	m2	m3	m4	seq	gen	
bracket	∼3418		367	222	398	349			
	3349	550							18
phoenix	12936	20108	1000	441	1231	882			122
langley	70125	13832	4745	1432	4907	3763	16169	7274	
	70318	10600							483
spx	103488	37320					24009	11161	
	106678	35000							1561
f117	∼240000		22203	6504	20578	10030			
	243751	80000							3563

Table 6.1: *Comparison of number of tetrahedral strips in a model.*

To show the distribution of length of tetrahedral strips, in the Table 6.2 we present the average length of strips and maximal length of strips in a model (both values are presented in one cell, separated by slash).

models			[KCW01]				[WMKE04]		M-Path
name	tetrahedra	vertices	m1	m2	m3	m4	seq	gen	
bracket	∼3418		440/9	119/15	188/9	148/10			
	3349	550							847/186
phoenix	12936	20108	182/13	347/29	962/11	293/15			522/106
langley	70125	13832	370/15	865/49	3486/14	676/19	281/4	2249/10	
	70318	10600							901/146
spx	103488	37320					281/4	2235/9	
	106678	35000							413/68
f117	∼240000		404/11	1472/37	6238/12	2229/24			
	243751	80000							508/68

Table 6.2: *Comparison of the longest and average length of the strips in a model (both values are presented in one cell, separated by slash).*

Both existing algorithms produce a few long strips and a lot of short strips or even isolated tetrahedra. Such a distribution of length of strips is not very good. Our algorithm does not produce these long strips, and the length of all strips in the stripification is more equal. Our stripification also usually tends to avoid the isolated tetrahedra and the shortest strip in most of the cases is the connection of two tetrahedra.

6.6 Summary

As the problem of hardware volume visualization is very important in many computer graphics applications, it is very probable that some kind of tetrahedral topology compression and reordering will be necessary.

We proposed a modification of our Multi-Path algorithm for stripification of tetrahedral meshes. In comparison to existing algorithms ([KCW01, WMKE04]), our algorithm produces much lower number of tetrahedral strips. Unfortunately, we are not able to make more comparisons, as the topic of tetrahedral stripification is quite new and there is no hardware on which we can measure the rendering speedup of our tetrahedral strips.

Quadrilateral Meshes Stripification

In this chapter a new algorithm for stripification of purely quadrilateral meshes is described. The algorihtm is based on the SGI *algorithm. Creating strips of quads and splitting them into triangle strips afterwards can significantly improve the quality of stripification and increase the rendering speed.*

7.1 Quadrilateral Meshes

Quadrilateral meshes are nowadays very often used to store and visualize various geometric objects in many applications such as computer games and movie industry (subdivision surfaces [ZS00]), medical and scientific visualization (volume rendering, surface reconstruction from slices [SS04]), etc. In many of these applications a real time visualization is required. The speed of todays' high performance rendering engines is very often bounded by the rate at which the data is sent into the machine. Furthermore, most of the rendering engines can handle only triangle faces, thus the number of primitives increases.

7.2 Triangle and Quad Strips

To draw an independent set of i quads (quadrilaterals), we need to transmit $4i$ vertices. To reduce the amount of transmitted data, it is possible to split the quads into two triangles and connect them into triangle strips (or tristrips). In some graphic libraries a special type of primitives used for quads can be found (e.g., OpenGL). Rendering of quad strips is usually slower than rendering of triangle strips and the number of vertices is equal or higher than the number of vertices using tristrips (as we show next).

A *sequential triangle strip* can represent j quads with $j + 4$ vertices: in Figure 7.1 (a) the sequence (1,2,3,4,5,6) represents quads $\square 1243$ and $\square 3465$ (or triangles $\Delta 123$, $\Delta 324$, $\Delta 345$ and $\Delta 546$). A *sequential quad strip* is a sequence of $j + 4$ vertices that represents j quads: in Figure 7.1 (b) the sequence (1,2,3,4,5,6) represents quads $\square 1243$ and $\square 3465$.

(a) (b)

Figure 7.1: *An example of sequential triangle strip (a) and a sequential quad strip (b).*

In general situations, the quad adjacency must not allow a sequential encoding. In Figure 7.2 (a) the sequence (1,2,3,4,5,6,7,8) produces an invalid triangle $\Delta 567$. An extra vertex has to be added to change the sequence to (1,2,3,4,5,4,6,7,8). Using quad strip, the situation is worse. In Figure 7.2 (b) the sequence (1,2,3,4,5,6,7,8) produces an invalid quad $\square 5687$. To avoid this situation, it is necessary to make a *swap* at a cost of three additional vertices, i.e., a sequence (1,2,3,4,5,6,6,6,6,4,8,7).

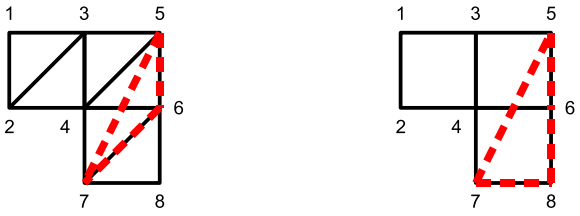


Figure 7.2: *An example of a generalized triangle strip (a) and a generalized quad strip (b).*

From the above example it is obvious that triangle strips are more general and more efficient than quad strips. For this reason we concentrate on triangle strips only. There are two possibilities how to construct triangle strips from not fully triangulated meshes. The first approach is to use some algorithm that triangulates the faces and then any stripification algorithm can be used. This way is general and can be used for any type of polygonal meshes. The main disadvantage of this approach is that it does not profit from the fact that the polygon can be triangulated arbitrarily. The other approach searches for strips in the untriangulated model and triangulates faces on the fly. Such an approach often leads to a better stripification.

To obtain a good stripification we decided for the second approach. Detailed description of our new algorithm is presented in the next section.

7.3 QStrip Algorithm

Our new algorithm (*QSTRIP*) is designed for meshes that are fully quadrilateral. It is based on a similar idea as the *SGI* algorithm for triangle meshes. As we are not working on a triangulated mesh, first we construct strips of quadrilaterals. Then we sequentially traverse these strips and triangulate the quadrilaterals with respect to the triangle strips.

In the first step, the algorithm chooses a quadrilateral with a low number of neighbors to start a new strip. This choice minimizes the number of short strips. In Figure 7.3 (a), the stripification process started in a quadrilateral with two neighbors and an isolated quadrilateral Q appeared. Usually we can avoid such a situation by starting from a quadrilatral with low a number of neighbors 7.3 (b).

The chosen quadrilateral is removed from the mesh and it is inserted into the strip. The mesh is locally updated to reflect the quadrilateral removal. Now the algorithm chooses

(a) (b)

Figure 7.3: *An example of a bad (a) and a good (b) choice of the starting quadilateral.*

a neighboring quadrilateral that will be adjacent in the strip. To decrease the number of vertices in the final stripification, the algorithm preferentially chooses a quadrilateral that does not produce a swap. The chosen quadrilateral is again removed from the mesh and inserted into the strip. These steps are repeated as long as it is possible (i.e., as long as there is a neighboring quadrilateral). If the mesh still contains some quadrilaterals, a new strip is started. A pseudo-code of this algorithm is presented in Figure 7.4.

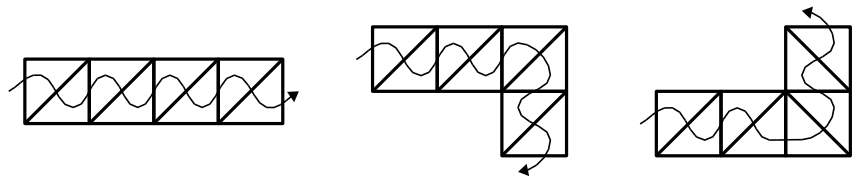
```
input:  list of quads
output:  strips of quads

while there is any quad in the mesh do
  start a new strip
  choose a quad with the lowest number of neighbors
  add the quad to the current strip
  remove the quad from the mesh
  locally update the mesh
  while there exists a neighbor of the current quad do
    choose a neighboring quad that does not produce a swap
    if such a quad does not exist then choose arbitrarily
    add the quad to the current strip
    remove the quad from the mesh
    update the mesh
  end while
end while
```

Figure 7.4: *Pseudo-code of the algorithm.*

The algorithm complexity is $O(s \cdot q + q)$, where q is the number of quads and s is number of strips in the final stripification, as we need $O(q)$ steps to find the starting quad for each strip. To speed up this algorithm, we use a priority queue for finding the quad with the lowest number of neighbors. Using such a structure decreases the complexity of finding the starting quad to $O(1)$, and the algorithm complexity is reduced to $O(s + q)$.

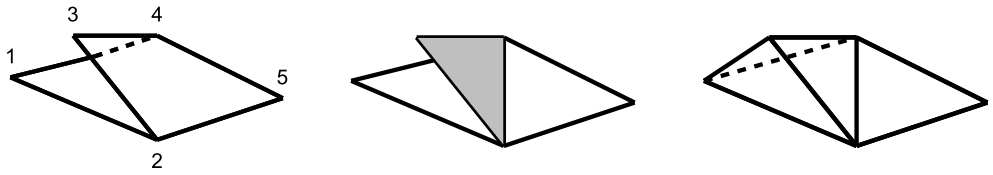
After the stripification phase, it is necessary to decompose the lists of quads into vertices of triangle strips. To provide a correct (counter-clockwise) orientation of triangle strips in the final mesh, it is necessary to start the first triangle of the strip in a counter-clockwise manner. This determines the diagonal of the first quad. As we cannot choose the first diagonal, three different situations can appear. In Figure 7.5 (a) a sequential strip for four quads is shown. If the sequence of quads is not straight, a strip is preserved at a cost of one swap (Figure 7.5 (b)) or two swaps (Figure 7.5 (c)).



(a) (b) (c)

Figure 7.5: *A straight sequence of quads can be covered by a sequential strip (a). To preserve a strip in a non-straight sequence of quads, it is necessary to use one swap (b) or two swaps (c).*

As the input meshes are fully 3D, in some cases it is not possible to split the quad arbitrarily, otherwise an incorrect triangle appears. Such a situation appears when two quads are neighboring via two edges (see Figure 7.6 (a), quadrilaterals $\square 1234$ and $\square 5432$). To avoid an incorrect triangle (Figure 7.6 (b)), at least one of the quads has to be split along the diagonal that starts in the vertex that is not common for these two quads (Figure 7.6 (c)). Respecting this criterion may lead to more swaps in the final stripification. Luckily this situation does not appear very often in a real life model.



(a) (b) (c)

Figure 7.6: *When two quads ($\square 1234$ and $\square 5432$) have two common edges (a), an incorrect triangle may appear (b); the incorrect triangle is grey colored. To avoid it, at least one of the quads has to be split along the diagonal that starts in the vertex that is not common for these two quads (c).*

7.4 Experiments and Results

Our new algorithm has been implemented in Borland Delphi 7.0 as a part of a program for surface reconstruction from orthogonal slices (the reconstructed mesh is purely quadrilateral). The experiments were performed on a PC INTEL Pentium 4, 2.8 GHz, 2 GB of RAM, ATI FireGL T32 graphic card, running on MS Windows XP.

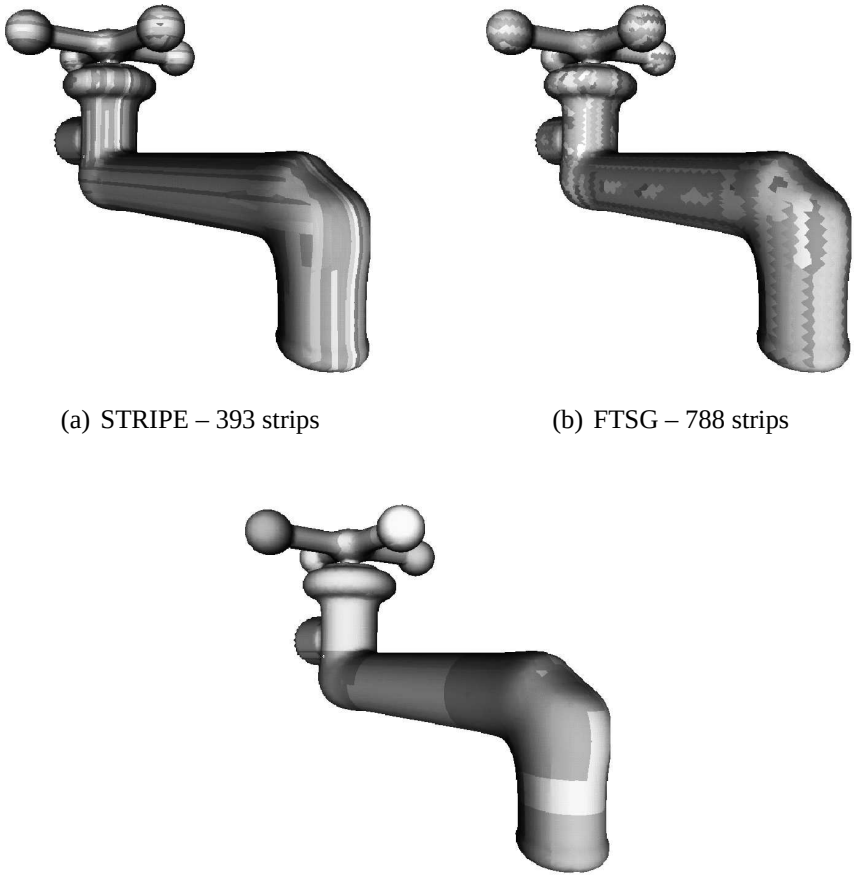
As our algorithm is designed specially for quad meshes, the quality of stripification is very high. We have compared our stripification algorithm with the *STRIPE* v.2 [Eva98], which is also designed for a quadrilateral meshes, and with the *FTSG* [XHM99], which can handle non-triangulated meshes. Both algorithms were compiled with *gcc/cygwin* compiler.

A comparison of stripification methods is presented in Table 7.1. In the first two columns the number of vertices and the number of quads of the tested model is presented. In the next columns the number of strips and number of vertices (including swaps) obtained by the tested algorithm is shown.

		STRIPE		FTSG		QSTRIP	
vertices	quads	strips	vertices	strips	vertices	strips	vertices
2112	2114	88	5101	113	5675	4	4903
4000	4002	111	9517	258	10955	33	9516
8240	8236	140	17096	293	21549	8	17922
12592	12588	391	29050	664	33725	32	28290
16288	16290	393	36570	788	43333	37	36558
25712	25714	570	57181	1084	67931	49	57386
36264	36266	817	79957	1522	95801	44	80078
41919	42005	1356	98276	2405	112840	102	95998

Table 7.1: *Comparison of stripification methods. For each method the number of strips and the number of vertices in strips (including swaps) is presented (grey cells emphasize the best values, black cells emphasize the worst values).*

Our new method produces more or less the same number of vertices as *STRIPE*, but usually it covers the mesh by much smaller number of strips (especially for larger models). Although the *FTSG* method produces a very good stripification for fully triangulated models, for quadrilateral models it produces stripification with very high number of vertices and strips in comparison to the *STRIPE* or the *QSTRIP*. The main reason for this big



(a) STRIPE – 393 strips (b) FTSG – 788 strips

(c) QSTRIP – 37 strips

Figure 7.7: *Visual comparison of stripification of a model of a tap (16290 vertices).*

difference is that the *FTSG* makes a triangulation of the quadrilateral mesh first and then it stripifies the triangulated model. The *STRIPE* algorithm did not surprisingly create large parches but usually it created long sequential strips of quads (see Figure 7.7 (a)). As these strips do not contain swaps, the number of vertices in the final stripification is comparable to our new algorithm although the *STRIPE* contains much higher number of strips. A visual comparison of the tested algorithms is presented in Figure 7.4. The tap model contains 16288 vertices and 16290 quads.

As the *STRIPE* algorithm outputs the stripification during the stripification process, it is not possible to exclude the time of I/O operations. For this reason we have included the time of I/O operations in all measurements, which can arise significant errors. To minimize these errors, all time measurements were performed five times and the minimal time is presented. Such a measurement can be a bit unfair to the *STRIPE* algorithm, as the

write operation is not continuous, but on the other hand it is the real time that is needed for stripification. The comparison of running times is published in Table 7.2.

vertices	quads	STRIPE	FTSG	QSTRIP
2112	2114	0.31	0.14	0.25
4000	4002	0.51	0.25	0.30
8240	8236	8.69	0.70	0.39
12592	12588	15.18	0.97	0.51
16288	16290	10.20	1.31	0.60
25712	25714	21.79	2.06	0.84
36264	36266	50.81	2.89	1.11
41919	42005	83.58	3.10	1.28

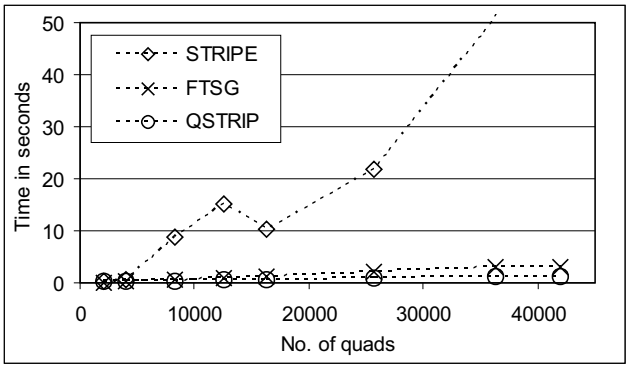


Table 7.2: *Comparison of running times (in seconds). For each method the running time (including I/O operations) is presented.*

The running times of the *FTSG* are comparable to the *QSTRIP*. The difference in the running times can be partially caused by the cygwin emulation, as some functions have to be called from the cygwin dynamic library, but the main reason is probably the dynamic programming part of the *FTSG* algorithm.

The most time consuming step in the *STRIPE* algorithm is the global analysis which searches for the patches. As this global analysis searches the longest possible sequence of quads in both directions for each quad, it has $O(n^2)$ complexity for fully quadrilateral meshes.

In the last table (Table 7.3) we present the average frame rate (FPS) for models stripified by the tested methods and a ratio of this frame rate to the frame rate for models rendered

with quads. For the measurement we used OpenGL and vertex buffer objects (VBO) as they are preferred in new GPUs [NVI03]. When using VBO, a sequential list of vertices is sent to the GPU (i.e., for each quad, four vertices are sent, for each triangle, three vertices are sent – 6 vertices for a quadrilateral face – and for each strip all vertices including swaps are sent).

no. of	QUADS		TRIS		STRIPE		FTSG		QSTRIP	
quads	FPS	ratio	FPS	ratio	FPS	ratio	FPS	ratio	FPS	ratio
2114	234	1.00	229	0.98	252	1.08	238	1.02	240	1.03
4002	291	1.00	261	0.90	305	1.05	312	1.07	320	1.10
8236	229	1.00	192	0.84	285	1.25	276	1.21	283	1.24
12588	180	1.00	147	0.82	220	1.22	214	1.19	224	1.25
16290	166	1.00	130	0.79	207	1.25	200	1.21	218	1.31
25714	125	1.00	95	0.76	167	1.33	159	1.27	176	1.40
36266	99	1.00	73	0.74	139	1.40	130	1.32	146	1.48
42005	80	1.00	60	0.75	111	1.39	100	1.25	111	1.40

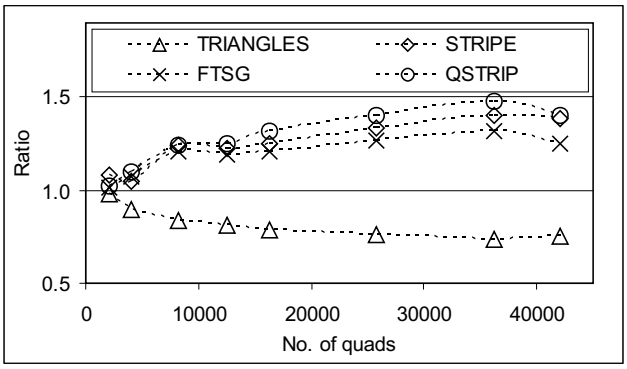


Table 7.3: *Comparison of frame rate achieved with models rendered with quads, triangles and tested stripifications.*

As for a triangle mesh we have to send 1.5 times more vertices than for a quadrilateral mesh, the frame rate is much lower even though rendering a triangle primitive is faster than rendering a quadrilateral.

Although the number of vertices using triangle strips is nearly twice smaller than the number of vertices when using quads, the speed up is not twice higher. The reason is similar to the quad vs. triangle speed up (e.g. drawing the triangle strip primitive is more time consuming than drawing the quad).

The comparison of frame rates of individual stripification methods did not get any surprising results. The *FTSG* produces a stripification that is rendered at the lowest frame rate as it contains high number of vertices and strips. The stripification produced by our algorithm runs at highest frame rate as the number of vertices and strips is low. Although *STRIPE* produces a stripification of nearly the same number of vertices as *QSTRIP*, the frame rate is in the middle between *QSTRIP* and *FTSG*. This is caused by the fact that *STRIPE* stripification contains higher number of strips and starting a new triangle strip costs some extra time.

In all these tests our new algorithm reached the best results. These tests are a bit unfair to *STRIPE* and *FTSG* as these algorithms can handle more general type of meshes, on the other side as far as we know, there is no other algorithm designed for fully quadrilateral meshes, thus we have chosen the best existing algorithms.

7.5 Summary

We have designed and implemented a new stripification algorithm for quadrilateral meshes. As we know the mesh structure, we can exploit it and produce a high quality stripification. In comparison to other methods that can stripify not fully triangulated meshes, our new algorithm produces a stripification with lower number of strips and vertices.

There is still some place to improve the quality of stripification. One of possible ways for the future work is to investigate the behavior of vertex caches that are implemented in todays GPUs and adapt the stripification to maximize the benefit of the cache.

Stripification and Topology

In this chapter we present an overall comparison of several most important stripification algorithms. The tests are performed on a set of real life models as well as on a set of artificial objects. We also show, how the triangle connectivity influences the stripification process.

8.1 Comparison

In this section we introduce a comparison of several algorithms that create a stripification without changes in topology. We have chosen the algorithms that are somehow important or interesting and their implementations are freely available on the Internet (or they can be obtained via email). For all tests, we have used PC INTEL Pentium 4, 2.8GHz, 1024MB of RAM, running on MS Windows XP with ATI FireGL T32 GPU (32MB).

Table 8.1 presents an overview of the tested algorithms. The first column shows the short name of the algorithm as it was presented in this paper. As the programming language and the compiler can influence the speed of the program, the second column ("Compiler") shows the used compiler. A short name under which the algorithm will be presented in tables is shown in the column "Label" (last character of the label distinguishes the strip minimizing algorithm – S; and the vertex minimizing algorithm – V). For all algorithms we have used the default parameters or parameters that were recommended by the authors. The concrete parameters are mentioned in the column "Parameters". Very often, the algorithm has implemented both the vertices minimizing function and the strips minimizing function.

Algorithm	Compiler	Label	Parameters	Minimizing
SGI	Delphi	SGS	-LNLN	strips
		SGV	-LS	vertices
MStrip	Cygwin, gcc	MSS	-m 2	strips
		MSV	-m 2 -q	vertices
STRIPE	Cygwin, gcc	STS	-l	strips
		STV	-q	vertices
FTSG	Cygwin, gcc	FTS	-dfs -concat -sgi	strips
		FTV	-dfs -concat -alt	vertices
Tunneling	Cygwin, gcc	TUS		strips
Multi-Path	Delphi	MPS		strips
Extended Multi-Path	Delphi	EMS		strips

Table 8.1: *Algorithms overview.*

- *SGI – SG(S/V)*: Although the original *tomesh.c* code is available on the Internet [AHB], we have used our own implementation of this algorithm in order to experiment with various heuristic functions [Van02]. We have used the standard SGI method (-LNLN) and vertex minimizing heuristic (-LS) for the tests.

- *MStrip – MS(S/V)*: The source code of the program is available on the Internet [SKP] under the GNU General Public License. The number of the simultaneous strips was set to two which produces the best results in most cases. The measurement for both heuristics (strips minimizing, vertices minimizing (-q)) was performed.

- *STRIPE – ST(S/V)*: The source code of the program is freely available on the Internet [Eva98] for non-commercial use. The mesh is exported during the stripification process, thus it is not possible to exclude the time of I/O operations. 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 – FT(S/V)*: The program is free for non-commercial purposes only and it can be obtained via e-mail [Xia]. 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).

- *Tunneling – TUS*: The program is not available on the internet, but it can be obtained via e-mail from the author. The tests were performed with the default settings. The program does not contain any vertex minimizing heuristic.

- *Multi-Path – MPS*: The program is available on request. The version we used for tests does not have any parameter that can influence the stripification quality. It minimizes the number of strips.

- *Extended Multi-Path – EMS*: Extension of Multi-Path algorithm is available on request. As well as Multi-Path algorithm, it minimizes the number of strips.

All experiments were performed on a PC INTEL Pentium 4, 2.8GHz, 2GB RAM, ATI T32, running on MS Windows XP. Naturally, times of I/O operations have been excluded from measurements (except *STRIPE*).

8.2 Real Models

For our tests, we have used the same set of models as in previous chapters (Table 4.1). These models are freely available for non-commercial purposes and they can be found in many papers on stripification.

Vertices

In Table 8.2, 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 bounds. 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 Figure 8.1 shows a comparison of vertices per triangle, i.e., the ratio of number of vertices to the number of triangles.

algorithm	SGI		MStrip		STRIPE		FTSG		Tunnel	M-Path	
model	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
cow	8K	7K	8K	7K	9K	7K	7K	7K	8K	8K	8K
demi	23K	22K	23K	22K	25K	22K	23K	22K	24K	24K	24K
bunny	87K	82K	86K	82K	102K	83K	85K	82K	99K	94K	86K
dinosaur	148K	139K	147K	140K	164K	141K	145K	140K	159K	159K	150K
balljoint	358K	338K	355K	341K	401K	343K	346K	340K	386K	387K	355K
club	532K	505K	527K	507K	613K	512K	522K	508K	581K	582K	537K
hand	876K	812K			966K	825K	856K	824K	939K	921K	890K
dragon	1237K	1130K			1290K	1153K	1196K	1156K	1261K	1254K	1243K
happy	1546K	1410K			1609K	1439K	1492K	1443K	1574K	1563K	1552K
blade	2294K	2135K			2609K	2160K	2249K	2166K	2542K	2426K	2364K

Table 8.2: *Number of vertices in strips (in thousands; grey cells emphasize the best values, black cells emphasize the worst values).*

The vertices minimizing algorithms — *SGI (SGV)*, *STRIPE (STV)*, *MStrip (MSV)* and *FTSG (FSV)* — produce nearly the same number of vertices. The average V/T for these algorithms is about 1.25. The *SGI (SGV)* 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 compensated by a huge number of strips.

The *STRIPE (STS)* algorithm produces a stripification with an average V/T about 1.47. In our 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).

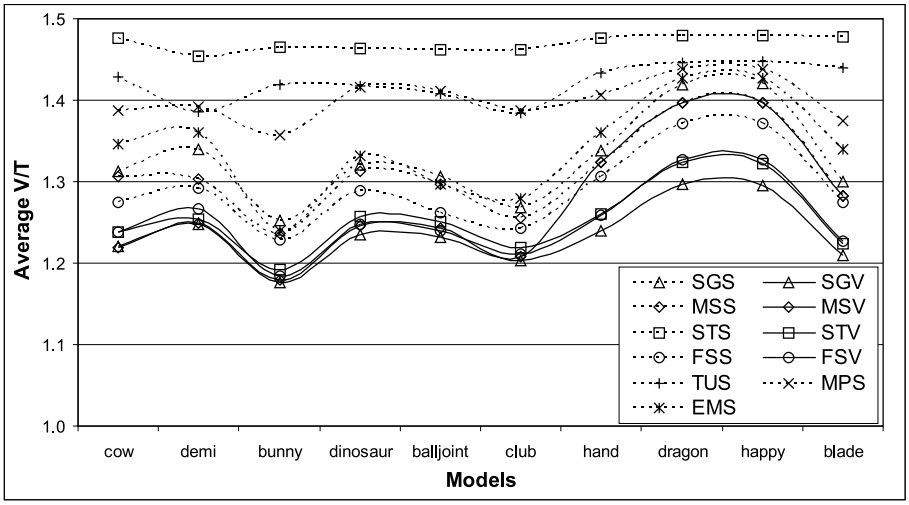


Figure 8.1: *Graph of average number of vertices in strips per triangle.*

It is quite interesting that nearly all algorithms, except *Tunneling (TUS)* and *STRIPE (STS)*, have the same behavior. For the bunny, club and blade model (which have nearly a regular structure), 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 (Figure 8.2).

The results of *MSTRIP* algorithms are not published for all models, as the algorithm did not work well on the Windows platform (for high resolution models, the program crashed).

Strips

The number of strips produced by the tested algorithms are presented in Table 8.3. 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 Figure 8.2. As mentioned earlier, nearly all algorithms have similar behavior which depends on topological regularity of the mesh.

The *Tunneling (TUS)*, *Multi-Path (MPS)* and *Extended Multi-Path (EMS)* algorithm produce more than three times lower number of triangle strips than all other algorithms. On the other side, to obtain such long triangle strips, it is necessary to use swaps (thus all these algorithms produce higher number of vertices).

algorithm	SGI		MStrip		STRIPE		FTSG		Tunnel	M-Path	
model	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
cow	98	352	78	127	141	136	105	312	19	17	16
demi	335	1183	293	419	456	418	286	1020	139	99	94
bunny	648	3560	575	1174	1531	1229	618	3238	166	154	86
dinosaur	1177	7276	1271	2422	2470	2498	1346	6411	260	324	197
balljoint	2279	17454	2519	5746	6145	5820	2446	15371	536	705	381
club	2658	23966	3111	7782	9210	8184	3054	21148	750	963	454
hand	8997	44710			15309	15422	10394	38779	1590	2166	1646
dragon	17399	71182			22928	25356	20571	58377	3331	4832	4380
happy	21578	88143			28563	31550	25576	72271	3710	5845	5250
blade	23125	115568			41128	35952	26779	99890	4606	5902	4281

Table 8.3: *Number of strips achieved by the tested algorithms.*

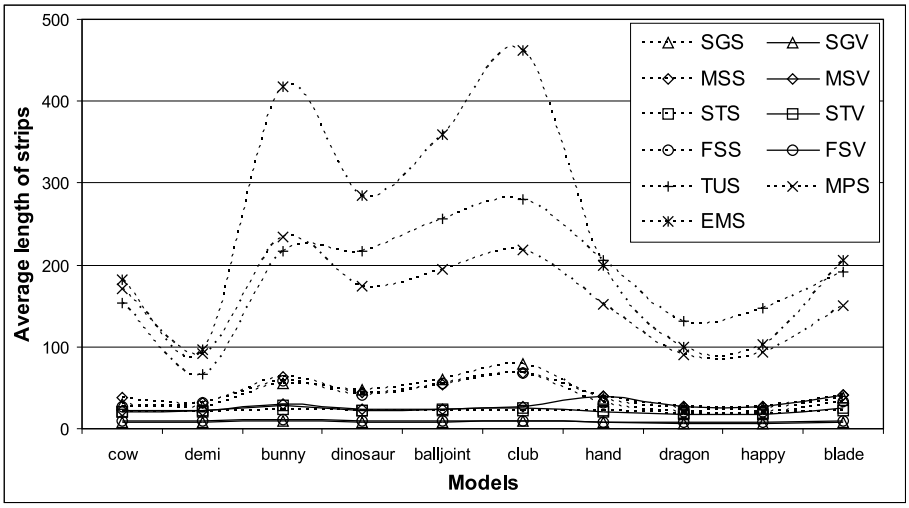


Figure 8.2: *Graph of average length of strips.*

The differences in the number of strips are very high. The *SGI (SGS)* algorithm produces stripification with more than 20 times higher number of strips than the *Tunneling (TUS)* or *Extended Multi-Path (EMS).*

Rendering Speed

As the triangle strips are mainly used to speed up the visualization, we have also tested the rendering speed of models stripified by different techniques (Table 8.4 and 8.5; the speed is stated in FPS – frames per second). For the comparison, we have used two different techniques: display lists, which are precompiled and cannot be modified during the running

time; and vertex buffer objects that allow modifications [NVI03]. We also present the rendering speed for original mesh (TRI) and the rendering speed for mesh that is reordered to have a better ACMR by Bogomjakov's method [BG01](BOG).

algorithm	Tris		SGI		MStrip		STRIPE		FTSG		Tunnel	M-Path	
model	TRI	BOG	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
cow	319.2	361.5	358.0	350.8	353.8	355.8	358.6	355.9	354.5	344.7	359.5	360.3	358.0
demi	299.8	361.2	358.9	351.1	357.9	353.6	358.8	353.7	357.3	347.9	360.4	360.3	358.8
bunny	122.8	162.3	231.7	217.7	226.3	221.4	236.6	221.5	222.8	210.1	240.5	234.8	229.4
dinosaur	116.9	117.3	228.1	216.2	227.2	224.4	227.7	224.9	226.0	216.5	231.4	231.1	229.8
balljoint	56.7	57.2	134.5	125.1	134.1	126.2	134.1	131.6	133.0	124.9	136.9	136.9	135.9
club	39.1	39.2	98.4	91.6	98.1	95.8	97.9	96.4	97.4	91.5	99.8	100.0	99.0
hand	26.0	26.0	69.0	62.7			68.9	67.5	67.8	63.0	71.1	70.9	70.7
dragon	9.8	9.4	55.0	48.1			54.4	53.0	53.5	48.8	56.7	56.6	56.4
happy	7.8	7.5	45.4	39.5			44.8	43.6	44.0	40.1	46.6	46.7	46.5
blade	4.8	4.6	28.4	25.8			28.3	27.2	27.9	25.8	29.3	29.1	29.0

Table 8.4: *The average FPS using display lists.*

While using the display lists, the commands are compiled from OpenGL's high-level command language into low-level hardware commands and stored in the memory. Whenever the list is being drawn, these precompiled data is used to save a lot of function calls and compilations. This rendering method is very fast, but it can be used for static data only. It also does not use the vertex caches, as vertices are defined by full coordinates and not by table of vertices and indices to this table (this is the reason, why the speed of randomly ordered triangles and cache friendly ordered triangles is nearly the same).

As shown in the Introduction, to draw a set of n independent triangles, it is necessary to send $n * 3$ vertices. As we use the lighting, we also need $n * 3$ normals. Each vertex coordinate and normal is defined by three floating point numbers (on 4 bytes). Thus, the real size of data is: $n * (3 * 3 * 4 + 3 * 3 * 4) = n * 72$ B, e.g., for model of hand (419554 triangles), the size of data is 28.8 MB (as the used GPU contains 32 MB memory, rendering an object with higher number of triangles leads to a significant slowdown). The first benefit of triangle strips is that the critical size of model is about twice higher. The second benefit is that it is possible to re-use the last two transformed vertices, thus avoid some transformation and lighting computation.

Using the stripification significantly increases the speed of rendering while using display lists. The speedup is higher than 2.5 and the differences between different stripifi-

cation techniques are smaller than 15%. The highest framerates are reached with models stripified by *Tunneling* and *Multi-Path* (generally, strip minimizing methods seem to produces more suitable stripification for rendering).

algorithm	Tris		SGI		MStrip		STRIPE		FTSG		Tunnel	M-Path	
model	TRI	BOG	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
cow	314.2	359.8	354.2	341.3	350.0	350.8	354.5	350.1	349.6	336.9	357.3	357.1	354.8
demi	288.0	363.1	351.3	324.6	350.6	340.2	350.9	342.7	350.1	324.8	354.7	356.0	353.6
bunny	93.1	213.3	187.9	162.0	184.0	165.2	192.0	174.3	177.9	158.5	202.6	193.9	188.0
dinosaur	64.9	179.0	165.5	122.2	160.6	148.3	164.0	150.6	156.5	135.8	172.1	167.9	161.9
balljoint	25.3	87.4	76.2	53.9	75.6	70.2	77.9	69.9	72.7	56.1	82.3	80.9	78.1
club	20.6	91.3	72.8	35.7	73.7	67.5	77.1	67.8	71.4	40.2	82.5	80.8	74.6
hand	64.4	68.7	56.1	19.4			55.3	50.3	51.6	22.3	63.9	61.1	58.8
dragon	31.3	55.2	48.1	12.2			37.3	36.8	45.0	16.2	51.9	50.3	50.9
happy	27.3	47.8	41.2	11.0			30.1	27.3	33.4	13.1	13.9	43.6	44.2
blade	0.2	0.2	10.5	7.1			11.0	8.0	9.4	6.7	13.6	11.6	10.2

Table 8.5: *The average FPS using vertex buffer objects with indexed primitives.*

The vertex buffer objects (VBO) consist of two (or more) arrays. The first array defines the vertices coordinates (it is also possible to define other vertex properties such as normals, colors or texture coordinates), the second array defines all primitives (triangle, triangle strips) as sets of indices to the vertex array. Such an approach allows to use the integrated vertex cache. The real size of data is approximately: $\frac{n}{2} * 24 + n * 12 = n * 24$ B, where n is the number of triangles and $\frac{n}{2}$ is the number of vertices.

As the Bogomjakov's method produces well ordered triangles (the ACMR is 0.72, see Table 8.6), the rendering speed is very high. The positive effect of the vertex cache is noticeable if we compare this speed to the speed of rendering of unordered triangles. Although the speed higher than the rendering speed of stripified models rendered by VBO, for most of the models, it is lower than rendering speed of stripified models rendered by display lists. But similarly to display lists, there is a big drop down in speed for large models (triangle model of blade with more than 1.7 million of triangles is larger than 40 MB, this size can be reduced to less than 30 MB by triangle strips).

The lower speed of rendering of triangle strip models can be caused by two facts: the *ACMR* is higher than in Bogomjakov's method, thus more vertices have to be transformed multiple times; and there is a function call overhead while using strips, as each strip has to be sent separately to the GPU, while all triangles can be sent in one call.

ACMR

New graphic cards often contain a small vertex cache (in tens of vertices) that can significantly speed up the rendering, as cached vertices do not need to be processed repeatedly. To maximize the benefit of this vertex cache, the strips somehow have to preserve the locality to minimize the average cache miss ratio (*ACMR*). The theoretical upper bound is about 3.0 – each of *n* vertices has to be processed every time, thus in an average mesh $ACMR(k) = \frac{6 \cdot n}{2 \cdot n} = 3$. The theoretical lower bound is 0.5 as each vertex has to be cached at least once ($ACMR(k) = \frac{n}{2 \cdot n} = 0.5$). In Table 8.6, we present the *ACMR* for the vertex cache of size 16, which is used quite often.

algorithm	Tris		SGI		MStrip		STRIPE		FTSG		Tunnel	M-Path	
model	TRI	BOG	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
cow	1.74	0.73	0.83	0.98	0.85	0.95	0.80	0.92	0.92	1.03	0.78	0.81	0.83
demi	2.24	0.73	0.81	0.97	0.86	0.93	0.81	0.91	0.88	1.01	0.80	0.81	0.84
bunny	2.08	0.74	0.90	0.99	0.92	0.99	0.82	0.96	0.97	1.03	0.81	0.87	0.90
dinosaur	2.85	0.73	0.86	0.99	0.88	0.96	0.81	0.93	0.93	1.03	0.80	0.82	0.87
balljoint	2.94	0.73	0.88	0.99	0.89	0.98	0.82	0.94	0.96	1.03	0.81	0.82	0.86
club	2.95	0.73	0.91	0.99	0.92	0.98	0.82	0.95	0.96	1.03	0.82	0.84	0.91
hand	0.99	0.72	0.84	0.99			0.81	0.91	0.91	1.02	0.79	0.82	0.83
dragon	1.66	0.71	0.79	0.97			0.80	0.87	0.87	1.01	0.77	0.78	0.79
happy	1.55	0.71	0.79	0.97			0.80	0.87	0.86	1.01	0.77	0.78	0.78
blade	1.24	0.73	0.87	0.99			0.81	0.92	0.93	1.03	0.79	0.85	0.86

Table 8.6: *The average cache miss rate (ACMR) for cache size k = 16 (grey cells emphasize the best values, black cells emphasize the worst values).*

In general, the vertex minimizing algorithm produce worse rendering sequence for all models. This is caused by the way the algorithms work. As they are minimizing the number of vertices, the strips produced by these algorithms are straight, thus they are not localized. Although the stripification algorithms are not designed for larger vertex caches, the ACMR is quite low (the difference between *Tunneling* and Bogomjakov's method is about 10%).

Execution Time

The time of the stripification process is actually not very crucial, as the stripification is usually made in a preprocessing stage. The execution times presented in Table 8.7 do not include the I/O operation (except the *STRIPE* algorithm, where the output operation runs during the stripification process).

algorithm	Tris	SGI		MStrip		STRIPE		FTSG		Tunnel	M-Path	
model	BOG	SGS	SGV	MSS	MSV	STS*	STV*	FTS	FTV	TUS	MPS	EMS
cow	0.33	0.05	0.02	0.01	0.01	0.20	0.22	0.02	0.02	0.64	0.03	0.03
demi	1.01	0.09	0.02	0.04	0.04	0.64	0.64	0.06	0.06	0.97	0.09	0.16
bunny	4.66	0.38	0.14	0.19	0.19	2.36	2.40	0.27	0.28	101.72	0.40	0.67
dinosaur	7.75	0.64	0.22	0.34	0.34	3.63	3.66	0.45	0.47	45.86	0.69	1.06
balljoint	20.93	1.64	0.64	0.99	1.01	8.89	8.95	1.19	1.19	109.67	1.74	15.42
club	34.45	2.54	0.92	1.50	1.51	13.56	13.79	1.80	1.81	367.53	2.64	32.13
hand	57.18	3.52	1.20			26.37	26.30	2.24	2.21	343.06	3.64	11.51
dragon	84.38	5.02	1.92			27.39	27.16	3.25	3.48	672.55	5.18	134.80
happy	113.21	6.30	2.56			33.90	33.56	4.12	4.42	897.42	6.49	163.09
blade	118.51	9.89	4.08			86.33	86.48	6.24	6.58	3448.61	10.24	72.16

Table 8.7: *The computation time in seconds (grey cells emphasize the best values, black cells emphasize the worst values; * as the* STRIPE *algorithm saves the result during the stripification process, it is impossible to show the time of stripification and we present the total time minus average of I/O times for other algorithms).*

All algorithms except *STRIPE*, *Extended Multi-Path* and *Tunneling* produce the stripification in about the same time. One reason why the *STRIPE* is slow is that the algorithm saves the result during the stripification process and it is impossible exclude the I/O time from the measurement (we present the total running time minus the average time of I/O operations for other algorithms).

As *Tunneling* 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 running time of *Extended Multi-Path* is higher mainly for irregular meshes, as in irregular mesh, it takes longer time to find the node with minimal weight difference.

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

Memory Usage

As different algorithms use different data structures, the amount of allocated memory can differ (Table 8.8). 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.

algorithm	SGI		MStrip		STRIPE		FTSG		Tunneling	M-Path	
model	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
cow	1.1	1.6	4.6	5.1	3.5	4.1	2.3	2.0	5.1	2.0	2.1
demi	2.1	2.4	6.0	6.5	8.3	8.1	4.2	3.3	11.2	3.4	3.6
bunny	6.8	6.0	12.4	12.9	27.0	26.9	13.6	12.9	37.2	9.4	9.9
dinosaur	9.3	11.0	17.6	18.1	42.2	42.5	17.5	17.5	57.8	14.4	15.2
balljoint	21.0	22.2	37.4	37.9	100.5	100.6	38.8	38.8	137.6	33.3	34.9
club	34.9	34.0	55.2	55.2	152.8	152.9	58.0	58.0	209.3	50.0	52.5
hand	53.9	52.0			237.3	237.3	118.2	118.2	325.0	78.0	81.9
dragon	71.5	68.0			315.2	315.2	122.1	122.1	435.1	104.4	109.6
happy	88.9	86.0			391.8	392.8	190.2	190.2	540.8	129.8	136.3
blade	143.3	136.3			430.5	427.2	298.8	298.8	879.0	207.1	211.2

Table 8.8: *The amount of allocated memory in MB (grey cells emphasize the best values, black cells emphasize the worst values).*

The *Tunneling* 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 we do not know the reason. As far as we 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 into a file.

8.3 Regular Data

We have also tested the behavior of these algorithms on an artificial object – torus. This object has several advantages: it has no borders, all vertices has six neighboring triangles and it is very easy to generate a torus with various complexity. For our tests, we have used 20-sided torus with 20 to 10000 segments. We present only the most important tables, i.e., number of strips (Table 8.9) and number of vertices (Table 8.10).

All strip minimizing algorithms (except MStrip, which starts building two strips and does not succeed to connect them) produce a single strip stripification. The number of strips produced by vertex minimizing methods (except STRIPE) depends on the number of sides of the torus and on the divisibility of number of segments and number of sides.

The number of vertices is presented in Table 8.10. Nearly all methods produce a stripification of about the same size as the number of triangles in the stripified model. As the

algorithm	SGI		MStrip		STRIPE		FTSG		Tunneling	M-Path	
model	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
20x20	1	19	2	17	19	1	1	19	1	1	1
20x50	1	19	2	19	40	1	1	9	1	1	1
20x100	1	19	2	19	87	1	1	19	1	1	1
20x200	1	19	2	19	162	1	1	19	1	1	1
20x500	1	19			397	1	1	19	1	1	1
20x1000	1	19			801		1	19	1	1	1
20x2000	1	19			1626		1	19		1	1
20x5000	1	19			4028		1	19		1	1
20x10000	1	19			8036		1	19		1	1

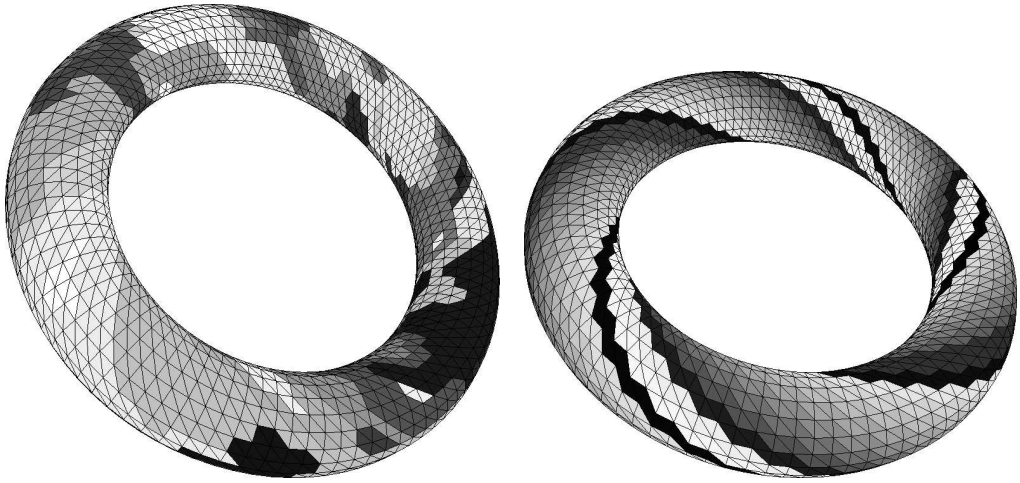
Table 8.9: *Number of strips achieved by the tested algorithms.*

stripification produced by STRIPE (STS) seems to produce random strips, the number of vertices is higher (Figure 8.3 (a)). The Multi-Path algorithm produces a stripification that contains a high number of swaps, thus the number of vertices is very high (Figure 8.3 (b), the brightness indicates the order of triangle in the strip; black is the first triangle in the strip, white is the last).

algorithm	SGI		MStrip		STRIPE		FTSG		Tunneling	M-Path	
model	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
20x20	1K	1K	1K	1K	1K	1K	1K	1K	1K	1K	1K
20x50	2K	2K	2K	2K	3K	2K	2K	2K	2K	3K	2K
20x100	4K	4K	4K	4K	6K	4K	4K	4K	4K	6K	4K
20x200	8K	8K	8K	8K	12K	8K	8K	8K	8K	12K	8K
20x500	20K	20K			29K	20K	20K	20K	20K	30K	20K
20x1000	40K	40K			59K		40K	40K	40K	61K	40K
20x2000	80K	80K			117K		80K	80K		122K	80K
20x5000	200K	200K			292K		200K	200K		305K	200K
20x10000	400K	400K			584K		400K	400K		610K	400K

Table 8.10: *Number of vertices achieved by the tested algorithms.*

We do not present running times, as the time for stripification does not vary too much from the time of stripification of irregular meshes. For the same reason we do not show the memory usage that mainly depends on the number of triangles of the input mesh. As nearly all methods produce the same number of vertices and very long straight strips, i.e., the vertex cache is used only for common edges, in most cases the ACMR is close to 1. The only significant difference in ACMR is produced by STRIPE (STS; Figure 8.3 (a)), which produces a "random" stripification and the ACMR is comparable to ACMR on irregular meshes ($\sim$0.80).



(a) STS (b) MPS

Figure 8.3: *Examples of stripification of regular torus. The stripification produced by STS algorithm is quite irregular (a). The Multi-Path algorithm produces a stripification with a high number of swaps – the brightness indicates the order of triangle in the strip (b).*

8.4 Topology

We were surprised by the similar behavior of all algorithms for various models (Figures 8.1 and 8.2) and on regular data, and we have performed a series of tests, whether it is possible to improve or worsen the quality of stripification by changes in topology (i.e., changes of degrees of vertices). We have chosen two extreme models (bunny and dragon) and we have computed the histogram of vertex degrees (Figure 8.4).

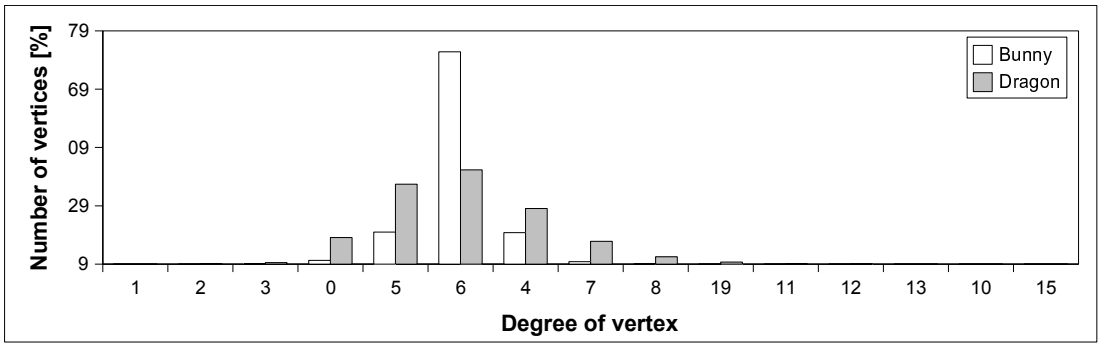


Figure 8.4: *Histogram of degree of vertices. The bunny model is quite regular as more than 70% of vertices have degree six. With less than 33% of vertices of degree six and with 2 vertices of degree 15, the dragon model is highly irregular.*

First, we have tested, what will happen with the quality of stripification if we make the bunny model irregular. To do this, we have randomly swapped 10000, 20000 and 30000 edges. The random swapping has changed the histogram of degrees of vertices as shown in Figure 8.5. We have used these inputs a we have performed the stripification for each tested method.

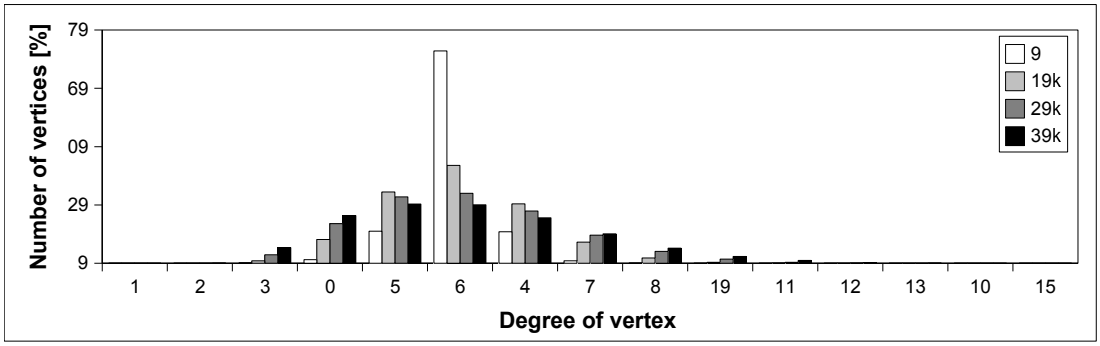


Figure 8.5: *Histogram of degree of vertices. The edges in the bunny model were randomly swapped, which produces more irregular model.*

Increasing the irregularity, the number of vertices (Table 8.11) and the number of strips (Table 8.12) increases as we have expected. For nearly all algorithms, the number of vertices used in stripification increased about 10-15%, as in irregular mesh, the possibility of sequential strip decreases, thus the number of swaps is higher. As *Tunneling* and *Multi-Path* produces triangle strips with high number of swaps even for regular meshes, the increase of vertices is not so high.

algorithm	SGI		MStrip		STRIPE		FTSG		Tunneling	M-Path	
model	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
bunny	87K	82K	86K	82K	102K	83K	85K	82K	99K	94K	89K
10k	97K	90K	96K	91K	102K	91K	94K	92K	100K	100K	98K
20k	99K	90K	98K	91K	102K	92K	96K	94K	100K	100K	100K
30k	100K	90K	98K	91K	103K	92K	96K	94K	100K	100K	101K

Table 8.11: *Number of vertices achieved by the tested algorithms for original bunny, 10k, 20k and 30k random swaps.*

While increasing the number of vertices in strips, the ACMR decreases as the strips are becoming more local (Table 8.14). Although the low ACMR should improve the rendering speed, the increase of number of strips and number of vertices is too high and the rendering does not vary too much (Table 8.13).

algorithm	SGI		MStrip		STRIPE		FTSG		Tunneling	M-Path	
model	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
bunny	648	3560	573	1173	1531	1229	618	3238	166	154	88
10k	1143	5626	1117	1749	1655	1816	1417	4555	225	325	242
20k	1520	6206	1414	1879	1861	2014	1815	5044	290	490	430
30k	1644	6537	1590	1975	2048	2101	2059	5140	346	639	605

Table 8.12: *Number of strips.*

algorithm	SGI		MStrip		STRIPE		FTSG		Tunneling	M-Path	
model	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
bunny	232.1	218.1	226.2	220.8	236.7	221.5	223.2	210.4	240.7	235.2	229.2
10k	236.1	216.9	228.9	220.6	234.8	229.1	231.9	210.1	239.3	237.9	237.6
20k	236.1	218.1	228.5	221.2	234.1	230.9	232.4	210.5	237.4	234.6	235.9
30k	234.6	218.1	228.2	221.9	233.1	230.9	232.5	210.0	236.5	232.3	232.5

Table 8.13: *The rendering speed in frames per second (FPS).*

algorithm	SGI		MStrip		STRIPE		FTSG		Tunneling	M-Path	
model	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
bunny	0.99	0.99	0.92	0.99	0.82	0.96	0.97	1.03	0.81	0.87	0.90
10k	0.98	0.98	0.83	0.89	0.81	0.87	0.87	1.02	0.78	0.79	0.80
20k	0.96	0.96	0.81	0.85	0.79	0.83	0.84	1.00	0.76	0.78	0.78
30k	0.95	0.95	0.80	0.84	0.78	0.80	0.82	0.99	0.75	0.77	0.78

Table 8.14: *The average cache miss rate (ACMR) for cache size* k = 16.

The processing time of most of the algorithms does not depend on the regularity of the mesh (Table 8.15). As the Extended Multi-Path algorithm is designed for regular meshes, the processing time is higher for irregular meshes. In regular meshes, there usually exist several candidates with zero weight difference, thus the searching for the next triangle is done very quickly. In irregular meshes, it is usually necessary to search in larger list of possible candidates for the minimal weight different, which increases the processing time.

algorithm	SGI		MStrip		STRIPE		FTSG		Tunneling	M-Path	
model	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
bunny	0.386	0.125	0.189	0.191	2.996	3.043	0.265	0.266	101.282	0.396	0.675
10k	0.342	0.141	0.193	0.193	2.986	2.961	0.265	0.288	16.048	0.351	0.957
20k	0.354	0.140	0.193	0.194	2.970	2.959	0.266	0.288	10.266	0.360	0.951
30k	0.353	0.141	0.193	0.197	2.974	2.955	0.266	0.288	10.203	0.365	1.953

Table 8.15: *The processing time in seconds.*

Surprisingly, the *Tunneling* algorithm is much faster for irregular meshes (for a bunny model with 30 thousand of edge swaps, the time decreases by a factor of 10). This is probably caused by the minimal length of tunnels that can be found in irregular meshes. In regular mesh, the strips are usually long and straight, thus the tunnels (i.e., a path between ends of two different strips) are long. As the tunnels are searched by a breadth first search algorithm, the running time is drastically increasing.

We have also tested, whether it is possible to improve the quality of stripification by regularizing the input mesh. We have chosen an irregular mesh (dragon) and we iteratively improved the mesh by swapping the edges that are incident to high degree vertices. The changes in the histogram of vertex degrees are presented in Figure 8.6. Similarly to previous test, the number of vertices (Table 8.16) and number of strips (Table 8.17) decrease with increase of regularity.

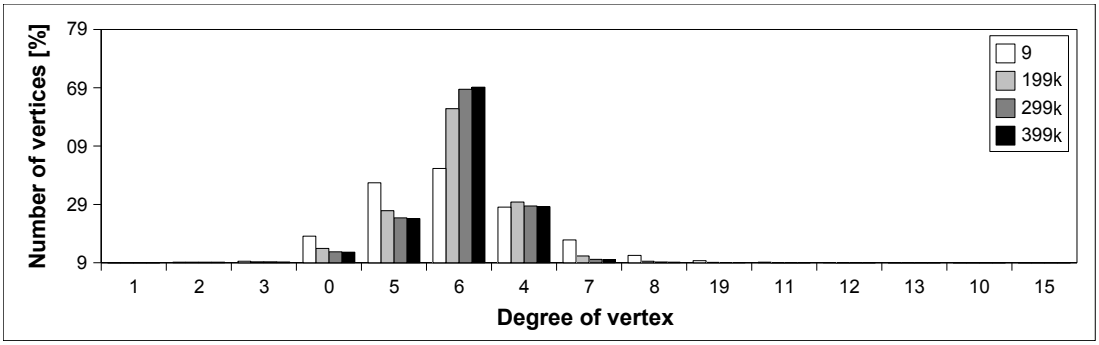


Figure 8.6: *Histogram of degree of vertices. The edges incident to high degree vertices in the dragon model were swapped, to produce more regular model.*

algorithm	SGI		MStrip		STRIPE		FTSG		Tunneling	M-Path	
model	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
dragon	1237K	1130K			1290K	1153K	1196K	1156K	1261K	1254K	1243K
100k	1195K	1098K			1286K	1119K	1167K	1108K	1257K	1242K	1227K
200k	1179K	1087K			1287K	1105K	1156K	1092K	1252K	1237K	1220K
300k	1177K	1085K			1286K	1103K	1155K	1091K	1254K	1237K	1220K

Table 8.16: *Number of vertices achieved by the tested algorithms for original dragon, 100k, 200k and 300k swaps of edges that are incident to high degree vertices.*

algorithm	SGI		MStrip		STRIPE		FTSG		Tunneling	M-Path	
model	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
dragon	17399	71182			22928	25355	20571	58377	3331	4832	4287
100k	14105	66036			21292	21899	17249	56968	2761	3615	3229
200k	13069	63391			20920	20660	15928	55965	2516	3430	2852
300k	12957	63121			20891	20914	15917	55579	2591	3210	2794

Table 8.17: *Number of vertices in strips.*

8.5 Summary

As the *SGI* vertex minimizing algorithm produces the lowest number of vertices, it seems to be the best choice for storing data in strip representation. Due to high number of strips produced by this algorithm, the rendering speed is low. The speed can be improved by using other vertex minimizing algorithms (specially *MStrip* and *STRIPE*) which produce nearly the same number of vertices as the *SGI*.

The strip minimizing algorithms produce stripifications that are usually rendered faster than other stripifications. This is probably caused by the smaller number of system calls when starting a new strip and by lower cache-miss ratio. The *Tunneling* algorithm produces the best stripification in most of the cases, but the stripification process is memory and time-consuming. Similar stripifications can be obtained by *Multi-Path* algorithm, which is much faster.

We show that although triangle stripping algorithms are not designed for vertex caches, they produce a good rendering sequences and they still have a place in real-time applications.

We also show that the quality of stripification depends on the topology of input mesh – for regular meshes, the stripification contains a lower number of strips and vertices, but the ACMR is higher and vice versa.

Conclusions and Future Work

As we have presented several different stripification algorithms, in this chapter, we summarize the main results of our work. We also address several issues for a future work.

9.1 Main Results

In this work, we have suggested several stripification methods for various kind of problems:

1. For a set of points, we have designed a method that creates a Delaunay stripification in $O(n \log n)$. This method is based on an incremental insertion method for Delaunay triangulation. Although our approach produces a high number of vertices and strips, still there is a significant rendering speedup. As the method is based on incremental insertion, it can be used for visualization of level of details or for fast on-line visualization (Chapter 3).

2. For fully triangulated models, we present a fast stripification method that produces the lowest number of strips in $O(n)$ time. The number of strips produced by our method is more than three times lower than the number of strips produced by other linear-time stripification methods (Chapter 4). To decrease the number of vertices in the stripification and/or to give the user a better control of stripification process, we have also designed an extension of our algorithm (Chapter 5).

3. As the problematic of fast visualization and computation of volume tetrahedral data is becoming very important in many applications, we have suggested a modification of our stripification algorithm for tetrahedral meshes. Similarly to triangle meshes stripification, our algorithm produces much lower number for tetrahedral strips than already existing algorithms. Unfortunately, as there is no specification and no hardware support of tetrahedral meshes, we are not able to make a solid comparison of tetrahedral stripification methods (Chapter 6).

4. As the importance of quadrilateral meshes in real-time computer graphic increases, we have designed a triangle stripping method for this type of meshes. According to all our tests, our new method produces better stripification than any other stripification method (Chapter 7).

We also present an overall comparison of important stripification methods. Our tests were performed on a set of very often used triangular meshes and on artificial datasets. We have also studied the impact of mesh topology on the quality of stripification. We show that although the stripification methods and triangle strips in general are not designed for the new GPUs with large vertex caches, they are still important in real-time visualization (Chapter 8).

9.2 Future Work

The topic of stripification is quite well explored and we think, there is no need to design some new general stripification algorithms. Still there is a place for researching stripification methods for some specialized problems:

- Although there exist some stripification methods for visualization of CLOD meshes ([Ste01, RCBR04, PSS05]) or view-dependent meshes ([ESAV99, SP03, RCBR04, DBPM05]), there is probably still a place to optimize these methods or to find some new approaches.
- There exist a lot of topology compression algorithms for triangle meshes. Usually, these methods are not based on stripification, thus it is necessary to use some reordering or stripification algorithms to achieve a better rendering speed after decompression. On the other hand, lot of these methods uses some kind of path searching or spanning tree algorithm to get the sequences of triangles and there is a possibility to modify these algorithms to produce sequences that are more strip friendly in decompression stage.
- There is still an open problem in the field of tetrahedral meshes. We think that this problem cannot be solved unless there will be some hardware support for tetrahedral rendering and tetrahedral strips.

References

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

[AHB90] K. Akeley, P. Haeberli, and D. Burns. tomesh.c. C Program on SGI Develope's Toolbox CD, 1990.

[AHMS96] E.M. Arkin, M. Held, J.S.B. Mitchell, and S.S. Skiena. Hamiltonian Triangulations for Fast Rendering. *The Visual Computer*, 12(9):429–444, 1996.

[BA02] J. Behr and M. Alexa. Fast and Effective Striping. In *OpenSG'02*, 2002.

[BD00] C. Beeson and J. Demer. NvTriStrip Library, 2000. Software available via NVidia Internet web site http:// developer.nvidia.com/ object/ nvtristrip_library.html.

[BG01] A. Bogomjakov and C. Gotsman. Universal Rendering Sequences for Transparent Vertex Caching of Progressive Meshes. In B. Watson and J. W. Buchanan, editors, *Proceedings of Graphics Interface 2001*, pages 81–90, 2001.

[Bro79] K.Q. Brown. Voronoi Diagrams from Convex Hulls. In *Information Processing Letters*, pages 223–228, 1979.

[BRRC01] 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.

[BYG96] R. Bar-Yehuda and C. Gotsman. Time/Space Tradeoffs for Polygon Mesh Rendering. *ACM Transactions on Graphics*, 15(2):141–152, 1996.

[CC99] S.W. Cheng and J.S. Cheong. A Triangulation for Optimal Strip Decomposition in Simple Polygons. In *The Second ACM Hong Kong Postgraduate Research Day*, 1999.

[Chr75] N. Christophides. *Graph Theory, an Algorithmic Approach*. Academic Press, New York, 1975.

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

[DBPM05] P. Diaz, A. Bhushan, R. Pajarola, and Gopi M. Constrained Strip Generation and Management for Efficient Interactive 3D Rendering. In *Computer Graphics International 2005*, 2005.

[Dee95] M. Deering. Geometry Compression. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, pages 13–20. ACM Press, 1995.

[Dwy86] 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*, pages 276–284. ACM Press, 1986.

[EG04] D. Eppstein and M. Gopi. Single-strip Triangulation of Manifolds with Arbitrary Topology. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 455–456, 2004.

[EMX02] R. Estkowski, J.S.B. Mitchell, and X. Xiang. Optimal Decomposition of Polygonal Models into Triangle Strips. In *Proceedings of the eighteenth annual symposium on Computational geometry*, pages 254–263, 2002.

[ESAV99] 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*, pages 131–138. IEEE Computer Society Press, 1999.

[ESV96a] 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.

[ESV96b] F. Evans, S. Skiena, and A. Varshney. Optimizing Triangle Strips for Fast Rendering. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pages 319–326, 1996.

[Eva98] F. Evans. Stripe, 1998. http://www.cs.sunysb.edu/ ˜stripe/.

[Geo] Georgia Institute of Technology. Large geometric models archive. http:// www.cc.gatech.edu/ projects/ large_models/.

[Hop99] H. Hoppe. Optimization of Mesh Locality for Transparent Vertex Caching. In Alyn Rockwood, editor, *Siggraph 1999, Computer Graphics Proceedings*, pages 269–276, Los Angeles, 1999. Addison Wesley Longman.

[KCW01] D. King1 and H.J. Wolters C.M. Wittenbrink. An Architecture for Interactive Tetrahedral Volume Rendering. Technical report, HP Laboratories Palo Alto, 2001.

[Koc92] W. Kocay. An Extension of the Multi-Path Algorithm for Finding Hamilton cycles. *Discrete Mathematics 101*, pages 171–188, 1992.

[Kor99] D. Kornmann. Fast and Simple Triangle Strip Generation. Technical report, VMS Finland, Espoo, Finland, 1999.

[KŽ02] I. Kolingerová and B. Žalik. Improvements to Randomized Incremental Delaunay Insertion. *Computers & Graphics*, 26:477–490, 2002.

[LGS90] 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*, pages 414–431. Springer, New York, 1990.

[NVI03] NVIDIA Corporation. Using Vertex Buffer Objects. White Paper: http:// developer.nvidia.com/ object/ using_VBOs.html, 2003.

[PS03] M.B. Porcu and R. Scateni. An Iterative Stripification Algorithm Based on Dual Graph Operations. In *Eurographics '03 (Short)*, 2003.

[PS04] M.B. Porcu and R. Scateni. An Iterative Stripification of Triangle Mesh: Focus on Data Structures. In *Proceedings of the 12th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG'2004) (Poster)*, 2004.

[PSS05] M. B. Porcu, N. Sanna, and R. Scateni. Efficiently Keeping an Optimal Stripification over a CLOD Mesh. In *Proceedings of the 13th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG'2005)*, 2005.

[PZ04] D. Pospíšil and F. Zbořil. Building Triangle Strips Using Hopfield Neural Network. In *Electronic Computers and Informatics 2004*, pages 445–451. Technical University of Košice, 9 2004.

[RCBR04] J.F. Ramos, M. Chover, O. Belmonte, and C. Rebollo. An Approach to Improve Strip-Based Multiresolution Schemes. In *International Conferences in Central Europe WSCG 04*, 2004.

[SD95] P. Su and R.L.(Scot) Drysdale. A Comparison of Sequential Delaunay Triangulation Algorithms. In *Symposium on Computational Geometry*, pages 61–70, 1995.

[Ším04] J. Šíma. Tristrips on Hopfield Networks. Technical report, 2004.

[SKP] 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.

[SKP02] M.V.G.da Silva, O.M.van Kaick, and H. Pedrini. Fast Mesh Rendering through Efficient Triangle Strip Generation. In *WSCG'2002*, pages 127–134, 2002.

[SP03] M. Shafae and R. Pajarola. DStrips: Dynamic Triangle Strips for Real-Time Mesh Simplification and Rendering. In *Proceedings Pacific Graphics 2003*, pages 271–280, 2003.

[SS97] B. Speckmann and J. Snoeyink. Easy Triangle Strips for TIN Terrain Models. In *Canadian Conference on Computational Geometry*, pages 239–244, 1997.

[SS04] R. Sviták and V. Skala. Robust Surface Reconstruction from Orthogonal Slices. In *Electronic Computers and Informatics (ECI'04)*, pages 451–456, 2004.

[Sta] Stanford Computer Graphics Laboratory. The stanford 3d scanning repository. http:// graphics.stanford.edu/ data/ 3Dscanrep/.

[Ste01] J. Stewart. Tunneling for Triangle Strips in Continuous Level-of-Detail Meshes. In *Graphics Interface*, pages 91–100, 2001.

[Tau02] G. Taubin. Constructing Hamiltonian Triangle Strips on Quadrilateral Meshes. In *International Workshop on Visualization and Mathematics 2002*, 2002.

[Van02] P. Vaněček. Comparison of Stripification Techniques. In *6-th Central European Seminar on Computer Graphics CESCG'02*, pages 65–74, 2002.

[Van04] P. Vaněček. Triangle Strips For Fast Rendering. Technical Report DCSE/TR-2004-05, 2004.

[VFG99] L. Velho, L.H.de Figueiredo, and J. Gomes. Hierarchical Generalized Triangle Strips. *The Visual Computer*, 15(1):21–35, 1999.

[VK03] P. Vaněček and I. Kolingerová. Fast Delaunay Stripification. In *Proceedings of the 19th Spring Conference on Computer graphics*, 2003.

[VK04a] P. Vaněček and I. Kolingerová. Multi-Path Algorithm for Triangle Strips. In *Computer Graphics International 2004*, pages 2–9, 2004.

[VK04b] P. Vaněček and I. Kolingerová. Weighted Multi-Path Algorithm for Triangle Strips. In *Electronic Computers and Informatics 2004*, pages 445–451. Technical University of Košice, 9 2004.

[VSKS05] P. Vaněček, R. Sviták, I. Kolingerová, and V. Skala. Quadrilateral Meshes Stripification. In *Algoritmy 2005*, pages 300–308. Slovak University of Technology, 2005.

[WMKE04] M. Weiler, P. N. Mallón, M. Kraus, and T. Ertl. Texture-Encoded Tetrahedral Strips. In *Proceedings Symposium on Volume Visualization 2004*, pages 71–78. IEEE, 2004.

[XHM99] 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.

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

[ZS00] D. Zorin and P. Schröder. Subdivision for Modeling and Animation. Technical report, SIGGRAPH 2000, 2000. course notes.

Appendix

A Activities

Publications

- Vaněček P. and Kolingerová I. Comaprison of Triangle Strips Algorithms, In journal of *Computers and Graphics* (preliminarily accepted).
- Vaněček P. and Kolingerová Quadrilateral Meshes Stripfication, *Algoritmy 2005*, pages 300–308, Slovakia, 2005.
- Vaněček P. and Kolingerová I. Weighted Multi-Path Algorithm for Triangle Strips, *Electronic Computers and Informatics 2004*, pages 445–451, Herlany, Slovakia, 2004.
- Vaněček P. and Kolingerová I. Multi-Path Algorithm for Triangle Strips, In *Computer Graphics International (CGI) 2004*, pages 2–9, Crete, Greece, 2004.
- Vaněček P. Triangle Strips For Fast Rendering. (Techreport), 2004.
- Vaněček P. and Kolingerová I. Fast Delaunay Stripification, In *Spring Conference on Computer Graphics (SCCG) 2003*, pages 83–88, Budmerice, Slovakia, 2003 (also published in ACM).
- Vaněček P. Triangle Strips Generation. MSc Thesis, 2002.
- Vaněček P. Comparison of Stripification Techniques. In *6-th Central European Seminar on Computer Graphics (CESCG) 2002*, pages 65–74, Budmerice, Slovakia, 2002.

References

- P. Diaz-Gutierrez, A. Bhushan, M. Gopi, R. Pajarola. Constrained Strip Generation and Management Efficient Interactive 3D Rendering. In *Computer Graphics International (CGI) 2005*.

Related Talks

- Vaněček P. Trojúhelníkové stripy - včera dnes a zítra, Center of Computer Graphics and Data Visualization, University of West Bohemia, Pilsen, Czech Republic, June 2005.
- Vaněček P. Triangle Strips For Fast Rendering, University of Maribor, Slovenia, May 2005.
- Vaněček P. Multi-Path Algorithm for Triangle Strips, Technical University of Graz, Austria, September 2004.
- Vaněček P. Teorie grafu 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

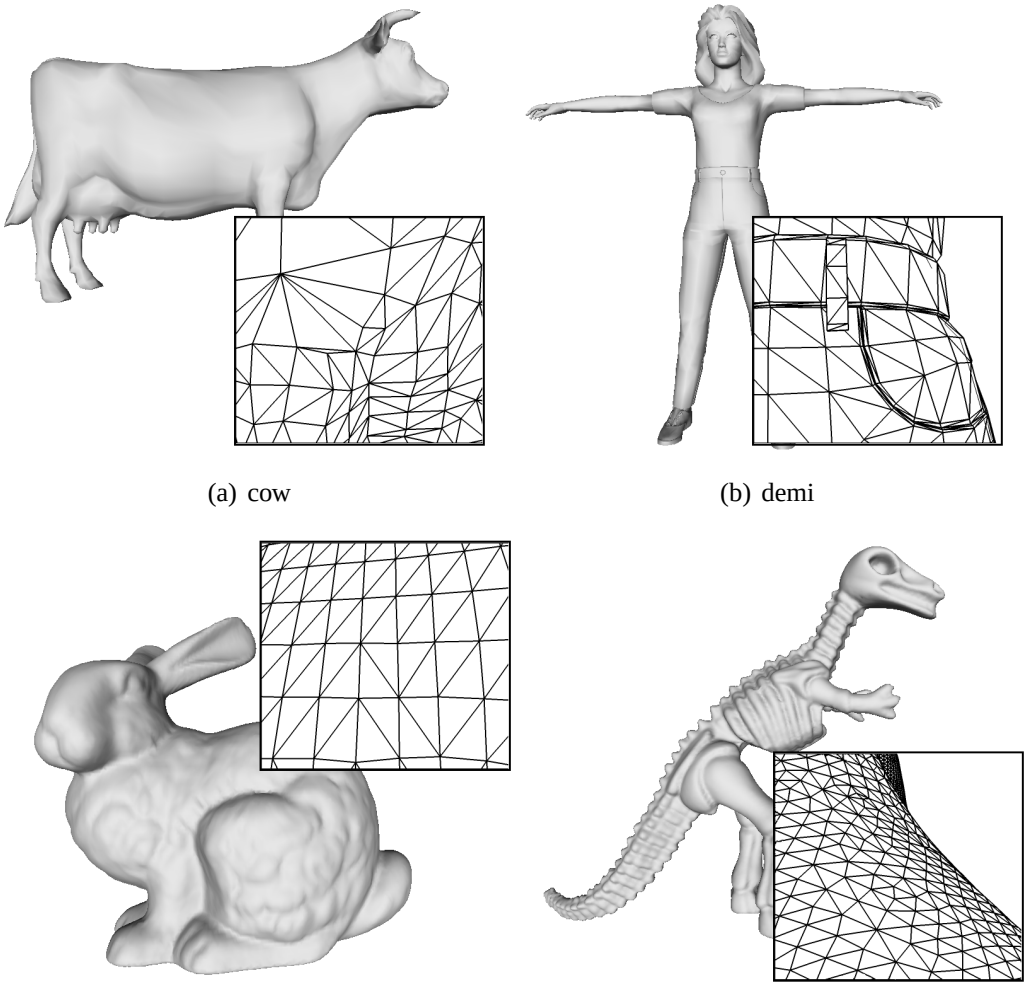
- University of Maribor, Slovenia, May 2005.
- Technical University of Graz, Austria, September 2004.
- Technical University of Graz, Austria, October 2003.
- University of Maribor, Slovenia, September 2003.
- University of Ioannina, Greece, February – August 2001.

Grants

- Projekt CESNET Sdílené virtuální světy, 2005.
- Projekt FRVŠ/G1 - Využití stripové reprezentace pro morphing (spoluřešitel), 2005.
- Projekt FRVŠ/G1 - Rekonstrukce dat z paralelních řezu a jejich vizualizace (hlavní řešitel), 2004.
- Projekt Aktion (s TU Graz), výzkumný pracovník, 2003 – 2004.
- Projekt Kontakt (s University of Maribor), výzkumný pracovník, 2002 – 2005.
- Projekt MSM 235200005 - Informační technologie "Záměry", 2001 – 2004.

B Models

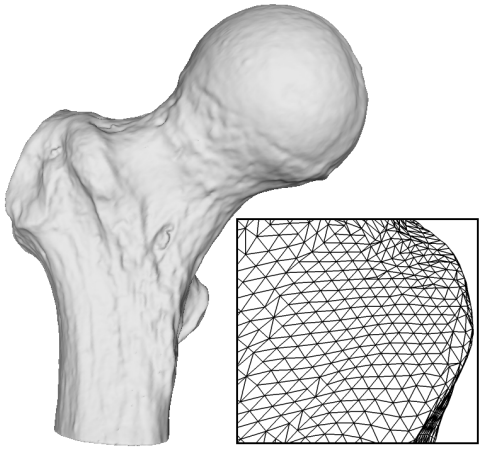
In this section, we present the figures of models used in most of the tests we have performed and the examples of the topology structure of these models.



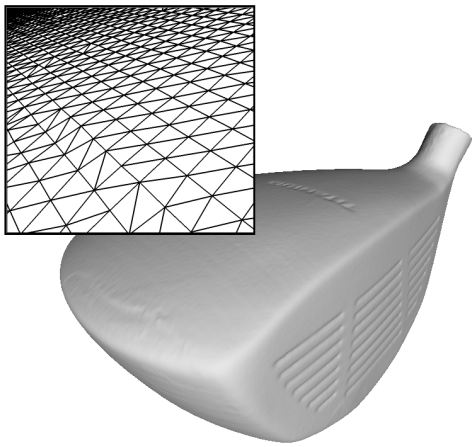
(a) cow (b) demi

(c) bunny (d) dinosaur

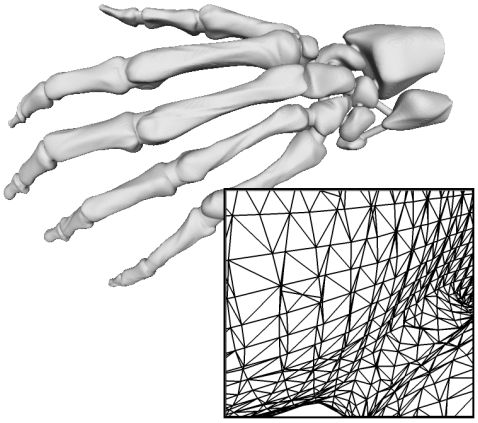
Figure B.1: *Tested models and a part of their triangulation*



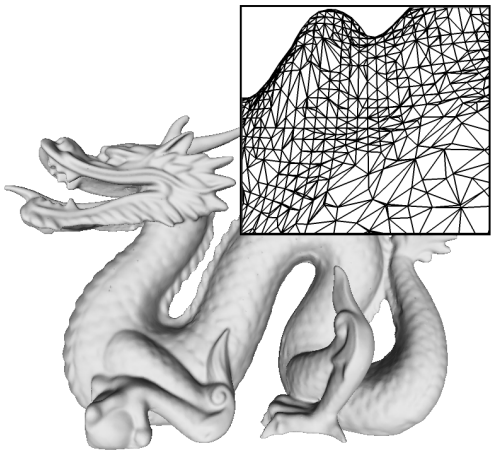
(a) balljoint



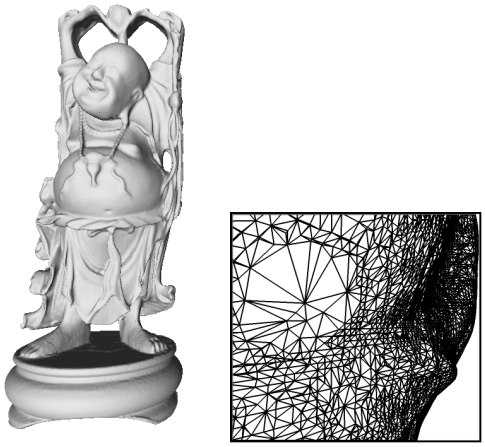
(b) club



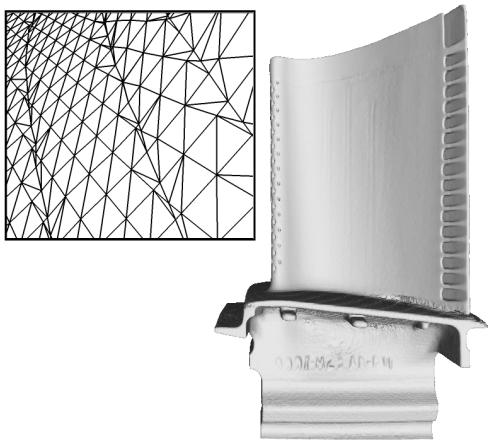
(c) hand



(d) dragon



(e) happy



(f) blade

Figure B.2: *Tested models and a part of their triangulation*

C Results

We present the tests of rendering speed and computation time on various computers.

INTEL Pentium 4 2.8GHz, 1 GB RAM, ATI FireGL T32 32MB

algorithm	Tris		SGI		MStrip		STRIPE		FTSG		Tunnel	M-Path	
model	TRI	BOG	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
The average FPS using display lists													
cow	319.2	361.5	358.0	350.8	353.8	355.8	358.6	355.9	354.5	344.7	359.5	360.3	358.0
demi	299.8	361.2	358.9	351.1	357.9	353.6	358.8	353.7	357.3	347.9	360.4	360.3	358.8
bunny	122.8	162.3	231.7	217.7	226.3	221.4	236.6	221.5	222.8	210.1	240.5	234.8	229.4
dinosaur	116.9	117.3	228.1	216.2	227.2	224.4	227.7	224.9	226.0	216.5	231.4	231.1	229.8
balljoint	56.7	57.2	134.5	125.1	134.1	126.2	134.1	131.6	133.0	124.9	136.9	136.9	135.9
club	39.1	39.2	98.4	91.6	98.1	95.8	97.9	96.4	97.4	91.5	99.8	100.0	99.0
hand	26.0	26.0	69.0	62.7			68.9	67.5	67.8	63.0	71.1	70.9	70.7
dragon	9.8	9.4	55.0	48.1			54.4	53.0	53.5	48.8	56.7	56.6	56.4
happy	7.8	7.5	45.4	39.5			44.8	43.6	44.0	40.1	46.6	46.7	46.5
blade	4.8	4.6	28.4	25.8			28.3	27.2	27.9	25.8	29.3	29.1	29.0
The average FPS using vertex buffer objects													
cow	314.2	359.8	354.2	341.3	350.0	350.8	354.5	350.1	349.6	336.9	357.3	357.1	354.8
demi	288.0	363.1	351.3	324.6	350.6	340.2	350.9	342.7	350.1	324.8	354.7	356.0	353.6
bunny	93.1	213.3	187.9	162.0	184.0	165.2	192.0	174.3	177.9	158.5	202.6	193.9	188.0
dinosaur	64.9	179.0	165.5	122.2	160.6	148.3	164.0	150.6	156.5	135.8	172.1	167.9	161.9
balljoint	25.3	87.4	76.2	53.9	75.6	70.2	77.9	69.9	72.7	56.1	82.3	80.9	78.1
club	20.6	91.3	72.8	35.7	73.7	67.5	77.1	67.8	71.4	40.2	82.5	80.8	74.6
hand	64.4	68.7	56.1	19.4			55.3	50.3	51.6	22.3	63.9	61.1	58.8
dragon	31.3	55.2	48.1	12.2			37.3	36.8	45.0	16.2	51.9	50.3	50.9
happy	27.3	47.8	41.2	11.0			30.1	27.3	33.4	13.1	13.9	43.6	44.2
blade	0.2	0.2	10.5	7.1			11.0	8.0	9.4	6.7	13.6	11.6	10.2
The computation time in seconds													
cow		0.3	0.1	0.0	0.0	0.0	0.2	0.2	0.0	0.0	0.6	0.0	0.0
demi		1.0	0.1	0.0	0.0	0.0	0.6	0.6	0.0	0.1	1.0	0.1	0.2
bunny		4.7	0.4	0.1	0.2	0.2	2.4	2.4	0.3	0.3	101.7	0.4	0.7
dinosaur		7.8	0.6	0.2	0.3	0.3	3.6	3.7	0.5	0.5	45.9	0.7	1.1
balljoint		20.9	1.6	0.6	1.0	1.0	8.9	9.0	1.2	1.2	109.7	1.7	15.4
club		34.5	2.5	0.9	1.5	1.5	13.6	13.8	1.8	1.8	367.5	2.6	32.1
hand		57.2	3.5	1.2			26.4	26.3	2.2	2.2	343.0	3.6	11.5
dragon		84.4	5.0	1.9			27.4	27.2	3.3	3.5	672.6	5.2	134.8
happy		113.2	6.3	2.6			33.9	33.6	4.1	4.4	897.4	6.5	163.1
blade		118.5	9.9	4.1			86.3	86.5	6.2	6.6	3448.6	10.2	72.2

AMD Athlon XP-M 2.1 GHz, 512 MB RAM, ATI Radeon Mobility 9600, 64 MB

algorithm	Tris		SGI		MStrip		STRIPE		FTSG		Tunnel	M-Path	
model	TRI	BOG	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
The average FPS using display lists													
cow	545.7	658.7	646.6	623.2	630.5	626.8	650.2	610.4	634.9	609.6	617.9	652.5	647.3
demi	458.4	614.9	616.5	592.3	616.5	603.3	613.5	595.6	611.0	588.7	612.7	621.7	620.7
bunny	157.8	180.4	340.3	315.9	340.3	309.2	341.8	321.4	331.4	306.9	350.9	351.8	343.5
dinosaur	121.5	121.6	274.1	248.1	273.2	266.2	270.3	261.7	270.6	246.5	283.9	282.3	281.4
balljoint	54.4	54.4	137.5	121.4	136.6	132.4	134.1	132.6	135.4	120.9	141.3	141.3	140.8
club	36.2	36.1	95.1	84.6	94.6	91.6	92.2	91.1	94.0	84.4	96.3	97.1	96.3
hand	23.7	23.6	62.6	54.8			61.5	60.9	61.2	52.4	65.0	65.4	65.6
dragon	4.6	4.5	48.1	38.5			46.4	44.5	46.5	35.8	47.2	50.8	50.8
happy	3.7	3.6	38.1	30.9			35.0	36.6	37.5	28.6	38.2	41.2	41.0
blade	1.6	1.6	23.5	20.6	23.1	22.8			22.7	19.3	24.0	24.6	24.8
The average FPS using vertex buffer objects													
cow	530.1	654.2	633.7	594.2	621.1	607.3	633.0	616.5	618.1	583.3	635.6	642.9	637.9
demi	442.1	640.7	603.2	525.3	601.3	571.1	597.9	576.6	598.9	529.5	613.5	616.1	611.8
bunny	108.8	291.9	247.5	205.8	242.4	214.2	256.1	226.7	233.8	202.2	273.9	258.5	249.8
dinosaur	69.6	218.3	202.4	150.1	195.4	177.2	198.9	181.0	191.2	151.5	211.4	205.9	199.6
balljoint	27.8	98.8	85.7	64.4	84.6	74.6	86.7	77.4	81.1	65.4	92.9	91.2	88.2
club	21.6	95.4	76.4	56.7	76.7	66.9	78.1	69.1	74.1	55.8	86.5	84.2	77.8
hand	43.6	48.4	37.7	25.3			37.2	32.9	34.8	26.2	43.7	42.0	40.5
dragon	26.3	47.4	39.2	19.4			38.0	35.2	36.0	23.6	44.1	42.9	43.2
happy	24.9	45.7	37.6	15.6			36.1	33.8	35.5	19.0	43.1	41.8	42.2
blade	19.2	13.2	9.6	6.1	9.2	7.6			8.5	5.5	12.4	11.3	9.5
The computation time in seconds													
cow		0.4	0.1	0.0	0.0	0.0	0.2	0.3	0.0	0.0	0.4	0.1	0.1
demi		1.2	0.2	0.0	0.0	0.0	0.6	0.6	0.1	0.1	0.8	0.3	0.2
bunny		5.5	0.7	0.2	0.3	0.3	2.4	2.5	0.3	0.3	157.3	0.6	0.8
dinosaur		9.0	1.0	0.4	0.5	0.5	3.7	3.8	0.5	0.5	63.3	1.0	1.3
balljoint		24.5	2.5	1.1	1.4	1.4	9.3	9.4	1.5	1.3	153.2	2.8	21.0
club		39.9	4.0	1.4	2.2	2.2	14.9	15.1	2.2	2.0	557.8	4.3	46.7
hand		65.1	4.0	1.8			28.8	28.7	2.4	2.5	523.0	4.2	12.6
dragon		96.3	8.0	2.8			29.0	29.7	3.7	3.8	1069.7	8.1	205.2
happy		129.0	10.0	3.7			36.4	36.6	4.7	4.8	1481.7	6.9	247.7
blade		271.6	10.4	6.0	7.6	7.6			7.1	7.2	2514.9	12.7	120.2

AMD Athlon 64 2800+ 1.8 GHz, 1 GB RAM, ATI Radeon 9600 XT, 128 MB

algorithm	Tris		SGI		MStrip		STRIPE		FTSG		Tunnel	M-Path	
model	TRI	BOG	SGS	SGV	MSS	MSV	STS	STV	FTS	FTV	TUS	MPS	EMS
The average FPS using display lists													
cow	476.8	565.4	556.9	539.5	544.3	543.8	556.0	551.4	545.6	530.0	560.4	560.6	555.6
demi	429.0	555.8	549.4	532.7	548.5	538.0	548.4	539.2	547.2	526.7	554.9	554.4	551.7
bunny	155.6	196.5	308.1	289.4	299.7	276.4	313.6	296.1	298.7	275.7	323.9	315.7	306.5
dinosaur	135.8	135.9	296.3	267.9	294.0	288.9	294.2	288.6	292.9	273.0	301.6	301.3	299.6
balljoint	62.3	62.4	157.7	143.8	157.6	153.0	156.3	153.2	156.0	142.8	161.8	161.8	160.2
club	41.8	41.8	111.3	101.1	111.2	108.0	110.4	108.1	109.8	101.1	113.6	113.6	112.4
hand	27.6	27.5	75.9	67.7			75.3	73.7	74.1	67.4	78.7	78.3	78.3
dragon	20.9	20.9	58.0	50.6			58.7	56.9	57.5	49.9	61.8	61.6	61.5
happy	16.8	16.8	47.3	40.8			47.3	46.0	47.1	39.8	50.4	50.3	50.1
blade	3.2	3.1	29.8	26.5	29.7	29.1	29.5	29.2	29.3	26.5	30.2	30.9	30.9
The average FPS using vertex buffer objects													
cow	461.4	562.8	548.1	520.4	535.9	530.8	548.2	539.4	532.5	509.2	553.0	553.4	548.6
demi	410.3	565.1	538.8	487.4	532.8	514.1	537.2	519.4	535.8	489.0	547.3	547.1	544.7
bunny	119.2	282.6	242.7	210.1	234.1	213.5	251.8	227.7	232.7	203.0	266.0	254.7	244.7
dinosaur	90.9	263.1	247.5	184.1	236.7	213.6	240.9	221.2	233.4	188.3	255.0	248.0	242.4
balljoint	37.2	129.8	114.6	82.3	111.9	97.5	115.0	103.7	108.6	87.3	123.4	121.0	117.4
club	27.7	111.1	93.7	61.4	91.5	79.0	93.9	83.5	88.6	68.2	103.2	100.5	94.1
hand	45.7	52.3	42.0	29.9			41.5	36.8	38.9	30.6	47.3	45.9	44.9
dragon	27.5	55.3	46.2	21.2			43.7	40.5	42.8	25.8	51.4	49.9	49.9
happy	27.4	54.1	45.1	17.2			41.2	38.3	39.5	20.9	50.9	49.3	49.5
blade	23.0	34.9	27.4	13.1	28.4	24.4	26.8	24.6	25.4	15.2	32.9	30.5	30.2
The computation time in seconds													
cow		0.3	0.0	0.0	0.0	0.0	0.2	0.2	0.0	0.0	0.4	0.0	0.0
demi		1.0	0.1	0.0	0.0	0.0	0.7	0.7	0.0	0.0	0.6	0.1	0.1
bunny		4.8	0.3	0.2	0.2	0.2	2.3	2.4	0.2	0.2	68.1	0.4	0.6
dinosaur		8.1	0.5	0.3	0.4	0.4	3.6	3.6	0.3	0.3	28.3	0.6	0.9
balljoint		22.1	1.4	0.8	1.0	1.0	8.8	8.8	0.8	0.8	69.2	1.5	10.5
club		36.5	2.2	1.1	1.5	1.5	13.5	14.7	1.2	1.1	239.5	2.3	22.9
hand		61.0	3.1	1.6			29.1	28.8	1.5	1.6	244.8	3.3	8.9
dragon		90.2	4.4	2.4			28.4	28.2	2.3	2.4	494.4	4.7	102.0
happy		121.1	5.7	3.2			35.0	34.7	2.9	3.0	673.9	5.9	124.0
blade		232.0	9.0	5.3	5.7	5.6	99.4	99.5	4.4	4.5	2514.9	9.4	60.2