In Proceedings Pacific Graphics, pages 271-280, 2003.

DStrips: Dynamic Triangle Strips for Real-Time Mesh Simplification and Rendering

Michael Shafae and Renato Pajarola
Computer Graphics Lab
School of Information & Computer Science
University of California, Irvine
mshafae@ics.uci.edu, pajarola@acm.org

Abstract

DStrips is a simple and efficient method to dynamically manage and generate triangle strips for real-time view-dependent multiresolution meshing and rendering. Progressive view-dependent triangle mesh simplification and rendering is an important concept for interactive visualization environments. To minimize the rendering cost, triangle meshes are simplified to the maximal tolerated perceptual error. A further savings can be gained by using hardware optimized rendering primitives such as triangle strips. However, triangle strips have been rarely used successfully in interactive multiresolution meshes due to the costs involved with maintaining the coherency of the strips in the changing mesh. This paper introduces a new dynamic triangle stripping data structure and algorithm, DStrips, that is practical for use with multiresolution meshes. DStrips is aimed at preserving pre-computed triangle strips through changes in the mesh and generating reasonably good triangle strips in real-time. Furthermore, this data structure and algorithm can be easily adapted to any multiresolution mesh which has a face-to-edge/edge-to-face mapping. The presented approach is implemented on top of a real-time view-dependent meshing and rendering framework based on a half-edge data structure using progressive edge collapses and vertex splits. Direct comparisons are made to previous methods in triangle stripification of dynamic and static meshes.

1. Introduction

Multiresolution modeling techniques are important to cope with the increasingly complex polygonal models available today such as high-resolution isosurfaces, large terrains, and complex digitized shapes [22]. Large triangle meshes are difficult to render at interactive frame rates due to the large number of vertices to be processed by the graphics hardware. Adaptive multiresolution visualization techniques [14] allow rendering the same object using triangle meshes of variable complexity. Thus, the number of processed vertices is adjusted according to some metric relating the object's position and importance in the rendered scene to it's geometry. Many multiresolution triangulation methods [8], [15], [7], [23], [25] have been developed which discrete or hierarchical triangulations for multiresolution visualization. Although reducing the amount of geometry sent to the graphics pipeline elicits a performance gain, a further optimization can be achieved by using optimized rendering primitives, such as triangle strips.

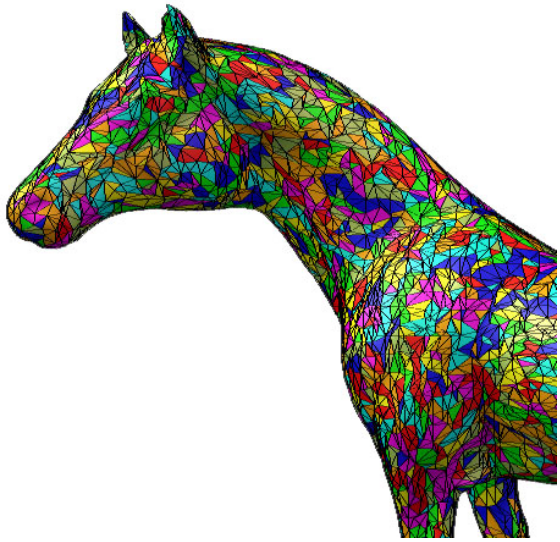


Figure 1. Example of dynamically generated triangle strips of a view-dependently simplified multiresolution mesh. Individual triangle strips are pseudo-colored for better distinction (15548 triangles represented by 3432 triangle strips).

Triangle strips have been used extensively for static mesh representations due to the widespread availability of tools such as the classic *tomesh.c* program [1], Stripe [12] and the more recent NVidia NVTriStrip tools [4] [3] and the effortless gain in rendering performance. Each of these tools exemplifies the driving ethos of using triangle strips; gener-

ate and encode triangle strips (*stripification*) in a near optimal fashion regardless of computational cost. However, using such stripification techniques are not practical for view-dependent multiresolution triangle meshes.

A view-dependent multiresolution visualization technique is geared almost uniquely at interactive applications. Where as the aforementioned stripification techniques are ill suited for interactive applications. The disjuncture lies in the fact that the previous stripification techniques endeavour to produce a near optimal stripification. The observation made is that the performance gained from a very-good stripification can be more desirable in a multiresolution paradigm than a near optimal stripification.

In this paper, *DStrips* is presented. DStrips is a simple, effective triangle strip management algorithm and data strucutre for real-time, interactive rendering of multiresolution triangle meshes. This approach imposes a pipeline-like structure on any mesh update event which grows, shrinks, or partially re-strips a triangle strip in a multiresolution mesh. DStrips is not another triangle strip generation technique. Conceptually, DStrips is an algorithm which enables dynamic, unstructured, multi-resolution meshes gain the rendering speed-up from using optimized rendering primitives. Furthermore, an evaluation is presented demonstrating the benefits of less than optimal stripifications.

The implementation presented in this paper is built around a multiresolution mesh using progressive edge collapse and vertex split operations [16] based on a half-edge representation of the triangle mesh connectivity [28]. DStrips is easily adapted to any multiresolution mesh so long as the mesh provides a mapping from an edge to its associated faces and vice-versa. Figure 1 presents an example screenshot of pseudo-colored triangle strips of a view-dependently simplified multiresolution mesh that was generated by DStrips.

The main contributions of our approach are a pipelineline algorithm which adapts the encoded triangle strips to the changes in the underlying multiresolution mesh via triangle strip savvy edge collapse and vertex split operations, a simple, space efficient triangle strip data structure based on maintaining the orientation of a face in relationship to the previous face in the strip, and an efficient partial triangle strip destruction and restripping algorithm.

2. Related Work

2.1. Mesh Simplification

Numerous methods for mesh simplification have been developed in the last decade, and a discussion is beyond the scope of this paper. For an overview on the various mesh simplification methods see [15], [7], or [25]. In this context we only want to highlight the work on progressive simplification of meshes [16]. A sequence of n *edge collapse* (ecol) operations is applied to simplify an arbitrary mesh M^n to a much simpler mesh M^0 of the same topology, reducing the number of vertices by n. Given the coarse mesh M^0, i different levels of detail approximations, M^i can be reconstructed by applying i *vertex split* (vsplit) operations — the inverse of the ecol operation — to the base mesh M^0. The multiresolution mesh underlying the presented description of DStrips is based on progressive edge collapses and vertex splits using the half-edge collapse variant.

2.2. Triangle Stripification

The basic problem of creating an optimal set of triangle strips for a given triangulation is NP-complete [10], [13], and most previous work [1], [12], [2], [30], [21], [27] has focused on heuristics to minimize the number of triangle strips for static triangle meshes. In particular, provably good triangle stripification has been presented in [30] which performs extremely well also in terms of performance. Further improvements in generating quality triangle strips were reported more recently [27]. Nevertheless, it is clear from performance measurements [27] that these "static" triangle stripping algorithms such as *tomesh.c* [1], Stripe [12], FTSG [30], NVTriStrip tools [4] [3], and Tunneling [27] are not suitable for computing triangle strips in real-time for "static" triangle meshes, let alone for large, view-dependent, multiresolution triangle meshes.

A topic which is not addressed in the previous works is what constitutes *long enough* or *too short* when triangle strips are discussed. It is globally accepted that the longer the strip length, the better the strips. However, from an interactive application's stand point, this is not the case.

For such real-time, continuously adaptive multiresolution meshes, it is much more important to be able to rapidly compute a reasonably good stripification than to create an optimized stripification for one particular level-of-detail (LOD) mesh instance. In view-dependent meshing approaches [29], [17], [24], [11], [26], [20] a LOD-mesh instance may only be rendered once for a frame. On the other hand, significant regions of the triangle mesh observe little to no changes in mesh connectivity between frames. Therefore, it is critical that the stripification is preserved between frames as much as possible with few incremental updates performed for each frame.

In Skip Strips [11], a triangle strip maintenance approach is presented that keeps track of incremental changes to a given initial triangle stripification. However, since these skip strips are only computed once at initialization, at runtime, the actually displayed strips are merely shortened versions of the original strips. Thus, the fragmentation of the triangle strips can get bad for coarse meshes. Furthermore, Skip Strips passes the display triangle strips through filters in order to remove excessive line triangles and other unwanted noise. Skip Strips requires a significant storage cost overhead to represent the skip-strip lists when compared to DStrips.

Similarly, the multiresolution triangle strips presented in [5] also capture a static initial stripification of the full reso-

lution triangle mesh which is subsequently updated (shortened and separated strips) to reflect mesh simplification operations. This method also has a monotonically degrading triangle strip fragmentation with mesh resolution as triangle strips are never merged or lengthen during a simplification operation. As shown in [5], the number of triangle strips rapidly approaches the number of displayed triangles.

The tunneling method [27] is not only used to optimize a "static" stripification, but is targeted to maintain a good stripification of a triangle mesh that changes over time. While the method does generate extremely good sets of triangle strips, its performance is an order of magnitude too slow for real-time view-dependent meshing and rendering. As reported in [27], the time required for tunneling of 5000 edge collapse/vertex split operations is more than 50 seconds (750MHz Pentium4). This performance is in dramatic contrast to real-time view-dependent meshing and rendering where 4000 mesh update operations are performed in less than 20 milliseconds as reported in [26] (450MHz UltraSparc II).

Table 1 summarizes the differences and similarities of the previous stripification techniques. A distinction can be drawn along two axes, how is the stripification managed and when is the stripification computed. DStrips is the only real-time approach which dynamically manages and computes the mesh stripification.

Some other noteworthy work related to triangle strip generation and rendering is the work on generalized triangle meshes [9] and optimized vertex caching [18], [6], as well as the triangle strip compression method presented in [19].

Name	Algorithm	Stripification	Strip Management
Dstrips	Online	Dynamic	Shorten, Grow, Merge, Partial Re-Strip
Tunneling	Online	Dynamic	Repair & Merge (Tunneling Operation)
Stripe	Offline	Static	Not Applicable
Skip Strips (Stripe)	Pre-Process	Static	Resize Pre-Computed Strips
Mulltiresolution $\triangle$ Strips	Pre-Process	Static	Resize Pre-Computed Strips

Table 1. A comparison of triangle stripification techniques. Note that a clear distinction can be drawn between the techniques which dynamically manage the triangle strips and those which shorten pre-computed triangle strips.

3. Multiresolution Model

DStrips was implemented on top of a multiresolution triangle mesh framework similar to [17], [20], [26], and [29]. The dynamic stripification algorithm presented in this paper is applicable to a wide range of adaptive meshing approaches. DStrips only requires that the mesh provide a mapping to and from a face to its associated edges and maintain a consistent orientation of the edges of a face. A brief introduction to the LOD-mesh framework is necessary in order to describe the fundamental interface between DStrips and the underlying LOD-mesh.

The mesh uses the half-edge data structure [28] to capture mesh connectivity. Each triangle face is implicitly represented by an ordered set of three oriented half-edges as shown in Figure 2. This ordering defines a simple mapping to and from an arbitrary face to its associated edges. Every half-edge h knows its reverse twin half-edge ($h.r$), the next ($h.n$) and previous ($h.p$) half-edges of that triangle, and the starting vertex ($h.v$) of the half-edge. A triangle mesh with n vertices and m faces consists of an array of n vertex coordinates and an array of $3m$ half-edge elements. This mesh representation allows efficient traversal and neighbor finding on the triangulated surface by simple indexing operations.

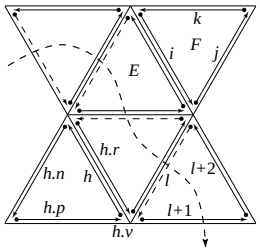


Figure 2. Each face is composed of three directed half-edges. The half-edge reverse as well as the previous and next information allows to find and "walk" to adjacent triangles. For the triangle strip (dashed curve), the half-edges that are used to orient oneself on a face – the entry edges – are drawn as dashed lines.

The adaptive meshing process used in this implementation is based on progressive edge collapse and vertex split operations [16]. Collectively, in this paper, operations which affect the connectivity or resolution of the mesh will be referred to as *mesh operations*. A sequence of mesh operations defines a binary multiresolution triangle mesh hierarchy [17], [20], [26], [29]. A particular view-dependent LOD-mesh is defined by a front through that hierarchy which separates the expanded and collapsed nodes of the multiresolution model as shown in Figure 3. At runtime, for every change in viewing parameters, this front of *active nodes* is traversed and updated dynamically according to view-dependent error metrics and mesh simplification heuristics. The reader is referred to the literature for more details [17], [20], [26], [29].

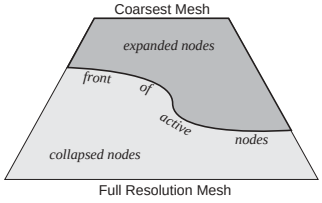


Figure 3. View-dependent multiresolution hierarchy. List of active nodes separates the collapsed edges from the expanded nodes.

Whereas the mesh data structure is concerned with the connectivity of the mesh, DStrips is concerned with how the connectivity, and changes to it, effect the mesh's stripification. A mapping must exist between the mesh edges and the edge's faces in order for DStrips and the multiresolution mesh to coordinate work on their data structures. The mapping can be procedural, or in this particular implementation, implicit.

To illustrate, in Figure 2, the mapping indicates for any given face F its edges i, j and k in a given order (i.e. counter-clockwise), and inversely for every edge i the mapping must provide the two adjacent faces F and E (or just one face if the edge is a boundary). Additionally, the order (i.e. counter-clockwise) of the edges i, j and k of a face F must stay consistent before and after any mesh operations. That is, the mapping for a face's *zeroth* edge should always be the same, with respect to the other edges of that face.

DStrips specifies an *entry-edge* of a face across which it is reached from the previous face in a triangle strip. The entry-edge is encoded as an offset (0, 1 or 2) from the zeroth edge in the given ordering. Given a consistent edge ordering per face and the entry-edge l, the triangle strip traversal direction *left/right* can also be mapped to an offset $l + 2/l + 1$ (modulo 3 addition and counter-clockwise orientation assumed) as shown in Figure 2. The adjacency information of a face is needed to traverse an edge i from a face F to find the edge-adjacent triangle face E.

DStrip's underlying view-dependent adaptive mesh consistently changes the connectivity in such a fashion as outlined above. The half-edge data structure used supports all adjacency relations required by DStrips. DStrips can be adapted to any similar multiresolution mesh data structure which can provide an edge to face mapping and mesh adjacency information.

4. Dynamic Stripification

DStrips is not another strip generation technique. DStrips is a data stucture and algorithm that dynamically manages the triangle strips of a view dependent multiresolution mesh. Unlike Stripe, DStrips does not solely generate triangle strips. DStrips does not solely shorten triangle strips to accommodate changes in the mesh's connectivity like Skip Strips. Furthermore, unlike the previous work, DStrips generates very-good triangle strips which are not meant to be optimal. This is a departure from the generally accepted ideal that longer strips are better strips and will be explained in section 4.3.

What DStrips does is provide a space and time efficient means to shrink, grow, merge triangle strips in a view-dependent, multiresolution mesh. The stripification is never recomputed from scratch. By observation, one can notice that changes in an multiresolution mesh affect only small portions of the mesh per frame. In a few situations, DStrips will remove a triangle strip and schedule the constituent faces to be re-stripped prior to rendering. This allows the stripification to adapt to any radical changes in the mesh's connectivity due to changes in the view position.

Figure 4 illustrates the DStrips framework. Each frame the active nodes of the mesh are traversed and edge collapse or vertex split operations are initiated. These mesh operations have to trigger DStrips actions for newly inserted faces and for faces that are removed from the mesh. DStrips itself modifies existing triangle strips, and in few cases removes strips; re-stripping the partially un-stripped triangle mesh prior to rendering. Finally, the mesh is rendered as a set of triangle strips.

Figure 4. DStrips LOD-mesh update pipeline.

4.1. Dynamic Updates to the Stripification

With changes to the view position comes changes to the connectivity of the multiresolution mesh. The triangle strips will need to be updated to keep coherent with the mesh. This is accomplished in one of two ways. In the first case, the changes to the mesh can be accommodated by *strip savvy* mesh operations – by growing or shrinking the triangle strips. In the second case, strips may need to be un-stripped. If a strip is un-stripped, after all mesh operations have completed, the un-stripped portions of the mesh are re-stripped.

A strip savvy operation will grow or shrink a strip if one of the following conditions is met: 1) a start or stop face is adjacent to the face being added or removed, or 2) a start or stop face is adjacent to the face being added or removed.

Unlike Skip Strips [11], DStrips's strips are not shortened revisions of the high resolution originals. This, in itself, distinguishes DStrips from the previous state of the art. Rather, the stripification is dynamically updated solely upon the local changes to the mesh. Furthermore, an optimization

step is done as well to merge triangle strips together.

4.1.1 Strip-Savvy Mesh Updates - Grow & Shrink

Inserting a split face into the middle of a strip as shown in Figure 5 a) is a simple operation. The flags of the face in front and behind of the new face are updated to have the strip pass through the new face. Is the inserted face adjacent to the start or end of a triangle strip as shown in Figure 5 b), the corresponding triangle strip is extended. If the face is added at the start of a strip, the flags of the old start face and the new start face must be updated accordingly as well as the strip header pointing to that strip.

If a face is inserted between triangle strips as shown in Figure 5 c), the adjacent triangle strips are (optionally) destroyed and the new inserted faces are added to the work list to be re-stripped before rendering. Since two inserted faces could match different configurations, for example as shown in Figures 5 d) or e), care has to be taken not to break the coherency of the stripification when both faces effect the same strip as in Figure 5 f). In that case, the destruction of a strip overrides any extension of that same strip.

Even for vertices of high degree, the configurations in Figure 5 are valid. When a vertex split event occurs, the "new" faces are added into the mesh first. DStrips, using the array of strip leaders (see Section 4.4), is able to quickly identify if the faces adjacent to either "new" face are in the same strip or start/stop faces.

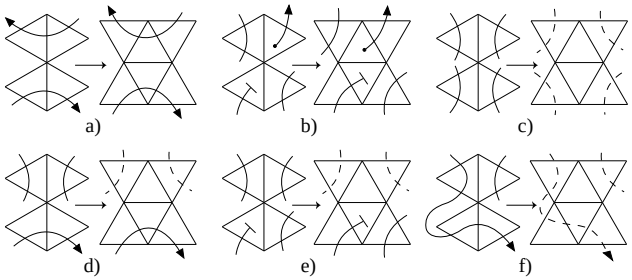


Figure 5. Vertex split configurations. Dashed lines denote triangle strips that are (optionally) destroyed and have to be re-stripped.

The edge collapse operation is less complicated. Only the configuration shown in Figure 6 results in the destruction of the affected triangle strip and scheduling that strip for re-stripping. All other edge collapse configurations are analogous to the situations in Figures 5 a) and b) and only require shortening of strips, or involve collapsing singleton strips (i.e. possible analogous to Figures 5 d), e) or f)) which can simply be destroyed.

4.1.2 Partial Re-Stripping

As mentioned previously, there are situations where growing or shrinking of a triangle strip is not possible and an existing triangle strip must be removed from the stripification.

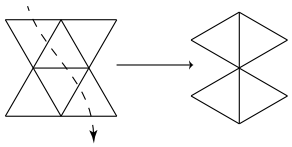


Figure 6. Fail situation of an edge collapse. Triangle strip crossing the edge collapse is destroyed and has to be re-stripped.

This can easily be achieved by "walking" a triangle strips entire length reseting each face's flag as well as its entry in the strip leader array. The start and stop faces of the triangle strip are added to the work list. Finally, its corresponding entry in the list of strip headers is removed. This can efficiently be done by accessing the strip header through the strip leader array.

A change to one strip can affect the adjacent triangle strips. If a face is introduced by a vertex split in a situation that does not satisfy the above criteria then the triangle strips of the faces adjacent to the cut-edges are removed as well. Furthermore, if an edge collapse removes two consecutive faces of a triangle strip it will be destroyed, see also Figure 6.

Figure 7 illustrates the basic triangle re-stripping method which is based on a work-list algorithm. The work list contains candidate faces to start new triangle strips. Note that the work list is sparse, it does not contain all un-stripped faces of the mesh, but at any time only holds at least one face per connected un-stripped mesh region. This is sufficient to guarantee that all currently un-stripped faces will be processed and that no un-stripped mesh part is accidentally ignored. The stripification algorithm proceeds by initializing a triangle strip from an un-stripped face taken from the work list and then iteratively selects an adjacent un-stripped face based on a heuristic. In Section 5, different heuristics are presented following this core algorithm. Note that whenever the stripification process has a choice of continuing to the left or to the right of the current face, because both neighbors are un-stripped, then the face that is not selected is added to the work list to preserve its invariant of holding at least one face per connected un-stripped mesh region.

4.2. Local Strip Optimization - Strip Merge

After each mesh operation, DStrips inspects the patch of triangles immediately surrounding the edge being operated on and attempts to make local optimizations to the stripification. The optimization is to merge strips together if possible and it requires the strip leaders array described in Section 4.4 (see also Figure 9). Note that this optimization is performed immediately after a mesh update and strip modifications as shown in Figure 4, and that it only considers valid triangle strips surrounding the mesh update but not any un-stripped faces. This optimization step serves as an inexpensive way to consolidate strips opportunistically.

Merging triangle strips is possible if an end face F_{end} of a triangle strip is adjacent to the start face F_{start} of another

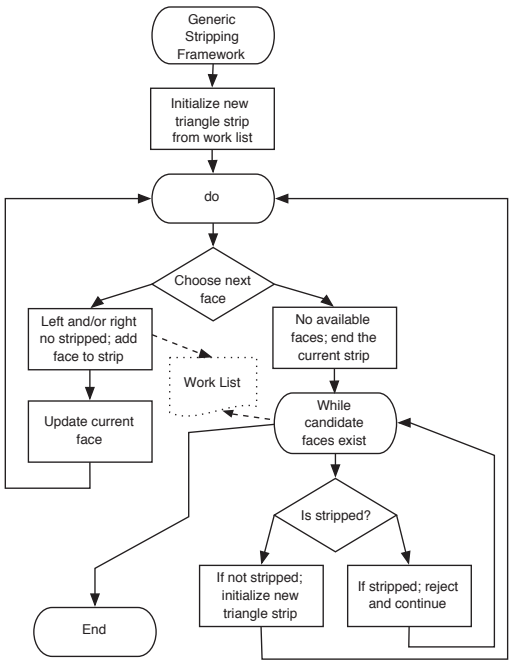


Figure 7. General re-stripping algorithm.

(the *merged strip*), for example as in Figure 5 b) after the vertex split has been performed. The flags of the faces F_{end} and F_{start} are updated to reflect the new, merged triangle strip configuration. Furthermore, the strip header (accessible through the strip leader information of F_{start}) of the merged strip must be removed, and the strip leaders of all faces of the merged strip are updated to point to the same strip header as F_{end}.

4.3. Strip Length

If given a triangle mesh of n triangles, at worse it would cost $3n$ vertices to render the entire mesh. At best, it would cost n vertices. However, reaching the optimal solution would result in having optimal triangle strips, which is NP-complete.

DStrips does not attempt or claim to compute near optimal triangle strips. Rather, it computes *very-good* triangle strips. Very-good can be qualified as costing $\frac{5n}{3}$ vertices or less to render a triangle mesh. If k is the average length of a strip over the entire mesh, then the number of vertices submitted to the graphics hardware will be $\frac{2+k}{k}$ per triangle, not allowing any swap operations. If this is to be less than $\frac{5}{3}$, then $k \geq 3$. This suggests that, so long as there are no swaps in a strip, the strip is long enough to give a $\frac{2}{3}$ savings already.

If swap operations are factored in then let p be the percent of swap operations in a strip over the entire mesh. Then $\frac{2+k+p \cdot k}{k} \leq \frac{5}{3}$. This leaves k as a function of p, $k \geq \frac{2}{\frac{2}{3}-p}$, suggesting that $p < \frac{2}{3}$.

Even longer strips are desirable but not necessary. From the experimental data show in Table 3 in Section 6, one can see that the strips do not degrade over time nor do the swaps adversely effect the strips on average. Furthermore, several trials were done using a non-view dependent multiresolution mesh to corroborate this paper's position on triangle strip quality. Figure 8 shows one of these trials using one of the stripping algorithms presented in this paper. From the data collected, k_{min} was computed and plotted along with the average strip length and the average number of swaps per triangle strip.

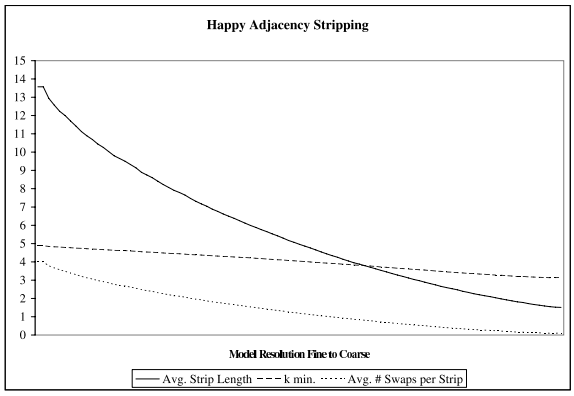


Figure 8. Using a multiresolution mesh with fixed LODs, data was collected regarding average strip length, average number of swaps and percent swaps per strip.

4.4. Triangle Strip Data Structure

The basic data structures as illustrated in Figure 9 that comprise DStrips are fairly simple. DStrips stores information per face that indicates the state of each face with respect to the stripification process. At any time after LOD-mesh updates and corresponding DStrips actions (see also Figure 4), all visible faces are part of active triangle strips. In addition to the per-face information, DStrips maintains a list of active triangle strip headers pointing to the first face of each strip. An array of strip leaders is kept for each face which is used to test for relationships between faces. Optional per triangle strip information can be stored in the strip headers. An example of this are vertex arrays which are discussed in Section 4.5.

The triangle strips are maintained by encoding the following information about a strip into the faces that compose it: 1) is the face *stripped?* (1 bit), 2) is the face the *start* face of a triangle strip (1 bit), 3) the *entry* edge of the strip (2 bits), and 4) the *direction* of the triangle strip — left, right, stop (2 bits).

These four fields together use less than one byte per face. The *stripped?* flag is used to determine triangles that have to be re-stripped. While traversing a strip, the *direction* flag defines how to reach the next face relative to the entry-edge or flags the end of a strip. The *entry* flag identifies which edge of the face (first, second, or third) is the strip's entry

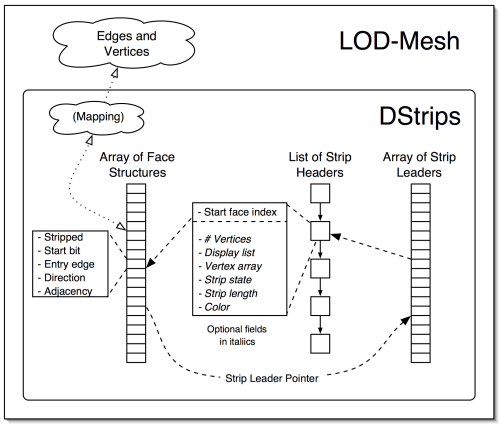


Figure 9. DStrips data structures.

point. The *direction* and *entry* flags allow a walk of the strip forwards towards the stop face and backwards towards the start face from any given face.

As mentioned above, each strip is addressed from a singly linked list of triangle strip headers. These headers store an index into the array of triangle faces, but can also store per strip properties as well. Fundamentally, the index to the first triangle face is all that is needed. For improved rendering one may store an OpenGL display list or a vertex array for each strip, and other information such as the *strip state* which indicates if the display list or vertex array have to be updated. Whenever a strip is deleted its corresponding strip header becomes invalid and must be removed from this list.

The array of strip leaders is an auxiliary data structure that is used to identify relationships between faces. A simple test can be done to see if a face is in the same strip as another face. This simplifies removal of invalid strip headers, for strip savvy mesh operations and the local optimization heuristic.

An additional data structure used by DStrips is the *work list* which is used when a partial re-stripping of the LOD-mesh takes place, see also Figures 4 and 7. In the event that portions of the LOD-mesh become un-stripped, DStrips places candidate faces to start new triangle strips into this work list. In fact, each connected component of un-stripped mesh must be represented by at least one candidate face in the work list. Once the work list is empty, DStrips has re-stripped the entire mesh. Two different work list data structures were experimented with. The first is a first-in-first-out (FIFO) queue and the second is a more complicated multi-headed queue. Since each of these data structures is closely tied to a particular stripping method, they are discussed in more detail in Section 5.

4.5. Rendering

After performing all mesh operations and triangle strip updates, the basic display method renders a triangle strip for each strip header. This is done by initializing the triangle strip from the start face of a given strip header and then "walking" the consecutive faces according to their flags. The triangle strip is completed and sent to the graphics hardware when the stop face is reached. Using OpenGL, this is known as immediate mode. Note that the *direction* flag allows generalized triangle strips that do not require strictly alternating left-right traversals, see also Section 5.

Vertex arrays optimize the transfer of (static) geometry information from main memory to the graphics hardware. For example, a triangle strip can effectively be represented by a list of vertices which avoids "walking" the individual faces of this strip every time it has to be rendered as mentioned above. In DStrips, an optional vertex array can be stored with each strip header as shown in Figure 9.

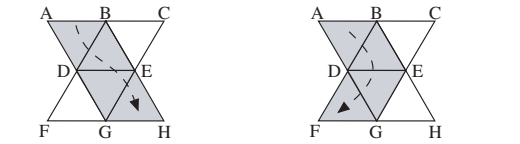
Along with managing the triangle strips, DStrips can update the per strip vertex arrays as needed. Since triangle strips can be modified (growing, shrinking, merging), additional information must be maintained with each strip if display or vertex lists are used. The strip state denotes if the vertex array is clean, or dirty because the strip was modified. In case a strip is marked as dirty its vertex array must be re-indexed before it can be rendered. The number of vertices and strip length (including swap operations) is also stored for correct memory allocation of vertex arrays.

For static meshes, representing the entire mesh as a single OpenGL vertex array or compiled vertex array is straight forward. When a multiresolution mesh is used, enabling a single vertex array for the entire mesh looses any value due to the per-frame re-indexing and re-allocation of memory for the single vertex array. Since the mesh is already partitioned into a set of triangle strips and because relatively few triangle strips will be altered between frames, building per strip vertex arrays provides a significant performance increase at the cost of pre-fetching the strip vertex indices.

5. Triangle Stripping Algorithms

A basic triangle strip consist of an edge-adjacent sequence of triangles visited in a strictly alternating left-right traversal order as shown in Figure 10 a). In a generalized triangle strip the strict left-right sequence is relaxed to allow multiple consecutive left or right traversals. This requires so called *swap* operations as shown in Figure 10 b) which can be simulated by vertex repetition if not supported by the graphics API; which is the case with OpenGL. With generalized triangle strips, swap operations that are simulated by repeated vertices cause degenerate line triangles (however, these can effectively be recognized and ignored by the graphics hardware in some cases).

DStrips has adapted two heuristics discussed from [12] and [1] into a simple and efficient *work list* algorithm which can be used for partial or complete stripping of a mesh. The algorithms have been kept simple since the crux of DStrips novelty is the ability to dynamically grow, shrink and merge



a) b)

Figure 10. a) Basic, strictly alternating triangle strip. b) Generalized triangle strip with swap operation simulated by repeated vertices.

the triangle strips in an adaptive, view-dependent multiresolution mesh. Note that DStrips does not place any particular requirements on the behavior of the stripping procedures used and can be tuned to meet particular needs or hardware features. In this implementation, no tuning was done to take advantage of any available OpenGL extensions, such as triangle lists or vertex caching.

The algorithms work iteratively, adding consecutive faces to a strip using only *local information*. Local information is defined to be the faces immediately adjacent to the face being processed. The first heuristic is the *greedy* heuristic and the second is the *adjacency* heuristic. The greedy heuristic's goal is to maximize the length of the strip. while the adjacency heuristic's goal is to lead the growth of a strip into sparsely stripped areas of the mesh; avoiding singleton strips. In Figure 7, the general stripping algorithm is described.

5.1. Greedy Stripping Algorithm

The greedy stripping heuristic uses a simple FIFO queue as its work list data structure. The algorithm starts on the first un-stripped face taken from the FIFO queue and selects an initial traversal direction from that face (to another un-stripped face). At each new face, the traversal direction to the next un-stripped face is chosen in such a way to limit the number of swap operations and alternate between left and right traversal directions, if possible. During the stripping process, any un-stripped face that was not chosen as the next face is added to the FIFO work list. This is necessary to preserve the work list's invariance of holding at least one face per un-stripped connected mesh region. When no more un-stripped adjacent faces are available, the current triangle strip is completed and a new one is started should the work list be non-empty.

This process continues until the work list is exhausted. Sometimes a face may be added to the work list but may not require processing. This is because the face has been stripped while being in the work list and is discarded from the work list when it is encountered.

Greedy stripping computes very long strips but may introduce singleton strips. Furthermore, extremely long strips may not necessarily or desired per the afore discussed points raised in section 4.3. Enforcing a no-swap heuristic reduces the chances of singleton strips. To its credit, a greedy approach is very fast and results in good strips, regardless of the swaps in the strips.

5.2. Adjacency Stripping Algorithm

The adjacency heuristic algorithm works much the same as the greedy heuristic algorithm, however, in lieu of selecting the traversal direction based on the left-right alternation order, it selects the direction based on *adjacency information*. The adjacency information for each face is how many of its edge-neighbors are un-stripped. The adjacency stripping algorithm choses at each step the face with the lowest adjacency number to be added to the current triangle strip. Using this adjacency information constitutes a simple one-face look-ahead strategy.

The work list used in this algorithm is not a simple stack. Due to the particular goal of this algorithm, choosing good candidate as the start face of a triangle strip from the work list is important. The lower the adjacency number the better a candidate is. Therefore, the work list in this case is actually a set of four queues, one for each possible adjacency value 0 to 3. A face added to the work list is put into the queue of its current adjacency value. This multi-headed work list enables the adjacency stripping algorithm always to start on a face with lowest possible adjacency value by checking the queues for availability in corresponding order.

During stripification, adjacency values of faces change continually, especially for candidate faces in the work list's queues. These are not updated immediately but remain in the queue they currently are in. This constitutes a sub-optimality of choosing start faces but is more geared towards processing speed which is of utmost importance to real-time stripification.

The adjacency stripping algorithm prevents single strips in most cases. The main situation where the adjacency stripping algorithm could possibly create a singleton strip is when a face's adjacency could drop from 1 to 0, and subsequently generate a singleton strip, because the triangle strip under construction choses another face to proceed. Since the face under consideration must have a current adjacency value of 1 and since the adjacency stripping algorithm proceeds always to the face with lowest adjacency, the other selected face also must have had an adjacency of 1. Due to the one-face look-ahead strategy the stripping algorithm can detect this situation and prevent it by connecting the two 1-adjacency faces to form a strip of length 3.

6. Experiments

All experiments of DStrips reported here were performed on a Sun Microsystems Ultra 60 workstation with dual 450MHz UltraSparc II CPU and Expert3D graphics card. Table 2 shows the sizes of the different models we used for testing DStrips in the second and third columns.

In Table 2's fourth through sixth column shows the average number of faces, LOD-updates and triangle strips encountered each frame. The time to perform the edge collapse and vertex split updates each frame is also recorded here since it is independent of the rendering mode. In the last column, the average number of triangle strips per frame is given for the three stripping configurations: adjacency stripping, greedy stripping allowing swap operations and greedy stripping without swap operations (strictly left-right). One can see from Table 2 that adjacency stripping generates fewer strips than greedy stripping, in particular if strict left-right alternation is enforced.

Model	# Faces	# Vertices	# △ Drawn	# Updates	Update Time	# Strips ADJ	GS	GNS
happy	100,000	49,794	54,784	358	3ms	7,006	8,127	12,143
horse	96,966	48,485	39,584	519	4ms	5,008	5,428	7,808
phone	165,963	83,044	60,291	498	5ms	7,272	7,904	11,382

Table 2. The model's name, total number of triangle faces, total number of vertices, per frame average numbers of rendered triangles, LOD-mesh updates, and time to perform mesh updates. The average number of triangle strips is divided into adjacency stripping (ADJ) as well as greedy stripping with swap (GS) and without swap operations (GNS).

Table 3 presents an example of the average strip length for one of the models. The horse model was used as an example. The view position was animated and the average strip length was collected per frame. From the collected data, one can conclude that DStrips does not exhibit the same strip degradation problem that other Skip Strips [12] and Multiresolution Trianglee Strips [5] exhibit. A discussion of DStrips motivation to strive for very–good triangle strips is in Section 4.3.

Strip Heuristic	# Frames	Avg. Length	Avg. # Swaps
Left/Right Greedy	622	7.38	2.03
Left/Right Greedy – No Swaps	580	5.04	N.A.
Adjacency	656	8.50	2.53

Table 3. All results in this table are from the model horse.ply. The results were captured by animating the view position and collecting data per frame. The length of a strip is considered to be all non-degenerate triangles. See the discussion in Section 4.3 for further detail.

Table 4 presents rendering performance tests of DStrips. Note that DStrip's overall display time is the sum of stripping and rendering (immediate or vertex array mode). It is clear from Table 4 that DStrips can maintain a good stripification without any overhead introduced compared to standard rendering. In fact, in the basic immediate mode rendering DStrips is able to dynamically maintain triangle strips and render them in less time than a standard indexed triangle mesh rendering requires for the same LOD-mesh. The improvements are in the range of 5% to 20%. If vertex arrays are used to represent the triangle strips, further significant rendering improvements can be seen. Note that DStrips efficiently allows bookkeeping of vertex arrays which cannot easily be done in a standard LOD-mesh framework.

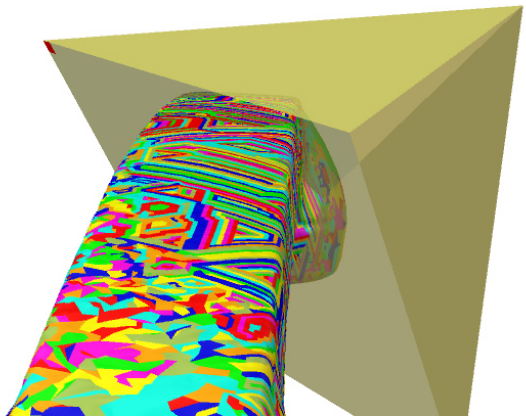


Figure 11: Phone model simplified for the given view-frustum (transparent yellow pyramid).

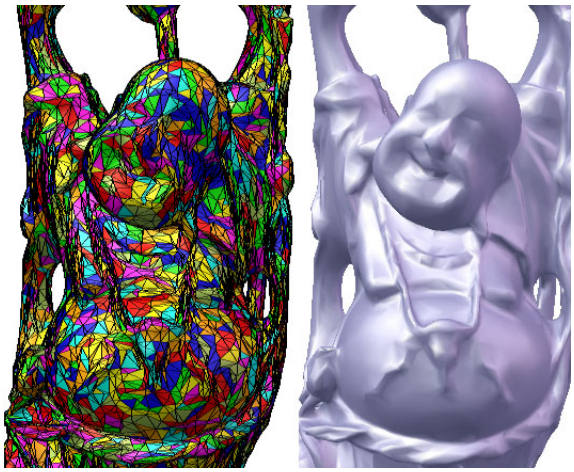


Figure 12: The happy model simplified to 34936 triangles represented by 6945 triangle strips.

Figures 11 and 12 show some dynamic triangle stripping examples as achieved by DStrip. Figure 11 shows a given view-frustum as transparent yellow pyramid and the phone model view-dependently simplified according to that view-frustum. Figure 12 shows the same view of the happy model once with pseudo colored triangle strips and once using smooth shading.

7. Conclusion

This paper has presented DStrips, a simple and efficient method to dynamically generate triangle strips for real-time level-of-detail (LOD) meshing and rendering. Built on top of a widely used LOD-mesh framework using a half-edge data structure based hierarchical multiresolution triangulation framework, DStrips has shown efficient data structures and algorithms to compute a mesh stripification and to manage it dynamically through strip grow and shrink op-

Model	Indexed	Stripping			Immediate			Vertex Arrays		
		ADJ	GS	GNS	ADJ	GS	GNS	ADJ	GS	GNS
happy	97	11	10	9	68	68	77	41	41	48
horse	64	13	12	12	44	45	50	29	28	34
phone	104	21	20	19	79	77	83	44	45	54

Table 4. Rendering and stripping times given in milliseconds and averaged per frame, with and without DStrips. Compared to standard rendering (Indexed), DStrips overall display time is the sum of stripping and rendering. For DStrips, plain immediate mode rendering time is reported as well as using vertex arrays.

erations, strip savvy mesh updates, and partial re-stripping of the LOD-mesh. Experimental results were presented to support these claims. Compared to other methods, DStrips uses a simpler and more compact data structure, is easily extended to incorporate application specific details, and can be adapted to other LOD-mesh frameworks.

8. Acknowledgments

The authors would like to thank the Stanford Computer Graphics Lab, the Georgia Tech Large Geometric Models Archive and Cyberware for providing freely-available high-resolution geometric models. As well as a thank you to M.D. for reading the initial draft.

References

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

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

[3] C. Beeson and J. Demer. Nvtristrip v1.1. Software available via Internet web site., November 2000. http://developer.nvidia.com/view.asp?IO=nvtristrip_v1_1.

[4] C. Beeson and J. Demer. Nvtristrip, library version. Software available via Internet web site., January 2002. http://developer.nvidia.com/view.asp?IO=nvtristrip_library.

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

[6] A. Bogomjakov and C. Gotsman. Universal rendering sequences for transparent vertex caching of progressive meshes. *Computer Graphics Forum*, 21(2):137–148, 2002.

[7] P. Cignoni, C. Montani, and R. Scopigno. A comparison of mesh simplification algorithms. *Computers & Graphics*, 22(1):37–54, 1998.

[8] L. De Floriani and E. Puppo. Hierarchical triangulation for multiresolution surface description. *ACM Transactions on Graphics*, 14(4):363–411, 1995.

[9] M. Deering. Geometry compression. In *Proceedings SIGGRAPH 95*, pages 13–20. ACM SIGGRAPH, 1995.

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

[11] J. El-Sana, E. Azanli, and A. Varshney. Skip strips: Maintaining triangle strips for view-dependent rendering. In *Proceedings IEEE Visualization 99*, pages 131–138. Computer Society Press, 1999.

[12] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In *Proceedings IEEE Visualization 96*, pages 319–326. Computer Society Press, 1996.

[13] F. Evans, S. S. Skiena, and A. Varshney. Completing sequential triangulations is hard. Technical report, Department of Computer Science, State University of New York at Stony Brook, 1996.

[14] T. Funkhouser and C. Sequin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings SIGGRAPH 93*, pages 247–254. ACM SIGGRAPH, 1993.

[15] P. S. Heckbert and M. Garland. Survey of polygonal surface simplification algorithms. SIGGRAPH 97 Course Notes 25, 1997.

[16] H. Hoppe. Progressive meshes. In *Proceedings SIGGRAPH 96*, pages 99–108. ACM SIGGRAPH, 1996.

[17] H. Hoppe. View-dependent refinement of progressive meshes. In *Proceedings SIGGRAPH 97*, pages 189–198. ACM SIGGRAPH, 1997.

[18] H. Hoppe. Optimization of mesh locality for transparent vertex caching. In *Proceedings SIGGRAPH 99*, pages 269–276. ACM SIGGRAPH, 1999.

[19] M. Isenburg. Triangle strip compression. In *Proceedings Graphics Interface 2000*, pages 197–204, 2000.

[20] J. Kim and S. Lee. Truly selective refinement of progressive meshes. In *Proceedings Graphics Interface 2001*, pages 101–110, 2001.

[21] D. Kornmann. Fast and simple triangle strip generation. Technical Report, 1999.

[22] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The digital michelangelo project: 3d scanning of large statues. In *Proceedings SIGGRAPH 2000*, pages 131–144. ACM SIGGRAPH, 2000.

[23] P. Lindstrom and G. Turk. Evaluation of memoryless simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):98–115, April-June 1999.

[24] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings SIGGRAPH 97*, pages 199–208. ACM SIGGRAPH, 1997.

[25] D. P. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics & Applications*, 21(3):24–35, May/June 2001.

[26] R. Pajarola. Fastmesh: Efficient view-dependent meshing. In *Proceedings Pacific Graphics 2001*, pages 22–30. IEEE, Computer Society Press, 2001.

[27] A. J. Stewart. Tunneling for triangle strips in continuous level-of-detail meshes. In *Proceedings Graphics Interface 01*, pages 91–100, 2001.

[28] K. Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics & Applications*, 5(1):21–40, January 1985.

[29] J. C. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. In *Proceedings IEEE Visualization 96*, pages 327–334. Computer Society Press, 1996.

[30] X. Xiang, M. Held, and J. S. B. Mitchell. Fast and effective stripification of polygonal surface models. In *Proceedings Symposium on Interactive 3D Graphics*, pages 71–78. ACM Press, 1999.