Iterative Stripification of a Triangle Mesh: Focus on Data Structures

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

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

ABSTRACT

In this paper we describe the data structure and some implementation details of the *tunneling algorithm* for generating a set of triangle strips from a mesh of triangles. The algorithm uses a simple topological operation on the dual graph of the mesh, to generate an initial stripification and iteratively rearrange and decrease the number of strips. Our method is a major improvement of a proposed one originally devised for both static and continuous level-of-detail (CLOD) meshes and retains this feature. The usage of a dynamical identification strategy for the strips allows us to drastically reduce the length of the searching paths in the graph needed for the rearrangement and produce loop-free triangle strips without any further controls and post-processing, while requiring a more sophisticated implementation to manage the *search* and *undo* operations.

Keywords

Computational Geometry and Object Modeling -- Geometric algorithms, languages, and systems.

1. Introduction

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

Since the CPU-GPU communication tend to be the most common bottleneck of the whole visualization process, it is evident that a good stripification strategy could virtually improve by a factor of three the CPU-GPU bandwidth (at the best case, when a single strip, representing the whole mesh, is produced) and, consequently, the whole visualization.

Unfortunately it has been proven [Gar76, Ark96] that a problem equivalent to searching the optimal single strip (finding a Hamiltonian path on the dual graph) is an NP-complete problem, thus the stripification process should be based on local heuristics.

In a previous paper [Por03] we presented a solution to the generation of a stripification based on the *tunneling operator*. This is a single topology operator that we apply to the dual graph of the triangulation and allows us to either optimize an existing stripification or to generate a new stripification from scratch. The implementation of the algorithm relies on a single relevant parameter, the *tunnel length*, which influences both the time spent to stripify the mesh and the final set of strips obtained (number and mean length). Thus is very easy to use even for non expert users.

In this work we present in finer detail the algorithm and the data structures we used to implement it.

The rest of this work is organized as follows: in section 2 we briefly go over the previous work done in stripification; we then show, in section 3, the relations existing between the triangle mesh and its dual graph, introduce the tunneling operator and our solution to the problem; section 4, the most relevant in this context, is dedicated to describe in detail the implementation of the algorithm and the data structure and techniques used for doing it; finally, in section 5 we draw our conclusions and describe the future evolutions of this work.

2. Previous Work

The so-called stripification techniques are a subset of all the techniques devised in recent years to face the problem of *compressing the geometry*, that is finding

WSCG POSTERS proceedings
WSCG'2004, February 2-6, 200 , Plzen, Czech Republic.

good strategies and algorithms from them to reduce the space needed to describe a mesh (tipically a mesh of triangles) in terms of vertices position and connectivity.

Synthetically, to generate a strip from a mesh means to rearrange the order in which the vertices are stored. The strips obtained are smaller than the original mesh when coming to the final rendering since, while the single triangle needs 3 vertices for its visualization to be sent to the GPU, the triangle strip needs n+2 vertices to be sent to the GPU to render n triangles. The optimal single strip encoding the whole mesh would reduce the number of vertices sent to the GPU by a factor of three.

The great advantage of using triangle strips consists of the availability of such a primitive in the OpenGL graphics library. Generating a stripification of a mesh means to be able to feed the GPU with the obtained structure without any further effort. It is actually to point out that OpenGL supports, without any vertex replication, only the sequential triangle strips. Generalized strips could thus get us to send more than once some vertices to the GPU. It is beyond the scope of our current implementation to tackle this problem, but we plan to investigate this.

This explains why a lot of effort has been spent in elaborating good heuristics to stripify a mesh [Eva96, Cho97, Spe97, Xia99, ElS99, ElS00, Ise01, Est02].

It is worthwhile to explicitly mention a technique [Hop99] that uses a greedy algorithm to take advantage of the caching strategy of the graphics boards, thus differentiating in a way from the others cited, and the work [Ste01] that first proposed to use the tunnelling operator on the dual graph, which is described in detail in the next section.

3. The Triangle Mesh and its Dual Graph

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

For the details regarding the tunneling operator and our approach to the solution we suggest to read the works of Stewart [Ste01], refined by ourselves [Por03]; we give here just a description of the data structure.

The Data Structure

We now focus our attention on the description of the data structure used as a support to the implementation of the tunneling algorithm, and on some of the most interesting details of the implementation itself.

You will not find, instead, any detail of the functionalities related to visualize the three-dimensional scene since it is not interesting for the purposes of this work.

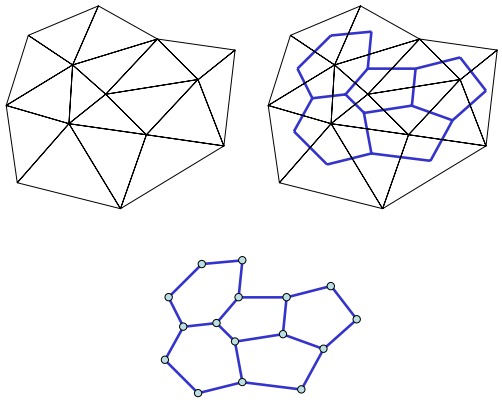

Figure 1: A triangle mesh and its dual graph.

Our implementation of the tunnelling algorithm is not trivial, since it requires that the supporting data structure should be able to keep track of the dynamical identifier update. Moreover, since the data structure should support an *undo* operation, we should, in fact, be able to discard any change, if the tunnel search does not end successfully.

First of all, it is worthwhile to tell that we chose the OO paradigm to be the development framework to be used. Alongside with obvious readability, manageability and reusability issues we thought that the encapsulation given by OOP was very useful to abstract the behaviour of the data structures we used.

As we described before, the tunnelling algorithm works on the mesh's dual graph. Its implementation requires first to find a terminal node for a path in the graph and then to traverse the graph (i.e., the structure representing it) to find a complementable path (a path alternating *solid* and *dashed* links, beginning and ending with a *solid* one). We thus need a robust support for the traversal allowing us to get a snapshot of the situation, step by step during the algorithm execution.

We can see a sketch of the structure in Figure 2. The graph nodes (corresponding one to one to the triangle in the mesh) are each represented by an object instance of the class Triangle, encapsulating the geometrical (vertices position) and topological (connectivity, i.e. the links between adjacent triangles) information.

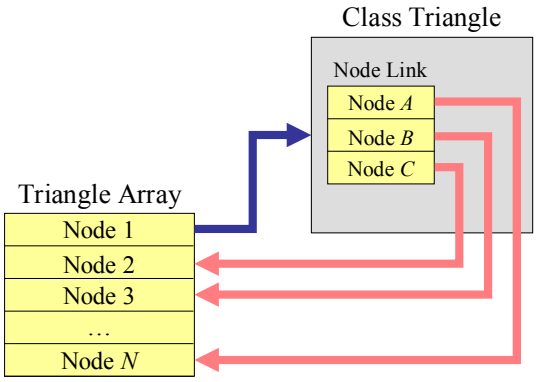

Figure 2: A simple sketch of the supporting data structure: the Triangle Array is an array of object

instance of the class *Triangle* that encapsulates the behavior of a triangle in the mesh (a node in the dual graph).

The whole graph is stored in a mono-dimensional array, each entry representing a node. The links are indices in the array. This allows us to randomly access a single node in a more proficient way compared to an implementation using pointers.

Since the core of the algorithm implementation can be found inside the class Triangle, we can actually tell that the class methods are the operations on which the algorithm relies on. What is really interesting to analyse is the search method that performs a recursive visit of the graph.

The internal structure of the Triangle class is depicted in Figure 3. The main information on connectivity is stored in the integer triplet (*Vert1, Vert2, Vert3*) that is used to find the positions of the triangle vertices in the coordinate vector. To easily access the neighbour triangles we keep a pointer to the three adjacent triangles (in this case we mean edge adjacency). Following these links we can pass in one step from one triangle to the adjacent one, or, thinking about the graph, from one node to another which is linked to it. Recall that, at most, each node in the graph has three neighbour nodes, that is three departing/arriving links (our graph is not oriented). We can thus randomly access the triangle list and traverse the graph using the same data structure that is, at the same time, a linear array with the properties of a double linked list.

Since we need to mark each link in the graph as *solid* or *dashed*, we associate a Boolean flag to each link: 1 for solid and 0 for dashed. Each *Triangle* object has also the integer identifier of the stripe it belongs to. The identifiers are unique for each strip.

4. Implementation details

Our tunneling algorithm uses a strategy that we called dynamic update of the identifiers. It does not actually dynamically change the stripe identifiers associated to each node, but also the flags for each link traversed by the searching procedure. Since the traversal does not always get us to find a complementable path, the dynamic update should consider the possibility of *undoing* the whole traversal (both on the identifiers and the colours of the links). We should *commit* the modification only when we are sure to have found a correct path. To support this, in the *Triangle* class, we duplicate the flags associated to the links (three for each node). These *temporary* flags are used only during the complement attempt, when the path is found, the changes are committed and the flag reset. Since each node of the graph can be visited as *primary* only once, also each link state can be changed once, and thus we need no more than one temporary flag per link.

What about the identifiers? Since each node can be visited as secondary as many times as possible, the identifier can change the same number of times. If we want to be able to undo the whole operation we need to use a *stack* of identifiers. We push a new identifier when we pass trough the node and we pop the top identifier during the undo process. Note that we are not obliged to abort the whole traversal, we can backtrack it to a consistent state and follow a different path.

Let us now see a description of the most relevant methods of the class *Triangle*.

The method **`IsTerminalNode(…)`** returns a boolean: 1 if the triangle is strip terminal and 0 if it is not. Its implementation is based on an analysis of the colours of the links to the node: less than two solid links means that the node is terminal, two solid links means that the node is non-terminal, three solid links are not possible by construction. This allows the method to work correctly whatever changes to the graph we made.

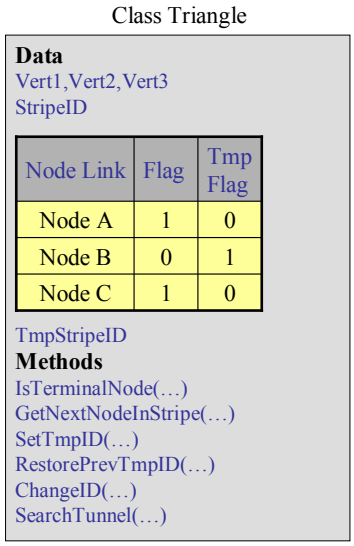

Figure 3: A more detailed example of the class *Triangle*.

The method **`GetNextNodeInStripe(…)`** returns the next triangle in the strip. If the node is non-terminal we can go up and down the strip and thus there are two next triangles, the direction os a parameter of the method.

We use the methods **`GetTmpID(…)`**, **`SetTmpID(…)`**, **`RestorePrevTmpID(…)`** and **`ChangeID(…)`** to manage the temporary identifiers during the tunnel search. The second one sets the identifier to a temporary value and the third set it back to the previous one. Note that setting the identifier means to set also the identifiers of the nodes linked to the changed node by a chain of solid links. Both methods follow this identifier propagation strategy. The last method is the one used to commit the changes when we find a good path.

The **`SearchTunnel(int p)`** method is the actual activation of the searching mechanism. The parameter **`p`** is the maximal tunnel length we want to have. It is a recursive method and the recursive step is performed decrementing **`p`**. The search strategy so defined is

actually a breadth-first search in the graph and takes into account only the path satisfying the imposed bounds. It can change and, in case no path is found, restore the state and restart using the structure and methods described before.

5. Conclusions and Future Work

We described a stripification algorithm based on a simple topological operation on the dual graph of the triangle mesh that is robust and easy to use. We focused our attention here on the data structures used and on the implementation details. The results obtained, which we still consider preliminary, make us confident that we shall be able to implement a version of the algorithm capable of operating also on CLOD meshes. It could be used to repair the inconsistencies in the stripification of an LOD when inserting new triangles. The choice of the search seeds is still an open issue. We plan to elaborate on strategies different from the current ones that choose randomly the starting node and move at random in the graph. One goal of such a strategy should also be the generation of strips being as sequential as possible, to accommodate the current requirements of the graphics libraries.

We also plan to investigate the limits of the stripification algorithm when applied to huge meshes, eventually adopting an out-of-core scheme allowing the stripification of meshes of any size. This will obviously result in a major reconsideration of both supporting data structures and implementation strategy, to be able to adapt the algorithm to work on the partition of data present in core memory at a given moment in time.

Currently the algorithm is, in principle, global over the dataset (the graph), splitting it in parts could bring one of two branches: a localization of the search strategy that, nowadays, is totally unclear whether it could be feasible or not, or a pagination strategy for the dataset.

6. References

[Ark96] ARKIN, E. M., HELD, M., MITCHELL, J. S. B., AND SKIENA, S. S. Hamiltonian triangulations for fast rendering. The Visual Computer 12, 9 (1996), 429–444.

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

[ElS99] EL-SANA, J. A., AZANLI, E., AND VARSHNEY, A. Skip strips: Maintaining triangle strips for viewdependent rendering. In *IEEE Visualization '99* (Oct. 1999), pp. 131–138.

[ElS00] EL-SANA, J., EVANS, F., KALAIAH, A., VARSHNEY, A., SKIENA, S., AND AZANLI, E. Efficiently computing and updating triangle strips for real-time rendering. *Computer-Aided Design 32*, 13 (Oct. 2000), 753–772.

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

[Eva96] EVANS, F., SKIENA, S. S., AND VARSHNEY, A. Optimizing triangle strips for fast rendering. In *IEEE Visualization '96* (Oct. 1996), pp. 319–326.

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

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

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

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

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

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

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