Constrained Strip Generation and Management for Efficient Interactive 3D Rendering

Pablo Diaz-Gutierrez* Anusheel Bhushan† M. Gopi‡ Renato Pajarola§

Computer Graphics Lab
University of California, Irvine

Abstract

Representing a triangulated two manifold using a single triangle strip is an NP-complete problem. By introducing a few Steiner vertices, recent works find such a single-strip and hence a linear ordering of edge-connected triangles of the entire triangulation. In this paper, we highlight and exploit this linear order in efficient triangle-strip management for high-performance rendering. We present new algorithms to generate weighted single-strip representations that respect different constraint-based clustering of triangles. These functional constraints can be application dependent; for example, normal-based constraints for efficient visibility culling or spatial constraints for highly coherent vertex-caching. We also present a hierarchical single-strip-management strategy for high-performance interactive 3D rendering.

CR Categories: I.3.5 [Computer Graphics]: Geometric Algorithms—Triangulation Stripification; K.7.m [Graph Algorithms]: Hamiltonian Cycle—Weighted Perfect Matching

Keywords: Single-strip, Weighted Perfect Matching, Hamiltonian Cycle, Vertex Cache, Visibility Culling.

1 Introduction

Triangle strip representation of a model has been traditionally used for efficient rendering. Vertex caching techniques to render these triangle strips improve coherence in memory access and boost the performance further. A *generalized* triangle strip is an edge-connected sequence of non-repeating triangles. In order to correctly render such strips, non-alternating vertices might have to be repeated or "swap" commands have to be used, if available. Recently, due to the availability of larger vertex caches in the graphics accelerators, remarkable performance increases can be achieved using generalized triangle strips. In this paper, we generate and manage generalized triangle strips.

Finding a single generalized triangle strip covering the entire model, without modifying the model is NP-complete. Hence, traditionally in computer graphics, multiple triangle strips are used to represent a model. Recent works [14, 13] introduce a small number of additional triangles to find a single strip representation, and hence a linear ordering of the triangles of the entire model. Applications of triangle strip representations include generation of space filling curves [14] and fundamental cycles [13] on manifolds, as well as unfolding of triangle strips for origami [19].

*e-mail: pablo@ics.uci.edu
†e-mail: anusheel@ics.uci.edu
‡e-mail: gopi@ics.uci.edu
§e-mail: pajarola@acm.org

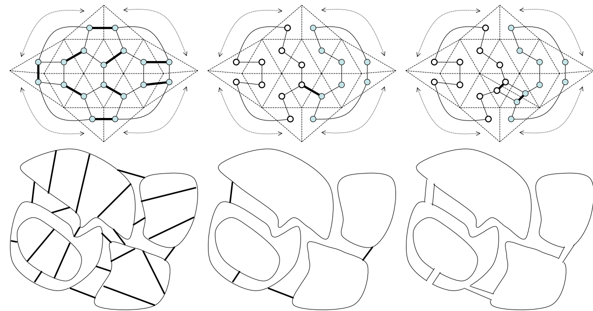

Figure 1: **Top:** (a) The dual degree three graph of the triangulation of a genus 0 manifold and a perfect matching shown by dark edges. (b) The set of unmatched edges create disjoint cycles. Two such cycles are shown. These disjoint cycles are connected to each other by matched edges. The algorithm construct a spanning tree of these disjoint cycles and hence choose matched edges that connect these cycles. (c) The triangle pair corresponding to chosen matched edges in the tree are split creating two new triangles. Matching is toggled around the new (nodal) vertices resulting in a triangulation with a Hamiltonian cycle of unmatched edges. **Bottom:** (d-f) A generalized example of the same process shown just on the dual graph. [14]

In rendering, longer triangle strips generally yield higher performance. In this paper, we show the benefits of the linear ordering of the triangles provided by the single strip representation, which go beyond obtaining a higher frame rate. Some of the advantages include simplicity in the data structure, efficiency in data management, and elegance of the algorithms for high performance rendering applications.

Most computer graphics applications benefit from early discarding of information that will not contribute to the final result. For example, not processing back-facing triangles can save almost half of the GPU bandwidth, which can boost the frame rates of interactive applications, even though current day hardware culls them very efficiently. Moreover, interactive graphics applications significantly increase their performance by making use of triangle strips. There are many existing algorithms for each of the mentioned applications. Our contribution is to create and manage efficiently the triangle strips that aid these rendering improvements.

Specifically, the following are the main contributions of this paper:

- We introduce a constraint-based single strip generation algorithm that can generate a single strip maximizing a functionally specified input constraint.
- We pose the back-face culling problem as a functional optimization problem and find a single strip that maximizes the spatial locality of similar-oriented triangles.
- We translate the patterns in the strip required for maximal vertex caching into a space-filling curve generation problem, and then cast it as a functional optimization problem for single strip generation.

- We also present an efficient strip management technique that uses the linear ordering of triangles provided by the single strip for interactive 3D rendering.
- Finally, we illustrate the generality of the method in terms of its ability to work with any arbitrary constraints, by combining the above constraints to achieve a strip that is designed both for smaller vertex cache-miss ratios and faster backface culling.

2 Related Work

As mentioned above, we note that the basic problem of finding an optimal set of triangle strips for a given triangulation is NP-complete [8, 10], and a large body of work has addressed the problem of heuristics to minimize the number of triangle strips for static triangle meshes [1, 11, 2, 27, 16, 25, 26]. Provably good and high-quality triangle strips have been reported in [27] and the *Tunneling* approach [25]. For real-time, continuously adaptive multiresolution meshes [18], it is much more important to compute a reasonably good set of triangle strips fast than to compute the optimal solution. In this context, methods such as *SkipStrips* [9] and [4] are based on an initial good stripification and subsequent management (mainly shortening) of incremental changes to the mesh, and hence the triangle strips. To reduce the overall shortening and fragmentation of adaptive strips, *DStrips* [24] manages triangle strips fully dynamically by locally growing, merging and partial re-computation of strips.

By introducing some extra data points, the *QuadTIN* approach [21] allows the representation and triangulation of any arbitrary irregular terrain height-field data set by one single triangle strip. Moreover, it supports dynamic view-dependent triangulation and stripification in real-time. However, QuadTIN [21] does not support nor maintain a specific initially given triangulation. Recent work on *Single-Strip* triangulations [14, 13] improves on that by finding a single-strip representation of a given manifold triangulation using only a low limited number of additional triangles.

Through the use of three vertex registers to hold temporary transformed geometry results, triangle strips have become an effective tool to improve rendering performance of large triangle meshes [1, 20]. Extending this concept, the efficient use of extra vertex registers has not only been exploited in geometry compression [7, 5] for bandwidth reduction, but also for improved rendering [15]. In [15], multi-register vertex caching is used to increase the locality of vertex references which reduces geometry transfer and transform costs significantly.

The presented approach using weighted-matching based single-strip representation of manifold triangle meshes seamlessly exploits extended vertex registers for fast rendering, and it allows for effective visibility culling. The single-strip representation and coherent vertex referencing further allows for a streaming-based rendering of large models.

3 Single-strip Creation

The problem of finding a single triangle strip is equivalent to finding a Hamiltonian path in the dual graph of a mesh. However, if we allow addition of a few Steiner vertices that do not change the geometric fidelity or the topology, we can find a single triangle strip. The algorithm presented by [14] is one such method that uses a perfect graph matching algorithm on the dual graph of the triangulated two manifold to create a single loop representation. Here, we briefly explain this algorithm for the sake of completion.

A matching in a graph is pairing a vertex with exactly one of its adjacent vertices. A perfect matching is one in which every vertex of the graph is matched. It is known from [22] that such a perfect matching exists for a 3-regular, 3-connected graph, such as the dual graph of a manifold without boundary. A perfect matching in its dual graph means that every triangle in the original mesh is matched with exactly one of its three edge-connected triangle neighbors. Triangle strip loops can be formed by connecting every triangle with its two unmatched neighbors. This yields, not one, but many disjoint strip loops.

Next, we iteratively join all the separate loops into one by means of two simple operations, each of which takes two or more loops and merges them. The first operation splits two adjacent triangles that belong to different loops (refer to Figure 1) and merges the loops into one. The second operation is called nodal-vertex processing. A *nodal vertex* with degree n is a vertex in the original mesh where n is even and the number of different loops incident on that vertex is $n/2$ (Figures 2(a), 2(b)). Around such a vertex, pairs of matched and unmatched triangles alternate. Swapping the matched and unmatched triangle relationship around a *nodal vertex* merges all the incident strip loops into one. Unlike triangle splitting, nodal-vertex processing merges loops without introducing additional faces to the mesh.

The main disadvantage of the above algorithm is that the strip growth is not controllable. In other words, unlike incremental strip growing algorithms [13], the above strip loop creation method can be neither locally or globally steered to satisfy certain constraints. In this paper, we introduce controllability to the above algorithm by using a *weighted perfect matching* method. We show that by imposing appropriate constraints for strip control, we can achieve well-behaved triangle strips that exhibit excellent properties for interactive high-performance rendering.

3.1 Weighted Perfect Matching

It can be shown that there are many different perfect matchings in the dual graph of a manifold without boundary, yielding many different single strip loops. In order to control the strip to satisfy certain properties, we have to control the choice of matching in the dual graph of the mesh.

We use a *weighted perfect matching algorithm* to find a matching that maximizes the sum of weights among the chosen matched edges. Higher weights indicate an edge more desirable to be matched, and therefore excluded from the strip. Maximizing the added weight of the chosen edges indirectly minimizes the total weight of the non chosen edges, which will form the single strip.

The weights of the edges of the graph (or of the edges of the mesh) are carefully chosen according to the application for which the strip is required. For example, to find a strip suitable for back-face culling, we would like to have neighboring triangles with similar normals to be the neighbors in the strip. Hence, the neighboring triangle with maximum normal deviation should be the matched triangle, thus that corresponding edge edge in the dual graph should be assigned a higher weight than the other two neighbors in order to be picked as a matched neighbor.

Assigning appropriate weights, by itself, is not enough to get the desired single strip. As in the case of the original algorithm, the matching yields many disjoint strip loops, each of which possibly satisfying the desired constraints. These disjoint loops have to be

combined into one loop while still preventing the strip to cross high weight matched edges.

3.2 Joining Disjoint Loops

There are only two ways to merge strip loops – nodal-vertex processing and edge (triangle) split operations. Both these operations reduce the overall weight of the chosen matched edges in the dual graph and hence the solution will be sub-optimal. The goal is to limit this reduction of weight as much as possible while merging loops into a single loop.

We assign a cost to each possible *nodal vertex processing* and *edge split* operations, and sort them in a priority list [23]. This cost is given by the decrease in the overall weight of the matched edges after a certain operation. The cost of a *nodal vertex processing* operation would be the difference between the sums of the weights of the matched and unmatched edges around the vertex. The cost of an *edge split* operation is twice the weight of the matched edge to split (refer Figure 2(c)) because this operation duplicates the originally matched edge, thus doubling the overall weight.

Valid operations correspond to edges connecting disjoint strip loops. Operations are popped from the front of the list and if they are valid, disjoint loops are merged. In general, many operations will be equivalent, in the sense that they join the same pair of loops. When one operation is applied, it invalidates all its equivalent operations which still remain in the list, and therefore will have no effect. The algorithm terminates when there are no more operations to be processed or all the loops have been merged.

The quality of the single strip created at the end of the algorithm is based on the order of the operations in the priority list. Variations of the above method to create different kinds of single strip can be achieved by giving priority to either one or other loop-joining operation. For example, some applications might have a high cost associated with the addition of a vertex. In this case, an *edge split* could be made more expensive than a *nodal vertex processing*.

We now look at a few applications of the above constraint-based stripification. We show how different weighing schemes lead to strips suitable for applications such as backface culling. Furthermore, we provide a simple framework for combining many weighing schemes to create one single strip satisfying multiple objectives.

4 VISIBILITY CULLING

In the first application of our technique, we explain how to create a triangle strip that is suitable for back face culling, and develop techniques for the per-frame management of the strip while performing the culling. We would like to note that in contrast to existing specialized visibility culling algorithms, our emphasis is on a means for creating and managing single triangle strips with the purpose of visibility culling.

First, we explain our scheme for assigning weights to the edges that are appropriate for efficient back-face culling. Then, we describe the data structure used to store the single strip returned by the algorithm elaborated in the previous section. Finally, we discuss the run-time management of this strip that integrates efficient back-face culling.

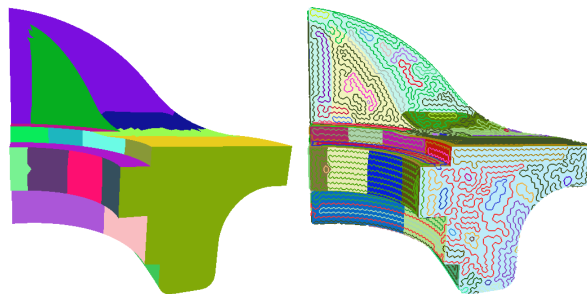

Figure 3: (a) **Left**: Triangle clustering based on normal deviation. (b) **Right**: Disjoint loops before merging.

4.1 Calculating edge weights

For efficient back-face culling, we want the strip to remain within planar regions for as long as possible, so high weights are assigned to sharp features of the mesh, while planar areas receive low weights. Local decisions on the sharpness of features might be misleading. For example, suitable refinement of triangulation can disguise a high curvature region into a low curvature region and vice versa. Hence, robust algorithms for feature detection base their decisions on global analysis of the model. We perform such an analysis on the model to cluster together triangles with similarly oriented normals. The output of this clustering algorithm is the input to our edge-weight assignment method.

Clustering Method: Identifying and building clusters of triangles that have similar oriented normals is a well studied problem ([12]). We use the *Variational Shape Approximation (VSA)* method, described in [6] that is popular for its simplicity and good results in face clustering. But this iterative method requires a reasonable initial estimate of clustering to converge quickly. For this we use a greedy approach to initialize the clusters for a given bound on normal deviation. The output of the *VSA* method is a clustering of triangles based on normal deviation (see Figure 3(a)). The number of clusters is dependent on an user-specified normal deviation tolerance. Furthermore, for each cluster, the average normal of the triangles in the cluster and the cone semi-angle, which is the maximum normal deviation from the average normal, are also computed.

Deriving edge weights: The clusters given by the *VSA* method represent regions through which the strip can grow without restrictions. Therefore, null weights will be assigned to edges separating two triangles within the same cluster. On the other hand, non-zero weights should be assigned to edges connecting triangles across different clusters. The value of these weights indicates numerically the undesirability of the strip to cross the associated border between clusters (see Figure 3(b)). Edges shared by two adjacent clusters with similar average normals receive a low weight, whereas edges connecting clusters with very different normals get higher weights. However, our experiments showed that this direct weighing schema gives excessive importance to the highest weighed edges and neglects those with a lower weight, but still not zero. Instead, taking the logarithm of the angle deviation between clusters as the actual weight resulted in strips that cross sharp edges less often. Once the triangulation edge weights are assigned, the dual graph with appropriate weights for the edges are used to find the single strip as explained in Section 3.

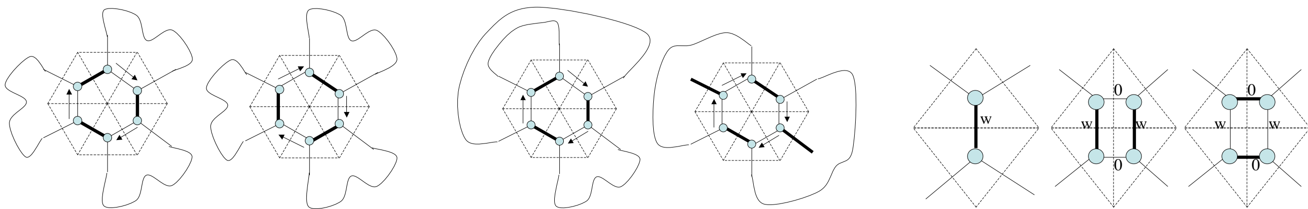

Figure 2: (a) **Left**: A nodal vertex with (six) even number of incident triangles and triangles belonging to three unique cycles. By switching the matched and unmatched edges, all these cycles can be merged to a single cycle. (b) **Middle**: Examples of non-nodal vertices. In both the examples, there are six incident triangles but only two unique cycles. (c) **Right**: Triangles before splitting; the weight of the matched edge is w. After splitting, the matched edge is duplicated with same weight w and the new unmatched edges have weight 0. After nodal vertex processing the reduction of weight is $2w$. [14]

4.2 Segment-tree data structure

In order for the single strip created by the method explained in Section 3 to be used in rendering applications, an appropriate data structure to store and access this strip has to be designed. We use a static hierarchical data structure, similar to the one described in [17], that stores in a node the result of merging the strips contained in its children. Hence, the root of the tree is the segment composed of the whole strip, and the leaves are the individual triangles of the mesh. Different horizontal slices of the tree consist of complete representations of the strip, split in segments of different granularity.

For every node of the tree, we compute the average normal deviation and the cone semi-angle of the contained triangles. The described segment-hierarchy has the desirable property that each node completely contains its two child segments, and nothing else (see Figure 4(a)). We can make use of this property to perform a recursive tree traversal and globally discard large, coherent back-facing portions of the triangle strip with just a few computations, without directly processing the individual triangles.

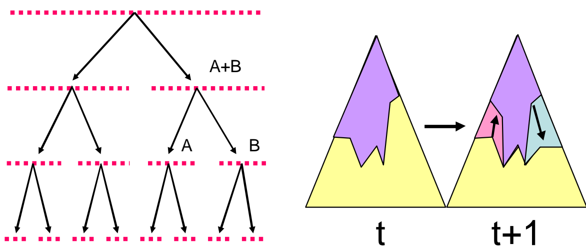

Figure 4: (a) **Left**: The segment-tree data structure. (b) **Right**: Evolution of the render front.

4.3 Segment-tree traversal

The nodes in the hierarchical tree data structure described in the previous section tend to contain contiguous strip segments composed of triangles with similar normals. We use this to our benefit by discarding or accepting large portions of the strip by only calculating a dot product. The most basic version of the rendering algorithm starts a recursive process at the root of the tree. In each node n, its average-normal and normal-deviation-angle are used to determine if the associated segment is *(a)*completely front facing, *(b)*completely back-facing or *(c)*somewhere across the silhouette of the rendered model. If the result is either *(a)* or *(b)*, then the segment is accepted or discarded, respectively. If there is uncertainty

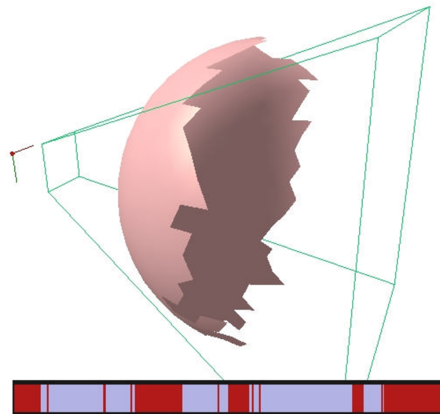

Figure 5: Real-time culling. The bar indicates visible (dark) and culled (light) parts of the strip. The high spatial coherence aids efficient culling.

(c), we go one step down in the hierarchy and check the two children separately. This process will continue down the tree until the processed segment is either discarded or rendered, or until the size of the segments is so small that checking segments takes longer than just rendering a few extra invisible triangles. If the segment has not been discarded, it is forwarded for rendering (see Figure 5).

In interactive applications, the difference in the position and orientation of the viewer in the displayed scene changes only gradually, hence the sets of rendered and discarded segments will be very similar in frames t and $t+1$. In order to exploit this coherence, we avoid traversing the tree for every frame by keeping an explicit *rendering front*, consisting of the lowest set of checked nodes from the last frame. In successive iterations, the process starts at the nodes in this rendering front, rather than at the root. Depending on the new point of view, these nodes are then split into their children, or merged up with their siblings, as shown in Figure 4(b). It is likely that most of the nodes in the rendering front will remain unmodified across a number of frames, thus saving traversal time.

4.4 Results

It is well known that the advantage of reducing the number of strips wanes as the total number of existing strips reduces because the amortized cost of starting a new strip becomes less important. However, keeping a *single* single strip along with a hierarchical structure of segments (Section 4.2) makes the cost of traversing it logarithmic on the number of faces. Instead of a single strip, if we maintain

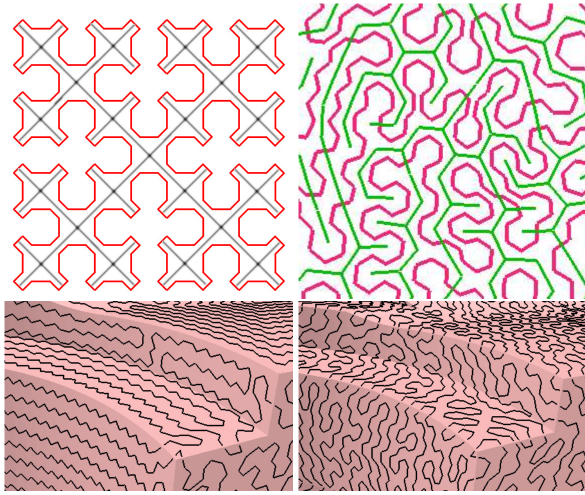

Figure 6: Left to right and top to bottom: (a) Sierpinski curve and its “medial axis”. (b) Our breadth-first tree and associated strip. (c) Single-strip on fandisk model, constructed for backface culling. (d) Cache-oriented single-strip on same model.

m separate strips and associated structures, this cost would become $m \log \frac{n}{m}$, reaching linearity as m grows. Furthermore, the results in Table 2 show an increase in the frame-rates obtained for all the models when constraints are applied to aid visibility culling, which demonstrates the utility of such constraints.

5 Transparent vertex caching

In our second application, we construct triangle strips with improved vertex caching usage. Most modern graphics processors have a vertex cache to reduce the data movement, benefiting from the locality in vertex references. *Transparent cache optimization* as in [15] refers to reordering the strip to maximize the access to vertices already in cache. In this section, we present a weighing heuristic that produces a single strip, presenting reasonably good vertex cache behavior for an *arbitrary cache size*.

We provide an interesting insight that forms the foundation for creating strips with high vertex cache coherence. The edges in the triangulation that the strip does not cross can be considered as the ”medial axis” of the strip. It is important to note that for a single-loop representation of the manifold, this medial axis is a spanning tree of the vertices of the triangulation (or two trees in case of genus zero objects). The strip actually loops around the vertices of the triangulation that form the leaves of this medial axis tree, creating high vertex cache coherence for that particular vertex. Maximizing the number of such leaf vertices of the medial axis tree increases the overall cache coherence. A breadth first spanning tree with low depth and large fan-out would maximize the number of leaf vertices and hence the vertex cache coherence. This property is exhibited by classical *closed* space filling curves like Sierpinski. Its medial axis emulates a breadth first tree (refer to Figure 6). The medial axis of the strip loop in the triangulation corresponds to the matched edges in the dual graph. To summarize, if the sequence of matched edges in the triangulation emulates a breadth-first tree, then the strip that goes around it would emulate a space filling curve and hence will have high vertex caching properties. We use this observation in our algorithm to find a suitable strip.

We build a breadth-first tree on the edges of the mesh. Since each triangle will have exactly one matched edge, no more than one edge per triangle is added to the tree. Edges in the tree receive positive weights, and the rest get the weight zero. The *weight-maximizing perfect matching* chooses these positive-weighted edges following the shape of the breadth first tree, which ensures the resulting single-strip will have good vertex locality.

5.1 Results

An advantage of generating a space filling strip is that it shows good caching behavior irrespective of the cache size. Thus, the same strip will exhibit good cache behavior for different cache sizes. The cache-size independence is a desirable feature because it eliminates the need for the application programmer to know the details of the system where the program will be deployed. Although knowledge about the actual cache size enables some improvements ([15]), explicit optimization for a given cache size can result in highly non-optimal behavior for other cache sizes. Furthermore, as shown in Table 1, our cache-size independent method achieves *cache-miss rates* (number of vertex-cache misses divided by the number of triangles) near those obtained by [15], where the size of the cache is known beforehand. These results indicate that for commonly used cache sizes, a large percentage of vertices need to be fetched only once.

Model	16v	32v
Sphere	0.76	0.66
Fandisk	0.79	0.71
Horse	0.79	0.71
Buddha	0.75	0.68

Table 1: Cache miss ratio for various models and cache sizes.

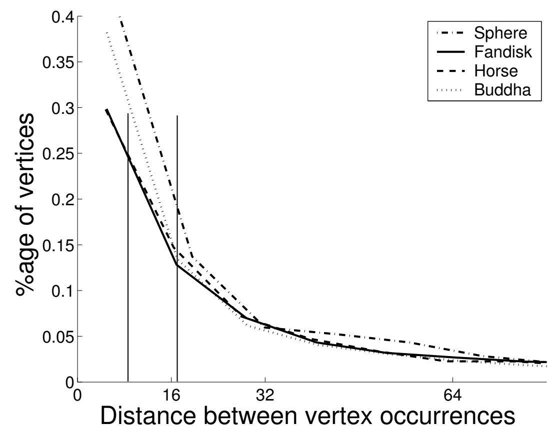

Figure 7: Histogram indicating percentage of vertices with different distances between successive appearances in the cache-constrained strips.

Figure 7 shows a histogram of the percentage of vertices having different maximum distances between successive occurrences in the single strip. Vertices with this distance equal or lower than d will cause no more than one cache miss in a cache of size d or larger, when using a *FIFO* replacement policy. It has been shown in [3] using an asymptotically optimal algorithm, that with a cache size of $12.72\sqrt{n}$, all n vertices are guaranteed to be in the cache. For the same cache-size, we observe that we achieve 90% of this result with our ‘space-filling’ stripification algorithm, even though it is not optimized for any specific cache-size.

6 Implementation and Discussion

The efficiency of matching methods is good enough for processing meshes of size in the order of hundreds of thousands of triangles in a few minutes, running on a current desktop computer. Significantly larger models would need a *patch-by-patch* treatment to be practical, because the algorithm's asymptotic efficiency is still super-linear. *LEDA's weighted perfect matching (WPM)* algorithm we used has a running time of $O(VE\log V)$.

We estimated the run-time improvement provided by our method on a set of standard models (Figure 9) representing manifolds without boundary. To do so, we moved the camera horizontally around the center of the models. Table 2 shows the measured frame-rate for each model, applying different constraints to the strip generation. For the sake of reference, the frame-rate has also been calculated for the models while being rendered using *GL_TRIANGLES* and an unconstrained single-strip. In the results, we appreciate a considerable performance increase for the larger models, when using constrained strips. The largest of all models (*Balljoint*, 274k faces) is an exception, showing not so relevant improvements, possibly due to memory thrashing. The last column reports the results of a strip constrained with a combination (explained in Section 6.1) of the two aforementioned targets (visibility culling and vertex cache optimization), and it demonstrates how the efficiency of the hybrid strip is always at least as efficient as the best single target strip.

Model (faces)	a	b	c	d	e
Sphere (1280)	1360	1540	2093	1226	2109
Fandisk (12946)	145	269	480	900	1000
Horse (96966)	38	99	332	304	340
Buddha (100k)	36	88	234	279	275
Balljoint (274k)	12	24	26	26	27

Table 2: Frame-rates for 5 models when rendered in a Pentium-4 2.4 GHz running GNU/Linux with a PNY 980XGL Quadro 4 as follows: (a) GL_TRIANGLES. The last 4 columns use single-strips with Vertex Buffers: (b) Unconstrained (c) Backface culling opt. (d) Vertex cache opt. (e) Hybrid of c and d.

6.1 Combining multiple targets

We have presented weighting schemes that are used to find a single-strip maximizing different functional constraints like face normal coherence and vertex-cache hit-ratio. However, finding a single-strip that maximizes multiple constraints simultaneously is a much more common scenario. For example, interactive rendering of large models would benefit from both reduced vertex cache-miss ratio and efficient visibility culling. Modifying a strip generation procedure to satisfy multiple constraints can be a much harder problem. The biggest advantage of our stripification method is that it is a two stage process in which the first stage consists of the user providing with the weights for the mesh edges and in the second stage the stripification is performed. Though the quality of the resulting strip is only as good as the scheme and accuracy of the weighting that the user chooses, the stripification algorithm itself is independent of both the weighting scheme and the assigned weights. If there are multiple, possibly contradicting, constraints then the user's weighting scheme should appropriately combine the constraints and assign numeric values to the mesh edges that reflect the relative importance of these constraints. In our experiments (with results reflected in the last column of Table 2), we computed the actual weight as a linear combination of the weights from each constraint.

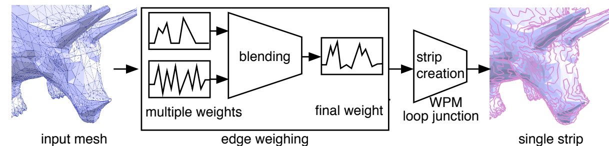

Figure 8: Single-strip creation pipeline.

Figure 9: Triangle meshes used in our work. **Top**: Fandisk, Balljoint. **Bottom**: Horse, Buddha.

7 Conclusion and Future Work

In this paper, we introduced a generic method for constructing constrained single-strips from a manifold mesh without boundaries. We have presented two mesh processing techniques that benefit directly from the use of constrained single strips. Finally, we outlined how multiple weighing techniques can be combined to obtain a single-strip under multiple constraints.

Most models we have experimented with in this paper (Figure 9) may fit completely into on-board video memory of the latest consumer graphics cards. However, large models require a view-dependent vertex-buffer management. Some investigation can be done on different priority schemes for loading segments of the strip on the GPU, so that parts of the strip that are expected to be required soon remain in the memory of the graphics hardware. Even off-core data could be tackled this way, with an appropriate paging mechanism.

Although the vertex cache results are satisfactory for our purposes, more time should be spent exploring alternatives to the given cache-optimizing weighing method. Namely, more precise space filling curves, which leave less freedom to the strip.

Finally, a whole set of classical problems have not been touched in our work. Two principal examples are mesh simplification and compression. For the former, certain mesh perturbations ought to be allowed, hence significant parts of our pipeline (Figure 8) shall be modified. For mesh compression, we identified a number of ways a single-strip could help. Weighting schemes can be devised aimed at minimizing the frequency of changes in vertex properties such as normal, color or material along the strip. Then standard compression techniques could be applied directly on the vertices of the single strip. Moreover, encoding the position of consecutive vertices in the strip would require fewer bits, given the expected

reduction in the intermediate distances.

References

[1] K. Akeley, P. Haeberli, and D. Burns. The tomesh.c program. Technical Report SGI Developer's Toolbox CD, Silicon Graphics, 1990.

[2] Esther M. Arkin, Martin Held, Joseph S. B. Mitchell, and Steven Skiena. Hamiltonian triangulations for fast rendering. *The Visual Computer*, 12(9):429–444, 1996.

[3] Reuven Bar-Yehuda and Craig Gotsman. Time/space tradeoffs for polygon mesh rendering. *SIGGRAPH 96*, 15(2):141–152, 1996.

[4] O. Belmonte, I. Remolar, J. Ribelles, M. Chover, C. Rebollo, and M. Fernandez. Multiresolution triangle strips. In *IASTED International Conference on Visualization, Imaging and Image Processing (VIIP 2001)*, pages 182–187, 2001.

[5] Mike M. Chow. Optimized geometry compression for real-time rendering. In *IEEE Vis. 97*, pages 347–354. IEEE, Computer Society Press, 1997.

[6] David Cohen-Steiner, Pierre Alliez, and Mathieu Desbrun. Variational shape approximation. *SIGGRAPH '04*, 23(3):905–914, 2004.

[7] Michael Deering. Geometry compression. In *SIGGRAPH 1995*, pages 13–20. ACM SIGGRAPH, 1995.

[8] Mike Dillencourt. Finding hamiltonian cycles in delaunay triangulations is np-complete. In *Canadian Conference on Computational Geometry (CCCG)*, pages 223–228, 1992.

[9] Jihad El-Sana, Elvir Azanli, and Amitabh Varshney. Skip strips: maintaining triangle strips for view-dependent rendering. In *IEEE Vis. 99*, pages 131–138. IEEE Computer Society Press, 1999.

[10] F. Evans, S. S. Skiena, and A. Varshney. Completing sequential triangulations is hard. Technical report, Dep. of Comp. Sci., SUNYSB, 1996.

[11] Francine Evans, Steven S. Skiena, and Amitabh Varshney. Optimizing triangle strips for fast rendering. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Vis. 96*, pages 319–326, 1996.

[12] Michael Garland, Andrew Willmott, and Paul S. Heckbert. Hierarchical face clustering on polygonal surfaces. In *I3D 2001*, pages 49–58. ACM Press, 2001.

[13] M. Gopi. Controllable single-strip generation for triangulated surfaces. In *Pacific Graphics 2004*, pages 61–69. IEEE, Computer Society Press, 2004.

[14] M. Gopi and David Eppstein. Single-strip triangulation of manifolds with arbitrary topology. *Computer Graphics Forum*, 23(3):371–379, 2004.

[15] Hugues Hoppe. Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH 1999*, pages 269–276. ACM Press/Addison-Wesley Publishing Co., 1999.

[16] David Kornmann. Fast and simple triangle strip generation. Technical Report, 1999.

[17] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH 1997*, pages 199–208. ACM Press/Addison-Wesley Publishing Co., 1997.

[18] David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers, San Francisco, California, 2003.

[19] Jun Mitani and Hiromasa Suzuki. Making papercraft toys from meshes using strip-based approximate unfolding. In *SIGGRAPH 2004*, pages 259–263. ACM Press, 2004.

[20] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison Wesley, Reading, Massachusetts, 1993.

[21] Renato Pajarola, Marc Antonijuan, and Roberto Lario. QuadTIN: Quadtree based triangulated irregular networks. In *IEEE Vis. 2002*, pages 395–402. Computer Society Press, 2002.

[22] Julius Peter Christian Peterson. Die theorie der regulären graphen. *Acta Mathematica*, 15:193–220, 1891.

[23] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[24] Michael Shafae and Renato Pajarola. DStrips: Dynamic triangle strips for real-time mesh simplification and rendering. In *Pacific Graphics 2003*, pages 271–280. IEEE, Computer Society Press, 2003.

[25] A. James Stewart. Tunneling for triangle strips in continuous level-of-detail meshes. In *Graphics Interface 2001*, pages 91–100, 2001.

[26] Petr Vanecek and Ivana Kolingerová. Multi-path algorithm for triangle strips. In *CGI '04*, pages 2–9, 2004.

[27] Xinyu Xiang, Martin Held, and Joseph S. B. Mitchell. Fast and effective stripification of polygonal surface models. In *I3D 1999*, pages 71–78. ACM Press, 1999.