The Visual Computer manuscript No.
(will be inserted by the editor)

Pablo Diaz-Gutierrez · Anusheel Bhushan · M. Gopi · Renato Pajarola

Single-strips for fast interactive rendering

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 extend previous results [10] that exploit this linear ordering in efficient triangle-strip management for high-performance rendering. We present new algorithms to generate single-strip representations that follow different user defined constraints or preferences in the form of edge weights. These functional constraints are application dependent; For example, normal-based constraints can be used for efficient rendering after visibility culling, or spatial constraints for highly coherent vertex-caching. We highlight the flexibility of this approach by generating single-strips with preferences as arbitrary as the orientation of the edges. We also present a hierarchical single-strip management strategy for high-performance interactive 3D rendering.

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

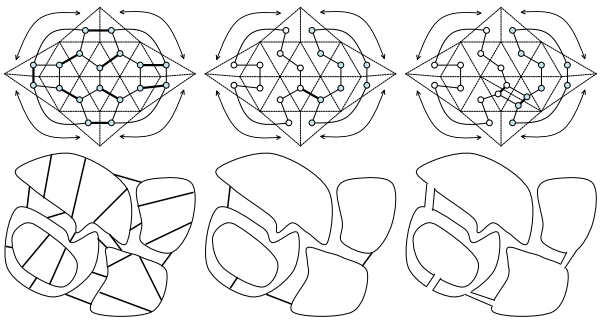


Fig. 1 Top, from left to right: (a) The dual degree three graph of the triangulation of a genus 0 manifold and a perfect matching shown by dark edges. (b) The set of unmatched edges create disjoint cycles. Two such cycles are shown. These disjoint cycles are connected to each other by matched edges. The algorithm construct a spanning tree of these disjoint cycles and hence choose matched edges that connect these cycles. (c) *Edge split* operation: 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, from left to right:** (d-f) A generalized example of the same process shown just on the dual graph. [19]

1 Introduction

Triangle strip representation of a model has been traditionally used for efficient rendering. Most interactive rendering packages support direct rendering of alternating triangle strips, in which vertices form triangles alternatively clockwise and counterclockwise. 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. In the former case, the number of sent vertices increases by roughly 50% on average. 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. On the other hand, for some applications it is not necessary to retain the original

Pablo Diaz-Gutierrez
University of California, Irvine
E-mail: pablo@ics.uci.edu

Anusheel Bhushan
University of California, Irvine
E-mail: anusheel@ics.uci.edu

M. Gopi
University of California, Irvine
E-mail: gopi@ics.uci.edu

Renato Pajarola
University of Zurich
E-mail: pajarola@acm.org

vertices and connectivity, as long as the geometry and appearance remains the same. Along that line, recent works [19,18] 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 [19] and fundamental cycles [18] on triangulated manifolds, as well as unfolding of triangle strips for origami [28].

Rendering triangle strips generally yields higher performance with longer strips. However, this improvement becomes smaller as the average strip length grows beyond a certain value. 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 by reducing the number of rendered vertices. Some of these advantages include simplicity in the data structure, efficiency in data management, elegance of the algorithms for high performance rendering applications and even other applications not necessarily linked to rendering, like mesh simplification and compression [12].

Most computer graphics applications benefit from discarding information that will not contribute to the final result. For example, culling back-facing triangles can save about half of the GPU bandwidth. Similarly, applications further increase their performance by using triangle strips. Here we are presenting a novel technique for efficiently culling back-facing triangles while having the remaining ones form strips as long as possible. We will see how the strips are used to improve the way we do the culling and, reciprocally, the way back-facing triangles are omitted helps maintaining long strips that contain the front-facing triangles.

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.
- 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 back-face culling.
- Finally, we discuss under what circumstances our approach for back-face culling or vertex cache improvement perform worse than expected, and comment on possible solutions.

In this paper, we discuss the creation of single-strips using the method from [19], its disadvantages and techniques to improve by assigning weights to the edges (Section 3). We formulate methods to assign these weights that would aid efficient rendering of front-facing triangles as strips (Section 4) and that would increase the vertex cache coherency (Section 5). It is also possible to combine the edge weights that are suitable for different applications to modulate and achieve a new goal (Section 6). The results of our experiments using these algorithms show a dramatic improvement in rendering efficiency (Section 7).

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 [13,15], and a large body of work has addressed the problem of heuristics to minimize the number of triangle strips for static triangle meshes [1,16,2,37, 22,34,35]. Provably good and high-quality triangle strips have been reported in [37] and the *Tunneling* approach [34]. For real-time, continuously adaptive multiresolution meshes [27], 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* [14], but also [4,12] are based on an initial good stripification and subsequent management (mainly shortening and strip merging) of incremental changes to the mesh, and hence the triangle strips. To reduce the overall shortening and fragmentation of adaptive strips, *DStrips* [33] manages triangle strips fully dynamically by locally growing, merging and partial re-computation of strips.

By introducing some extra data points, the *QuadTIN* approach [30] 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 [30] does not support nor maintain a specific initially given triangulation. Recent work on *Single-Strip* representations [19,18,11] improves on that by finding a single-strip representation of a given manifold using only a small number of additional geometric primitives, whether these are triangles, quadrilaterals or tetrahedra. In [11], the object of the stripification is not restricted to be a triangulated manifold, and the perfect matching is used as a step to find a 2-factor of a bounded degree graph. The work in [12] shows how linear ordering of triangles can be used not only for efficient interactive rendering, but also for mesh simplification and compression. Using cones of normals for a cluster of geometric primitives is common to several papers, like

[25,23], which uses them for mesh simplification through vertex clustering.

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 [1,29]. Extending this concept, the efficient use of extra vertex registers has not only been exploited in geometry compression [9,6] for bandwidth reduction, but also for improved rendering [21,5,38]. In [21], multi-register vertex caching is used to increase the locality of vertex references, which reduces geometry transfer and transform costs significantly. Similarly to our approach, [38] construct triangle strips that enhance the use of the vertex cache, regardless of its size. Further, [5,36] also explore the relationship between mesh locality and triangle strips.

The presented approach using weighted-matching based single-strip representation of manifold triangle meshes seamlessly exploits extended vertex registers for fast rendering, allowing for effective visibility culling. The single-strip representation and the high locality of its 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, which as we said above is known to be an NP-complete problem. 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 in polynomial time, with the drawback that the size of the processed mesh is larger after the stripification, due to the new vertices introduced. However, experimental evidence shows that this increase in the number of vertices is as low as 2%. The algorithm presented by [19] 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 [31] 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 use a fast greedy algorithm to iteratively join all the separate loops into one, by means of three operations, each of which taking two or more loops to merge them. After describing these operations, we will outline the greedy algorithm. The first operation, named *edge swap*, consists of a re-triangulation of a pair of triangles (as seen in Figure 2). A matched edge shared by two triangles –which belong to different strips– is removed. Subsequently, the two non common vertices from the affected triangles are connected by a new matched edge. This is a fundamental operations, because it is sufficient to merge all the loops. Another advantage is that it does not introduce new vertices to the mesh. The disadvantages come from the geometric implications of this re-triangulation: If the triangles across the edge are not coplanar, the surface represented by this triangulation will be different. Because of this, it should only be applied in planar regions and on pairs of triangles that form a convex quadrilateral. Otherwise, unacceptable model deformations could appear, such as flipped faces and dents in the surface.

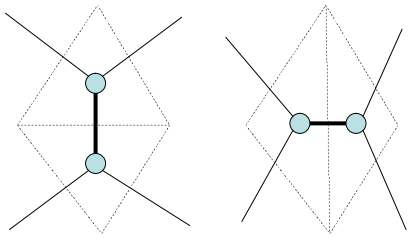


Fig. 2 Edge swap operation. Two previously disjoint strips are merged by re-triangulating around a separating edge.

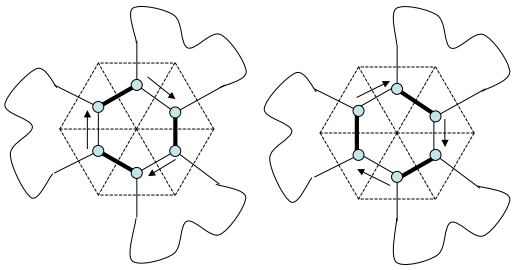


Fig. 3 A nodal vertex with (six) even number of incident triangles and triangles belonging to three unique cycles. By switching the matched and unmatched edges, all these cycles can be merged to a single cycle. [19]

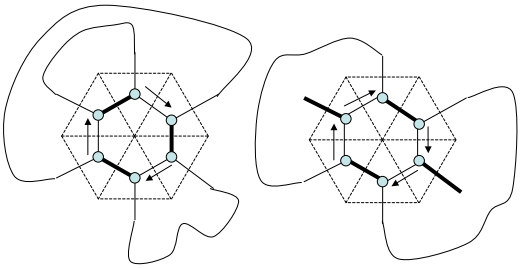


Fig. 4 Examples of non-nodal vertices. In both the examples, there are six incident triangles but only two unique cycles. [19]

The second loop merging 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 3, 4). Around such a vertex, pairs of matched and unmatched triangles alternate. Swapping the matched and unmatched edge relationships around a *nodal vertex* merges all the incident strip loops into one. This operation is generally preferred because, unlike the other two operations, nodal-vertex processing merges loops without modifying the mesh geometry.

The third operation, *edge split* (see Figures 1c and 5), introduces a Steiner vertex in the middle of the matched edge shared by two triangles. This vertex is connected to the opposite vertex of each triangle, much like in the *edge swap* operation, but without removing the initial matched edge. Note that the new vertex is a nodal vertex with 4 adjacent edges, so we can apply *nodal vertex processing* to join the loops. Just like *e*dge swap, this operation is sufficient to merge all the loops into one. However, *edge split* does not modify the appearance of the mesh, because the introduced vertex lies exactly on the separating edge. Its main disadvantage is precisely the introduction of a new vertex in the mesh. In general, operations that do not alter the mesh will be given preference, and we will usually use *edge split* when there is no other option. A rule of thumb to decide when to use *e*dge swap or *e*dge split is the following: If the two adjacent triangles are coplanar, or have a normal deviation below a user given threshold, *e*dge swap is preferable. Otherwise, we will use *e*dge split.

The greedy loop merging algorithm starts by finding all the initially valid loop merging operations. For each identified operation we compute a priority number, and sort them in a priority queue implemented as a skip list [32]. This priority favors operations that do not modify the mesh, and penalize those that introduce a larger geometric distortion. Operations are popped one by one from the queue, and it is checked whether they are still valid. A strip merging operation is valid if the loops it merges are disjoint. We use a union-find data structure to keep track of which loops have been already merged. Two operations are equivalent if they merge the same set of triangle loops. When one operation is applied, it invalidates all its equivalent operations which remain in the list. If the popped operation is valid, it is applied, and when the implied loops are merged this is recorded in the union-find data structure. The algorithm continues until all the loops have been merged, or there are no more operations left in the queue. If the mesh is a manifold without boundaries, there will be exactly one triangle strip loop for each connected component of the manifold.

The main disadvantage of the above algorithm is that the direction of the strip is not controllable. In other words, unlike incremental strip growing algorithms [18], the above strip loop creation method can be neither locally nor 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 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 from crossing high weight matched edges.

3.2 Joining Disjoint Loops

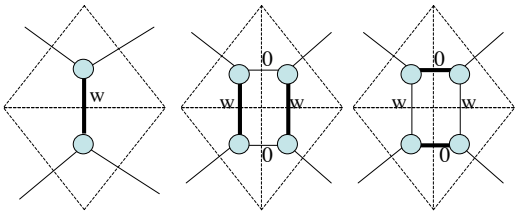


Fig. 5 Triangles before splitting; the weight of the matched edge is w. After splitting, the matched edge is duplicated with same weight w and the new unmatched edges have weight 0. After nodal vertex processing the reduction of weight is $2w$.

The three considered operations to merge strip loops –nodal vertex processing, edge (triangle) split and edge swap operations– reduce the overall weight of the chosen matched edges in the dual graph and hence the solution will be sub-optimal. Our goal is to limit this reduction of weight as much as possible while merging loops into a single loop. In order to do so, in the greedy loop merging algorithm we modify the priority or cost of each possible loop merging operation, before inserting them in the priority queue. In the unweighted version of this algorithm, this cost simply penalized operations that modify the mesh geometry. Now this cost must be also used to reduce the loss of weight in the perfect matching after applying the operations. Higher cost is given to operations that would produce higher loss of weight. This way, 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 nodal vertex. The cost of an *edge split* operation is twice the weight of the matched edge to split (refer Figure 5) because this operation duplicates the originally matched edge, thus doubling the overall weight. Finally we set the cost of an *edge swap* operation as twice the weight of the swapped edge. If the application supports weight recalculation at loop merging time, the weight of the new edge appearing after the operation would be subtracted from the cost of the operation. However, these are general guidelines that can be overridden, for example, if a particular application requires a merging operation to be always preferred over another one.

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 different loop-joining operations. For example, some applications might have a higher cost associated with the addition of a vertex. In this case, an *edge split* could be made more expensive than a *nodal vertex processing*.

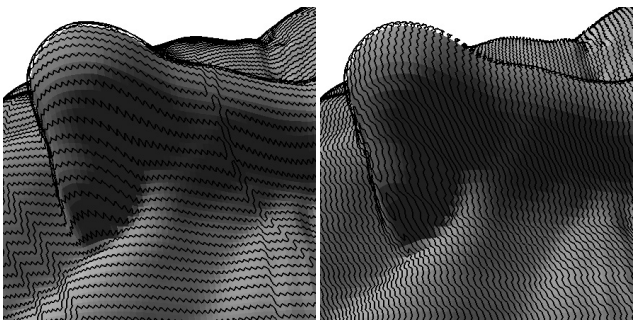


Fig. 6 Any weighing scheme can be used to indicate the preference for a particular type of single-strip. In the first figure (**left**), horizontal edges of the head model were assigned a high weight, while in the second figure (**right**), vertical edges were similarly penalized. The results are single-strips with different general orientation.

Using weights as a means to indicate the type of single-strip that we want gives the algorithm designer a great flexibility. For example, in order to compute the strips shown in Figure 6, we simply assigned weights based on the orientation of the edges of the mesh: If we assign high weight to mesh edges that are close to vertical, and zero to the rest, we get a single-strip which advances up and down, in the vertical direction. Analogously, if we assign high weight to mesh edges that are close to horizontal, and zero to the rest, we get a single-strip which moves left and right, in horizontal direction. In the remaining sections, we 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 stress 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, such that the front facing triangles are still rendered efficiently with long strips.

In this section, we first 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.

4.1 Calculating edge weights

A large category of visibility culling techniques group parts of geometry into spatially coherent clusters. Later, visibility is tested one cluster at a time, avoiding the cost of testing every geometric primitive individually. In case of back-face culling, the triangles are grouped based on their normal coherence. In the context of stripification, we would like the strips to remain within the the planar regions as long as possible. Next we will see how these two ideas (clustering primitives for visibility testing and making the strip remain within planar regions) complement each other.

Stripification for the back-face culling application has two advantages. First, long strips of triangles can be collectively tested for visibility. If the entire strip is facing backwards, all the triangles in the strip are culled. Second, all front facing triangles can be rendered as a strip, as they are already organized in a linear order. Similarly, retaining the strip in planar regions has two advantages. First, it enables collective orientation testing for triangles in the form of long strips, and

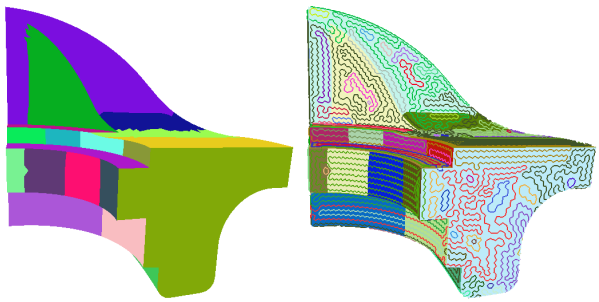


Fig. 7 (a) **Left**: Triangle clustering based on normal deviation. Triangles in the same cluster are shaded with the same color. Clusters minimize the normal deviation among the triangles contained. (b) **Right**: Disjoint triangle strip loops before merging. The weight maximizing edge matching produces strips that hardly cross the cluster boundaries.

second, when the strip is cut into pieces of front and back-facing segments, the number of such required cuts is minimized.

In order to achieve such a strip, suitable for back-face culling, high weights are assigned to edges that define sharp features of the mesh, while edges in planar regions receive low weights. Such a weighing favors edges defining high curvatures to be matched, and thus retaining the strip in the low curvature region. 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 similarly oriented normals is a well studied problem ([17]). We use the *Variational Shape Approximation (VSA)* method, described in [7] 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 7(a)). The number of clusters is dependent on an user-specified normal deviation tolerance. Finally, some geometric information is gathered from each cluster in order to aid efficient back-face culling. We will see this in detail later in this section.

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 how undesirable it is to have the strip cross the associated border between clusters (see Figure 7(b)). Although all cluster boundaries represent sets of edges that would rather be matched, this preference is stronger for certain boundaries. 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. The reason is that it is preferable to have the strip escape to an adjacent cluster with similar orientation that to another with a dramatically different average normal.

Our experiments show that using the deviation angle as a weight gives an inappropriate importance to the highest weighed edges while completely neglecting those with not much lower weight. The effect is that, in practice, all but the sharpest boundaries are ignored. Instead, if we take the logarithm of the cluster normal deviation as the actual edge weight, we produce strips that cross sharp edges less often. Once the edge weights are assigned, they are used to find the single strip, as explained in Section 3.

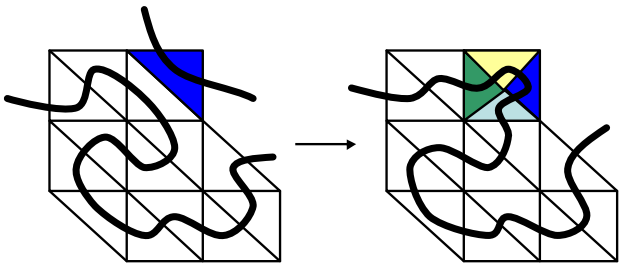


Fig. 8 Removing outgoing peaks from a cluster of triangles. The identified outgoing peak (in dark on the left figure) is split, together with an adjacent triangle inside the cluster. After the operation, no triangle in the resulting cluster has two boundary edges, and it is possible to reduce the number of times the strip crosses a cluster boundary (indicated with a thick black line).

Though the weighted matching algorithm greatly reduces the number of times the strip crosses the boundary of a cluster (by maximizing the number of boundary edges in the matching), there is a number of situations where such crossing is unavoidable. One example occurs when a triangle has two boundary edges, forcing at least one of them to be unmatched and allowing the strip to *escape* the cluster through it. This problem can be easily identified as a peak in the cluster, or *outgoing triangle*. The solution (illustrated in Figure 8) is splitting the outgoing triangle together with its neighbor from inside the cluster, at the cost of adding one vertex and two new triangles to the mesh. Depending on the average size of the clusters, and the number of strip crossovers, this can mean a great improvement in the resulting strip quality.

4.2 Segment-tree data structure

In order to use the single strip that we create as explained in Section 3 in interactive rendering applications, we must design an appropriate data structure to store and access this strip. We use a static hierarchical data structure, similar to the one described in [26], that stores in a node the result of merging the strips contained in its children. Hence, the root of the tree represents the segment composed of the whole strip, and the leaves are the individual triangles of the mesh. Different horizontal slices of the tree describe complete representations of the strip, split in segments of varying granularity.

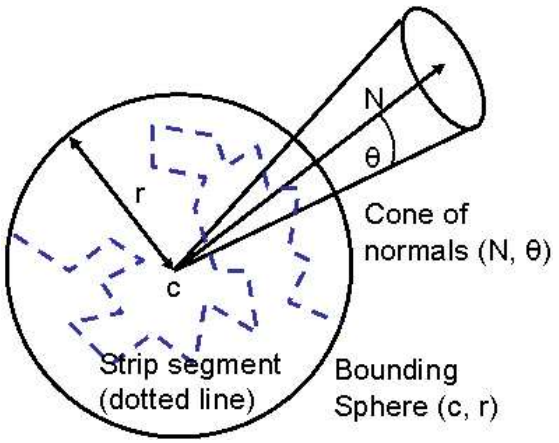


Fig. 9 Information associated to each segment of the triangle strip: Centroid of all vertices (*c*), bounding sphere radius (*r*), average face normal (*N*) and radius of the cone that contains all normals (θ).

We compute the following information for every node of the tree, along with its starting and ending position in the single-strip: Its average face normal *N*, the radius angle of the normal cone that contains all normals in the cluster θ, the centroid *c* of all vertices and the radius *r* of the smallest bounding sphere centered at *c* (Figure 9). All this information will be used for visibility testing at rendering time. Although it is not discussed here, the bounding sphere associated to each cluster can be used to perform frustum culling, if the hierarchical data structure is constructed appropriately.

The described segment-hierarchy has the desirable property that each node completely contains its two child segments, and nothing else. In other words, if a non-leaf segment-node starts at triangle *A* and ends at triangle *B*, its two children nodes must represent two consecutive triangle strips, the first one starting at *A* and the second one ending at *B* (see Figure 10(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.

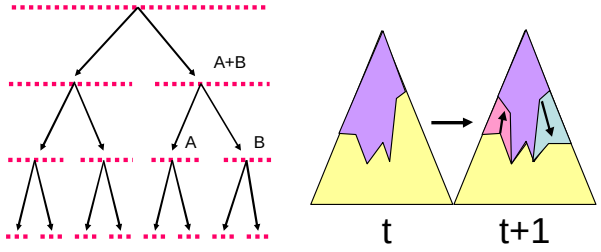


Fig. 10 (a) **Left**: The segment-tree data structure. Each tree node represents one segment of the single-strip. The union of all nodes at any given level comprises all the triangles in the model, with different granularity. The root node represents the whole single-strip. Note that the number of triangles in each node at the same level in the tree need not be the same. (b) **Right**: Evolution of the render front. As the interactive rendering progresses, some visible regions become back-facing, and vice-versa. The small differences between render fronts at successive iterations suggest storing this information to speed up rendering.

It can be argued that this hierarchical data structure ought to be balanced to enable the most efficient query times. However, generic tree balancing techniques are not applicable in our case. The reason is that, in general and for most models, not all visibility tests are equally likely. Visibility tests concentrate at the current silhouette of the model. Further, edges with higher curvature are more likely to be part of the silhouette than edges in planar areas of the model. These edges with high curvature should be accessible faster than others, hence should be closer to the root in the hierarchy. Since we construct the tree in a bottom-up manner, first merging parts of the mesh with low normal deviation, planar regions are grouped first, and regions separated by high curvature edges are not merged until the last stages of the construction. Thus our method for constructing the hierarchical data structure naturally provides short access time for the most frequent visibility queries.

4.3 Segment-tree traversal

Nodes in the hierarchical 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, 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. For 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 the segment can not be classified cleanly (case *c*), we go one step down in the hierarchy and test its two children independently. This

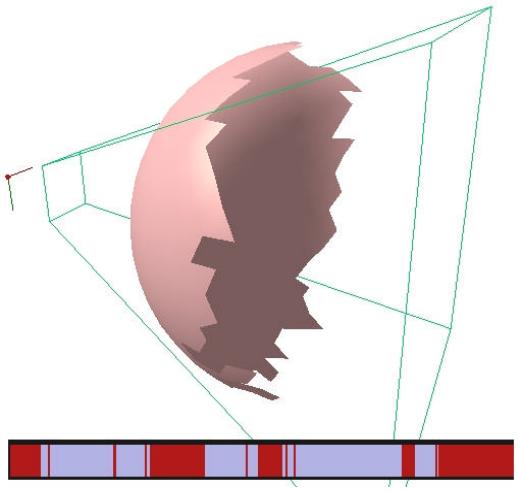


Fig. 11 Real time back-face culling in the sphere model. The bar indicates rendered parts of the single-strip, which are rendered, in dark color, and culled parts in light color. The high spatial coherence observed in the single-strip allows culling many triangles with few cuts.

process will continue down the tree until the processed segment is either discarded or accepted for rendering, or until the segments are so small that it is affordable to render them without further testing. In Figure 11 we represent the single strip as a colored horizontal band. Dark shaded segments in the band represent rendered triangles in the strip, while light segments indicate culled triangles. The coherence in the coloring of this band, which remains high during continuous movement of the viewpoint, demonstrates the relatively small number of visibility tests performed. A more quantitative evaluation of the strip coherence is given in Section 7.

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 consecutive frames. In order to exploit this coherence, we can avoid traversing the whole tree every frame by storing a *rendering front*, consisting of the lowest set of checked nodes from the last frame. In successive iterations, the process starts at the nodes in the 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 10(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 known that the advantage of reducing the number of triangle strips –as a means for limiting the bandwidth between CPU and graphics hardware– wanes as the total number of strips reduces. This is because the amortized cost of starting a new strip becomes less important with longer strips. However, keeping a *single-strip* along with a hierarchical structure of strip segments (Section 4.2) makes the cost of accessing a segment section logarithmic on the number of faces. If, instead of a single strip, we maintain m separate strips and their associated structures, the access cost becomes $m \log \frac{n}{m}$, rising towards linear cost as m approaches the number of triangles n. This is a clear disadvantage if we want to use the strips as Furthermore, the results in Table 4 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.

We have experimented with models of mostly smooth surfaces. In spite of using global clustering algorithms, rougher surfaces, with plenty of high curvature features, produce many small clusters, not so useful for efficient visibility testing. In other words, high frequency changes in the curvature reduce the quality of the strip when used for back-face culling. However, running a smoothing filter, such as a Laplacian filter, on the value of the normals used for weighing easily solves the problem.

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 [21] 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, inducing a high vertex cache coherence for that particular vertex. Hence maximizing the number of leaf vertices of the medial axis tree increases the overall cache coherence.

It is our goal to find a medial axis with the maximum possible number of leaves. Although a few algorithms have been proposed in the literature to find acyclic subgraphs with minimum number of non-leaves [20] and on the equivalent problem of maximizing the number of leaves [24], these algorithms are difficult to implement. We observed that the following simple heuristic worked well enough. A breadth first spanning tree with low depth and large fan-out would maximize the number of leaf vertices and hence the vertex cache

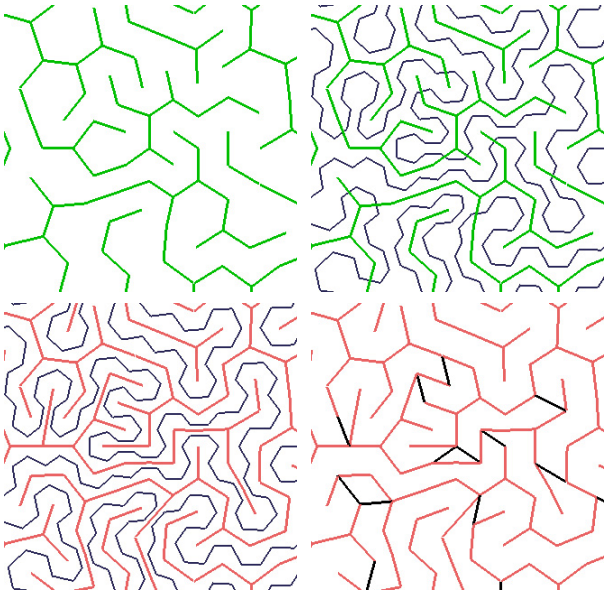


Fig. 12 Left to right and top to bottom: **(a)** Spanning tree of mesh edges, used to produce strips with low cache-miss ratio. Edges in the tree receive positive weight, and the rest get weight zero. **(b)** The produced single strip is superimposed on the spanning tree. **(c)** We substitute the spanning tree with the medial axis of the single strip. **(d)** Dark edges highlight the few differences between medial axis from 'c' and spanning tree from 'a'.

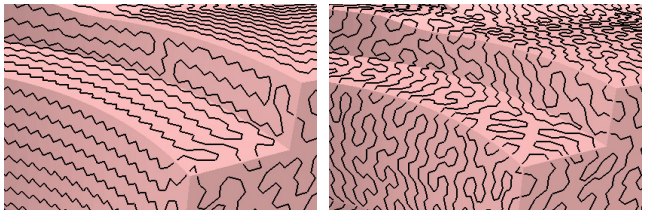


Fig. 13 Left: Single-strip on fandisk model, as constructed for backface culling. **Right:** Cache-oriented single-strip on same model. Notice the strip locality is much higher than in the other case.

coherence. This property is exhibited by classical *closed* space filling curves like Sierpinski's. Its medial axis emulates a breadth first tree (refer to Figure 12). 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, imitating the medial axis of a space filling curve. Our intention is to have the edges in this tree as matched edges. This forces the strip to follow the shape of a space filling curve whose medial axis is the computed breadth-first tree. Since each triangle will have exactly one matched edge, and we want as many of the edges to be matched, no more than one edge per triangle can be part of the breadth-first tree. Edges in the tree receive positive weights, and the rest get the weight zero. The *weight-maximizing perfect matching* chooses most of these non-zero weighted edges following the shape of the breadth first tree, which ensures the resulting single-strip will have good vertex locality.

5.1 Results

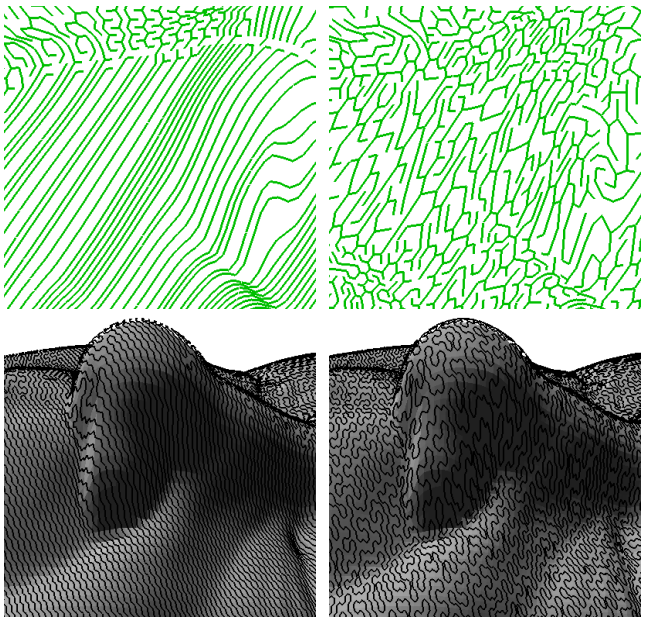


Fig. 14 Two versions of the spanning tree used for vertex cache optimization of single-strips, and the close-up on the resulting strips. **Top-left:** Breadth-first spanning tree. The high regularity of the head model produces long branches without bifurcation, taking a toll on the cache efficiency. **Top-right:** Randomized breadth-first spanning tree. It grows in an irregular fashion, producing shorter branches and more leaves. This reduces the cache-miss ratio of the resulting strip. **Bottom-left:** Strip resulting from breadth-first spanning tree method. Notice the low vertex locality. **Bottom-right:** Strip resulting from randomized breadth-first spanning tree method. The vertex locality is noticeable superior.

In most triangle meshes, the breadth-first tree growing procedure produces short branches with many bifurcations, and therefore many leaves, suitable for high vertex cache coherence stripification. However, this structure cannot always be generated with the simple spanning tree method. There are many triangle models obtained from height fields, for which each position in a regular 2D grid receives a height value. These meshes are extremely regular, and growing a spanning tree in a breadth-first manner might produce very few leaf nodes (see Figure 14). The single-strip obtained using this medial axis will have a very low cache-hit ratio. We solve this problem by introducing randomization in choosing the next edge to be added while growing the breadth-first tree. This breaks the symmetry in the deterministic tree growing

algorithm and introduces many branches, and hence leaves in the structure.

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. While knowledge about the actual cache size enables some improvements [21], explicit optimization for a given cache size can result in highly non-optimal behavior for other cache sizes. With graphics hardware vendors restricting information on their designs, and an increasingly large number of available GPU models, we expect such feature to gain further attention. The strips obtained with our cache-size independent optimization method achieve *cache-miss rates* (number of vertex-cache misses divided by the number of triangles) near those obtained by [21], with the difference that we do not assume anything about the size of the vertex cache. We also observe that [5] indicates comparable results. For example, with a cache size of 32 vertices, while [5] reports an average cache miss ratio between 0.6 and 0.68 for various models, our algorithm exhibits a value between 0.66 and 0.7. Our results indicate that for commonly used cache sizes, a large percentage of vertices need to be fetched only once.

Figure 15 plots the cache-miss ratios for the single-strips of three models, obtained with different weighing schemes (unconstrained strip, spanning tree of edges and randomized spanning tree). In all three cases, and for all reasonable cache sizes, our two cache-optimized strips produce significantly lower cache-miss ratios than the unconstrained strip. Notice that the theoretical lower limit for this ratio is 0.5. The heuristic followed by our optimization methods comes from the following: By increasing the locality of the single strip, we reduce the average distance between successive appearances of the same vertex. Vertices whose distance between instances is equal or lower than d will cause no more than one unavoidable 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 optimal performance, even though we do not know the specific cache-size beforehand.

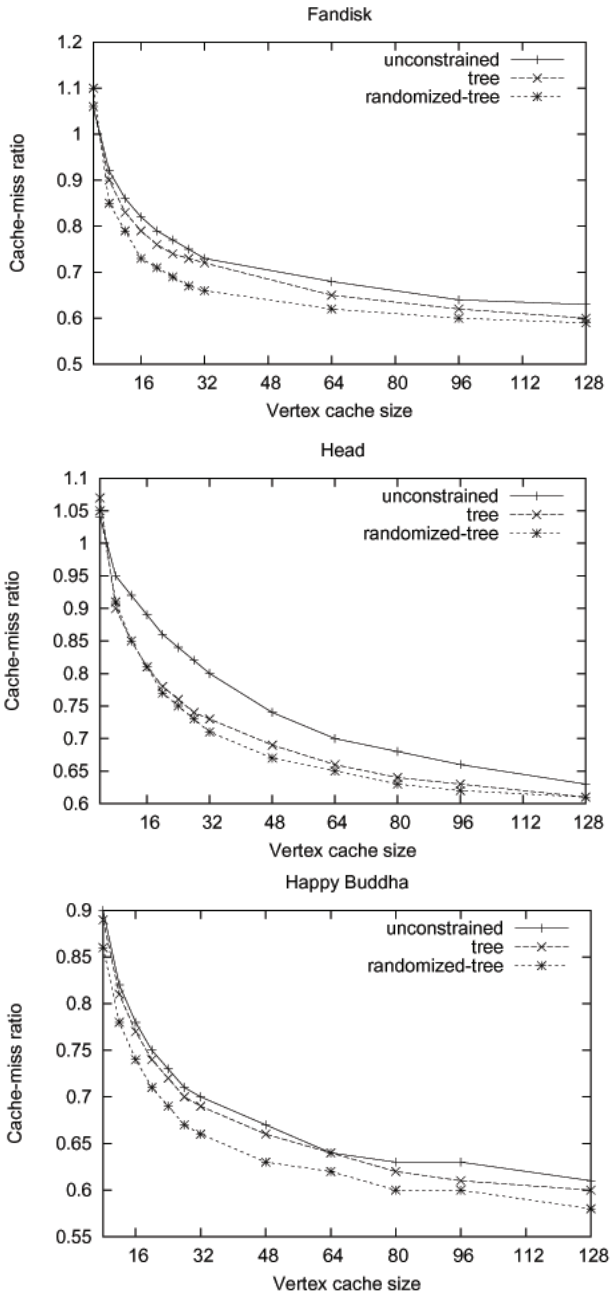


Fig. 15 Cache-miss ratio for the single-strips of three models, obtained in three different ways: Unconstrained strips, with the spanning tree method, and with the randomized spanning tree method. In all cases, the randomized version of our optimization performs better. Notice the specially large difference for the popular cache sizes of 16 and 32 vertices.

6 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 (see Figure 16). Although 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, we computed the actual weight as a linear combination of the weights from each constraint.

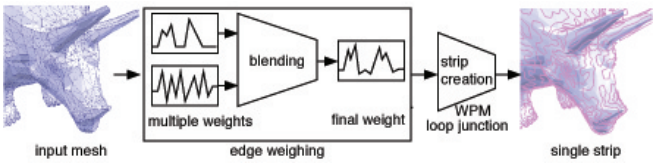


Fig. 16 Single-strip creation pipeline. The process comprises two stages. In the first one a weight is assigned to each edge, which produces a set of disjoint triangle strip loops. In the second stage, these loops are merged into one.

There is another interesting application of combining multiple weighing schemes. In back-face culling, the edges of the triangles belonging to planar regions receive zero weights, as the normal deviation across those edges is zero. Such a weight assignment drives the perfect matching algorithm for an exhaustive search to identify a critical point in the optimizing functional. A similar situation happens with many geometric optimization algorithms and in those cases randomization, or perturbation of the input data is a commonly used approach to bail the algorithm out of exhaustive search. Similarly, for the edges in the planar region, addition of small white noise to their weights tremendously accelerates the termination of the algorithm with almost no penalty to the quality of the resulting strip. For example, we saw the matching algorithm reducing its run time from more than 10 minutes to about 30 seconds, when random noise was added to the edge weights of the *Happy Buddha* model. Similar improvements were noticed with other models.

7 Implementation and Discussion

The creation time of the single strips was dominated by two principal algorithms: Perfect matching –common to all strips– and face clustering with Variational Shape Approximation –used for back-face culling. The efficiency of the matching method we used [8] is sufficient for processing meshes of size in the order of a million triangles in a few minutes, running on a current desktop computer. Treatment of significantly larger models needs an off-core approach to be practical. The face clustering algorithm runs in time comparable to that of perfect matching, empirically showing a super-linear time to convergence (see Table 1).

We estimated the run-time improvement provided by our method on a set of standard models (Figure 19) representing manifolds without boundary. To do so, we moved the camera in ascending spirals around the center of the models, as shown in Figure 17, rendering each model 591 times. All models were rendered using OpenGL vertex buffer objects. Tables 3 and 4 show the measured average frame-rates, applying different constraints to the strip generation. In the results, we observe the largest performance increase with the medium models, when using strips optimized both for cache coherence and back-face culling. Interestingly, the larger models show proportionally less improvement, probably due to memory thrashing.

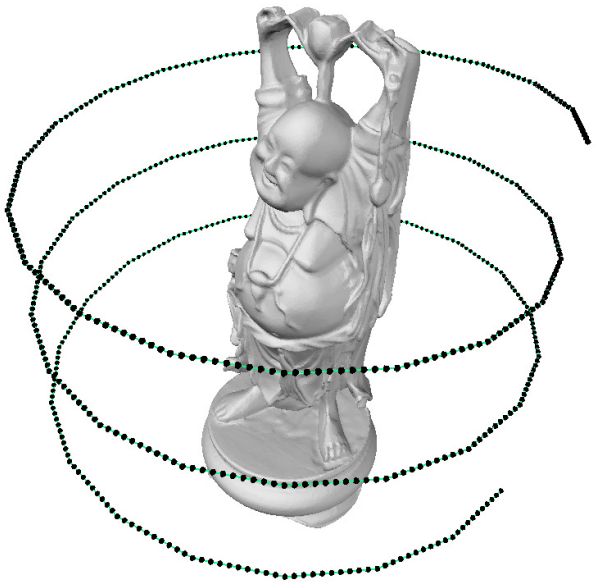


Fig. 17 Camera path used for measuring rendering frame rates. 591 frames are taken with the camera at the indicated positions in the ascending spiral, and looking towards the origin, at the center of the object.

Model	Face clustering	Perfect matching
Sphere	1	0
Cylinder	2	1.5
Trico	2	1
Fandisk	9	1
Head	14	2.5
Horse	48	7.75
Happy	32	120
Balljoint	132	13
Armadillo	210	50
Balljoint x4	593	103
Armadillo x4	991	828

Table 1 Preprocessing time: Average time (in seconds) spent in the two main preprocessing stages of our algorithm: Mesh face clustering for back-face culling optimization, and graph perfect matching of the dual graph.

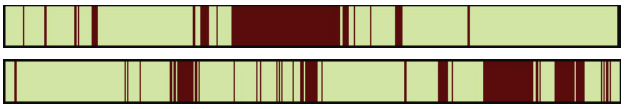


Fig. 18 Representation of the rendered and culled strips at a given moment while rendering the fandisk model. Dark segments of the bars indicate parts of the strip which were rendered; Light segments indicate culled parts of the single-strip. **Top:** Using a back-face culling optimized strip. **Bottom:** Using an unconstrained strip. The higher strip segment coherence in the first, optimized strip is apparent.

We have calculated the frame rates at which we can render our triangle strips, and used this as a measure of the quality of the strips. However, this method depends on the efficiency of our *ad-hoc* rendering system. It would be more appropriate to find a magnitude that can be evaluated independently of the implementation of the culling algorithm, and the machine used. A sensible measure that meets those properties is the number of continuous strip segments rendered. Given a back-face culling algorithm, a strip is well posed for back-face culling if it can be appropriately along the silhouette of the model using only few cuts. Therefore, when producing a single-strip suitable for back-face culling, our goal is to reduce the number of strip cuts at rendering time, while keeping the number of total vertices rendered low. In Table 2 we show the average number of strips rendered for several models when the camera moved along the spiral path of Figure 17, always looking towards the origin (center of the model). In all cases the back-face optimized strip needed fewer cuts before being sent for rendering. Similarly, this phenomenon can be observed directly if we represent in a colored bar the parts of the single strip which are rendered or culled, like in Figure 18. As expected, the lower number of cuts in the strip results in a more compact set of dark bands in the figure, representing fewer and longer strip segments being rendered.

Model	W/ Bf. opt	W/o Bf. opt
Cow	56	62
Cylinder	69	86
Fandisk	35	101
Head	178	221
Balljoint x4	3453	3886
Armadillo x4	3744	4498

Table 2 Strip coherence during backface culling: Average number of strips rendered for some models when strips were computed with and without back-face culling optimization. The lower number of strips in the first column indicates that in all cases the backface-optimized strips are better suited for quick culling than the unconstrained strips.

8 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.

Model	#triangles	a Unconstrained	b Span. tree
Sphere	1280	1951	1965
Cow	2218	1414	1486
Cylinder	4880	1181	1192
Fandisk	12946	537	733
Head	32744	226	227
Horse	96966	131	148
Happy	100000	110	117
Balljoint	274120	29	31
Armadillo	345944	24	25
Balljoint x4	1096480	9.2	10.2
Armadillo x4	1383776	6.33	6.80

Table 3 Rendering frame rates: The single-strips were obtained in the following manners: a) Unconstrained strips. b) Spanning tree method. All models were rendered in a Pentium-4 2.4 GHz running GNU/Linux with a NVidia PNY 980XGL Quadro 4 video card.

Model	#triangles	a Unconstrained	b Bf. culling
Sphere	1280	1760	3069
Cow	2218	1658	1751
Cylinder	4880	1181	1988
Fandisk	12946	538	1207
Head	32744	226	290
Horse	96966	31	93
Happy	100000	63	59
Balljoint	274120	18	34
Armadillo	345944	14.84	18.52
Balljoint x4	1096480	4.59	9.47
Armadillo x4	1383776	3.77	6.4

Table 4 Rendering frame rates: The single-strips were obtained in the following manners: a) Unconstrained strips. b) Optimizing for back-face culling. All models were rendered in a Pentium-4 2.8 GHz running GNU/Linux with a NVidia GeForce FX 5900 video card.

Most models we have experimented with in this paper (Figure 19) 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. A key part of any off-core stripification algorithm designed for gigantic meshes is handling the boundaries generated by the subdivisions that make the model manageable. The presence of boundaries adds a new level of complexity to the stripification procedure, and poses an interesting challenge.

Finally, weighing 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 com-

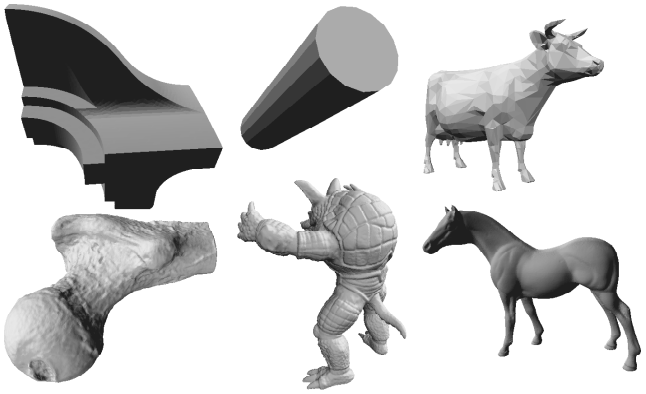


Fig. 19 Some of the triangle meshes used in our work: **Top:** Fandisk, Cylinder, Cow. **Bottom:** Balljoint, Armadillo, Horse.

pression 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.

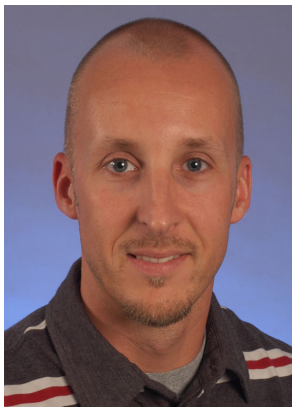
References

1. Akeley, K., Haeberli, P., Burns, D.: The tomesh.c program. Tech. Rep. SGI Developer's Toolbox CD, Silicon Graphics (1990)
2. Arkin, E.M., Held, M., Mitchell, J.S.B., Skiena, S.: Hamiltonian triangulations for fast rendering. The Visual Computer **12**(9), 429–444 (1996)
3. Bar-Yehuda, R., Gotsman, C.: Time/space tradeoffs for polygon mesh rendering. SIGGRAPH 96 **15**(2), 141–152 (1996)
4. Belmonte, O., Remolar, I., Ribelles, J., Chover, M., Rebollo, C., Fernandez, M.: Multiresolution triangle strips. In: IASTED International Conference on Visualization, Imaging and Image Processing (VIIP 2001), pp. 182–187 (2001)
5. Bogomjakov, A., Gotsman, C.: Universal rendering sequences for transparent vertex caching of progressive meshes. In: No description on Graphics interface 2001, pp. 81–90. Canadian Information Processing Society (2001)
6. Chow, M.M.: Optimized geometry compression for real-time rendering. In: IEEE Visualization, pp. 347–354 (1997)
7. Cohen-Steiner, D., Alliez, P., Desbrun, M.: Variational shape approximation. SIGGRAPH **23**(3), 905–914 (2004)
8. Cook, W., Rohe, A.: Computing minimum-weight perfect matchings. INFORMS Journal on Computing **11**, 138–148 (1999)
9. Deering, M.: Geometry compression. In: ACM SIGGRAPH, pp. 13–20 (1995)
10. Diaz-Gutierrez, P., Bhushan, A., Gopi, M., Pajarola, R.: Constrained Strip Generation and Management for Efficient Interactive 3D Rendering. In: Proceedings Computer Graphics International Conference, pp. 115–121 (2005)
11. Diaz-Gutierrez, P., Gopi, M.: Quadrilateral and Tetrahedral Mesh Stripification Using 2-Factor Partitioning of the Dual Graph. The Visual Computer (Special Issue for Pacific Graphics) **21**(8-10), 689–697 (2005)
12. Diaz-Gutierrez, P., Gopi, M., Pajarola, R.: Hierarchyless Simplification, Stripification and Compression of Triangulated Two-Manifolds. Computer Graphics Forum **24**(3), 457–467 (2005)
13. Dillencourt, M.: Finding hamiltonian cycles in delaunay triangulations is NP-complete. In: Canadian Conference on Computational Geometry (CCCG), pp. 223–228 (1992)
14. El-Sana, J., Azanli, E., Varshney, A.: Skip strips: maintaining triangle strips for view-dependent rendering. In: IEEE Visualization, pp. 131–138 (1999)
15. Evans, F., Skiena, S.S., Varshney, A.: Completing sequential triangulations is hard. Tech. rep., Dep. of Comp. Sci., SUNYSB (1996)
16. Evans, F., Skiena, S.S., Varshney, A.: Optimizing triangle strips for fast rendering. In: R. Yagel, G.M. Nielson (eds.) IEEE Visualization, pp. 319–326 (1996)
17. Garland, M., Willmott, A., Heckbert, P.S.: Hierarchical face clustering on polygonal surfaces. In: I3D, pp. 49–58 (2001)
18. Gopi, M.: Controllable single-strip generation for triangulated surfaces. In: Pacific Graphics, pp. 61–69. IEEE (2004)
19. Gopi, M., Eppstein, D.: Single-strip triangulation of manifolds with arbitrary topology. Computer Graphics Forum **23**(3), 371–379 (2004)
20. Guha, S., Khuller, S.: Approximation algorithms for connected dominating sets. In: European Symposium on Algorithms, pp. 179–193 (1996)
21. Hoppe, H.: Optimization of mesh locality for transparent vertex caching. In: SIGGRAPH, pp. 269–276 (1999)
22. Kornmann, D.: Fast and simple triangle strip generation. In: Varian Medical Systems Finland, Espoo (1999)
23. Low, K.L., Tan, T.S.: Model simplification using vertex-clustering. In: SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics, pp. 75–ff. New York, NY, USA (1997)
24. Lu, H.I., Ravi, R.: Approximating maximum leaf spanning trees in almost linear time. J. Algorithms **29**(1), 132–141 (1998)
25. Luebke, D., Erikson, C.: View-dependent simplification of arbitrary polygonal environments. In: SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques, pp. 199–208. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1997). DOI http://doi.acm.org/10.1145/258734.258847
26. Luebke, D., Erikson, C.: View-dependent simplification of arbitrary polygonal environments. In: ACM SIGGRAPH, pp. 199–208 (1997)
27. Luebke, D., Reddy, M., Cohen, J.D., Varshney, A., Watson, B., Huebner, R.: Level of Detail for 3D Graphics. Morgan Kaufmann Publishers, San Francisco, California (2003)
28. Mitani, J., Suzuki, H.: Making papercraft toys from meshes using strip-based approximate unfolding. In: ACM SIGGRAPH, pp. 259–263. ACM Press (2004)
29. Neider, J., Davis, T., Woo, M.: OpenGL Programming Guide. Addison Wesley (1993)
30. Pajarola, R., Antonijuan, M., Lario, R.: QuadTIN: Quadtree based triangulated irregular networks. In: IEEE Visualization, pp. 395–402 (2002)
31. Peterson, J.P.C.: Die theorie der regulären graphen. Acta Mathematica **15**, 193–220 (1891)
32. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Communications of the ACM **33**(6), 668–676 (1990)
33. Shafae, M., Pajarola, R.: DStrips: Dynamic triangle strips for real-time mesh simplification and rendering. In: Pacific Graphics, pp. 271–280. IEEE (2003)
34. Stewart, A.J.: Tunneling for triangle strips in continuous level-of-detail meshes. In: Graphics Interface 2001, pp. 91–100 (2001)
35. Vanecek, P., Kolingerová, I.: Multi-path algorithm for triangle strips. In: CGI '04, pp. 2–9 (2004)
36. Velho, L., de Figueiredo, L.H., Gomes, J.: Hierarchical generalized triangle strips. The Visual Computer **15**(1), 21–35 (1999)
37. Xiang, X., Held, M., Mitchell, J.S.B.: Fast and effective stripification of polygonal surface models. In: I3D 1999, pp. 71–78. ACM Press (1999)
38. Yoon, S.E., Lindstrom, P., Pascucci, V., Manocha, D.: Cache-oblivious mesh layouts. ACM Transactions on Graphics **24**(3), 886–893 (2005)

Pablo Diaz-Gutierrez is a Ph.D. student in the Department of Computer Science at the University of California, Irvine. He got his M.S. at the University of California, Irvine in 2005 and his B.S. in Computer Science at the University of Granada, Spain, in 2002. He was a software engineer working in geographic information systems for ESPELSA in Madrid, Spain, and his current research interests include mesh processing, computational geometry and fundamental data structures.

Anusheel Bhushan got his M.S. in Computer Science at the University of California, Irvine in Spring 2005 and his B.S. in Computer Science at the India Institute of Technology at New Delhi, India, in 2003. He worked in geometry processing, image based modeling and MPEG encoding of dynamic synthetic scenes. Currently he works as a software engineer for EBay in San Jose, California.

M. Gopi (Gopi Meenakshisundaram) is an Assistant Professor in the Department of Computer Science at the University of California, Irvine. He got is Ph.D from the University of North Carolina at Chapel Hill in 2001, M.S. from the Indian Institute of Science, Bangalore in 1995, and B.E from Thiagarajar College of Engineering, Madurai, India in 1992. He has worked on various geometric and topological problems in computer graphics. His current research focuses on applying graph algorithms to geometry processing problems in computer graphics.



Renato Pajarola received a Dr. sc. techn. in computer science in 1998 from the Swiss Federal Institute of Technology (ETH) Zürich. After a postdoc at Georgia Tech he joined the University of California Irvine in 1999 as an Assistant Professor. Since 2005 he has been an Associate Professor in computer science at the University of Zürich. His research interests include real-time 3D graphics, scientific visualization and interactive 3D multimedia. He is a frequent committee member and reviewer for top conferences and journals.