Efficiently Computing and Updating Triangle Strips for Real-Time Rendering

Jihad El-Sana[†‡] Francine Evans[†§] Aravind Kalaiah[†]

Amitabh Varshney[†] Steven Skiena[†] Elvir Azanli[†]

Abstract

Triangle strips are a widely used hardware-supported data-structure to compactly represent and efficiently render polygonal meshes. In this paper we survey the efficient generation of triangle strips as well as their variants. We present efficient algorithms for partitioning polygonal meshes into triangle strips. Triangle strips have traditionally used a buffer size of two vertices. In this paper we also study the impact of larger buffer sizes and various queuing disciplines on the effectiveness of triangle strips. View-dependent simplification has emerged as a powerful tool for graphics acceleration in visualization of complex environments. However, in a view-dependent framework the triangle mesh connectivity changes at every frame making it difficult to use triangle strips. In this paper we present a novel data-structure, Skip Strip, that efficiently maintains triangle strips during such view-dependent changes. A Skip Strip stores the vertex hierarchy nodes in a skip-list-like manner with path compression. We anticipate that Skip Strips will provide a road-map to combine rendering acceleration techniques for static datasets, typical of retained-mode graphics applications, with those for dynamic datasets found in immediate-mode applications.

1 Introduction

Recent advances in three-dimensional acquisition, simulation, and design technologies have led to generation of datasets that are beyond the interactive rendering capabilities of current graphics hardware. Several software and algorithmic solutions have been recently proposed to bridge the increasing gap between hardware capabilities and the complexity of the graphics datasets. These include level-of-detail rendering with multi-resolution hierarchies, occlusion culling, and image-based rendering. Graphics rendering has also been accelerated through compact representations of polygonal meshes using data-structures such as triangle strips and triangle fans.

Although each triangle can be specified by three vertices, but to maximize the use of the available data bandwidth, it is desirable to order the triangles so that consecutive triangles share an edge. Such ordered sequences of triangles are referred to as *triangle strips*. Using such an ordering, only the incremental change of one vertex per triangle need be specified, potentially reducing the rendering time by a factor of three by avoiding redundant transformation, clipping, and lighting computations. Besides, such an approach also has obvious benefits in compression for storing and transmitting models. In Section 2 we overview the previous work done in generation of triangle strips, include some variations to the simple model of a triangle strip as

[†]Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794, USA

[‡]Department of Mathematics & Computer Science, Ben-Gurion University, Beer-Sheva, 84105, Israel

[§]Schlumberger, 5599 San Felipe, Houston, TX 77056, USA

outlined above. In Section 3 we consider the problem of constructing good triangle strips from polygonal models and present several possible approaches.

Recently, view-dependent simplifications have been introduced to enable fine-grained changes to multiresolution hierarchies that depend on parameters such as view location, illumination, and speed of motion. Such simplifications change the mesh structure at every frame to adapt to just the right level of detail necessary for visual realism. One drawback of such schemes is that they make it difficult to take advantage of hardware-supported mechanisms for graphics acceleration, such as triangle strips. Luebke and Erikson [27] point out that view-dependent simplification being an immediate-mode technique has a relative disadvantage since most current graphics hardware takes advantage of retained-mode representations such as display lists that have static geometry and connectivity. To overcome this drawback Hoppe [23] has proposed a solution to compute triangle strips per frame for the view-dependent simplification specific to that frame. In Section 5 we present Skip Strips as a solution to this dichotomy of immediate-mode simplifications and retained-mode hardware-supported acceleration.

The results of our approaches are presented in Section 6. Special-purpose rendering hardware is needed to fully exploit the advantages of triangle strips, by maintaining a buffer with the k previously transmitted vertices as determined by a certain queuing discipline. Although current rendering engines use a buffer of size of $k = 2$ and FIFO queuing discipline, there has been recent interest in studying the impact of larger buffer sizes, for both rendering [4] and geometric compression [9] Towards this end, we provide extensive analysis of the impact of buffer size and queuing discipline on triangle strip performance in Section 6.2. We demonstrate that relatively small buffer sizes are sufficient to achieve most of the potential benefits of triangle strips, making for a desirable tradeoff between increasing hardware cost versus the speedup in rendering time. This paper presents a summary of research in [14, 11] in an archival form and augments it with a survey of the area of triangle strips and its variants.

2 A Brief Survey of Triangle Strips and Related Data-Structures

Triangle strips provide a compact representation of triangular meshes and are supported by several graphics APIs including OpenGL [31, 32]. Triangle strips enable fast rendering and transmission of triangular meshes. An example triangle strip in the model of a cow is shown in Figure 1. The set of triangles shown in Figure 2(a) can be compactly represented by a triangle strip $(1, 2, 3, 4, 5, 6)$, where the i^{th} triangle is described by the i^{th}, $(i + 1)^{st}$, and $(i + 2)^{nd}$ vertices in this sequence. Such triangle strips are referred to as *sequential triangle strips*. A sequential triangle strip allows rendering of n triangles using only $n + 2$ vertices instead of $3n$ vertices. This results in substantial savings for memory bandwidth and computation of per-vertex operations such as transformations, lighting, and clipping.

Sequential triangle strips cannot however represent general sequences of triangles, such as the one shown in Figure 2(b). To represent such triangle sequences, the notion of triangle strips has been extended to *generalized triangle strips* where the two vertices of the previous triangle can be swapped. This can be also simulated by repeating vertices. Thus, the triangle sequence in Figure 2(b) can be represented as $(1, 2, 3, 4, 5, 4, 6, 7)$.

Akeley *et al.* [1] have developed a program that constructs generalized triangle strips for a given triangle mesh model [1]. This algorithm seeks to create triangle strips that tend to minimize leaving isolated triangles. It is a greedy algorithm, which always chooses as the next triangle in a strip the triangle that is adjacent to the least number of neighbors (i.e. minimizes the number of *adjacencies*). When there is more than one triangle with the same, least number of neighbors, the algorithm looks one level ahead to its neighbors' neighbors, and chooses the direction of minimum degree, choosing arbitrarily if there is again a tie. After starting from an arbitrary lowest degree triangle, it extends its strips in both directions, so that each strip is as long as possible. There is no reluctance to generate swaps, and understandably so, since this algorithm was aimed

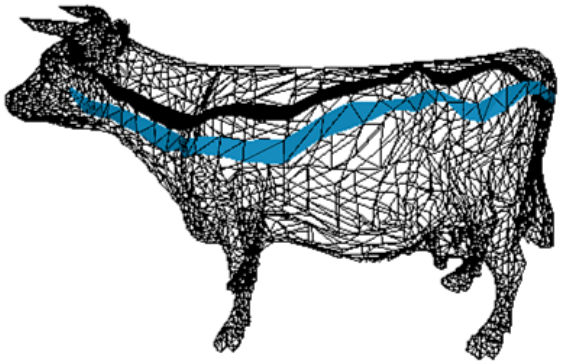


Figure 1: A triangle strip in a cow model

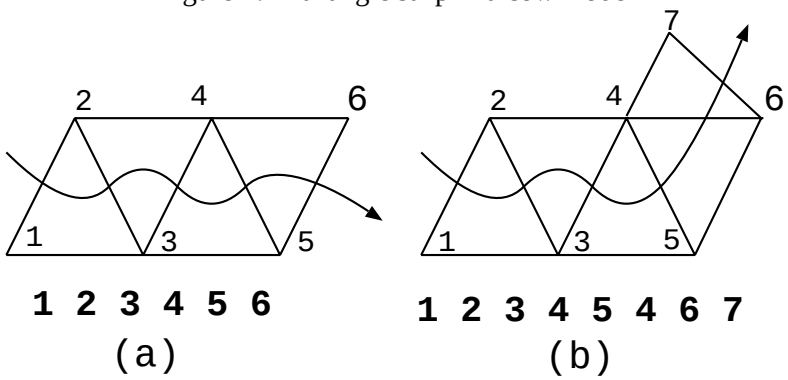


Figure 2: Examples of (a) Sequential and (b) Generalized triangle strip

at generating triangle strips for Iris GL in which the cost for a swap was just one bit. A fast, linear-time implementation of this algorithm is obtained by using hash tables to store the adjacency information, linked to a priority queue maintaining strip length to choose which triangle starts a new strip. Speckmann and Snoeyink [35] have computed the triangle strips for triangulated irregular networks by creating a spanning tree of the dual graph of the TIN and then traversing the tree in a modified depth-first fashion. More recently, Xiang *et al.* [40] have presented a triangle stripping algorithm that computes a spanning tree of the dual graph of a triangulation, partitions this tree into triangle strips, and then concatenates these triangle strips into larger strips.

Within computational geometry, interest has focused on constructing and recognizing Hamiltonian and sequential triangulations. A triangulation is *Hamiltonian* if its dual graph contains a Hamiltonian cycle. Hamiltonian triangulations can be represented by using generalized triangle strips (triangle strips with swaps). Arkin, *et.al.* [3] proved that every point set has a Hamiltonian triangulation. Further, they showed that the problem of testing whether a triangulation is Hamiltonian is NP-complete. They gave an $O(n^2)$ algorithm for constructing a Hamiltonian triangulation of a polygon that has since been improved to $O(n \lg n)$ by Narasimhan [29].

A triangulation is *sequential* if its dual graph contains a Hamiltonian cycle whose turns alternate left-right. Sequential triangulations can be represented by using one triangle strip without any swaps. A Hamiltonian triangulation is sequential if three consecutive edges do not share a common vertex. Arkin, et.al. [2] proved that for any $n \geq 9$ there exists a set of n points in general position that do not admit a sequential triangulation. Although linear time suffices to test whether a triangulation is sequential, we have proved that the problem of finding a sequential triangulation of a partially triangulated surface is NP-complete using a reduction from 3-satisfiability [15]. Hence, heuristics such as those described in this paper are required to find good sequential strips.

A simple path in the dual of a triangulation identifies a sequence of triangles that form a "strip" or a (triangular) "ribbon" . Bhattacharya and Rosenfeld [5] have studied geometric and topological properties of ribbons. The Hamiltonian triangulation problem can be considered that of identifying if a set of points or a polygon has a triangulation that consists of a single strip (triangular ribbon). Bose and Toussaint [6] have recently studied a set of problems involving *quadrangulation* of point sets, and have obtained several interesting results. A quadrangulation of a point set S is a decomposition of the convex hull into quadrilaterals, such that each point of S is a vertex of some quadrilateral. In particular, they have applied the notion of Hamiltonian triangulations to this problem, and they have obtained an alternate method of computing Hamiltonian path triangulations.

Although a triangle strip helps in avoiding the repeated processing of vertices that belong to shared edges inside the strip (such as edges $(2,3),(3,4)$, and $(4,5)$ in Figure 2(a)), the edges that occur on the sides of such a strip (for instance, edges $(2,4),(4,6),(1,3)$ and $(3,5)$ in Figure 2(a)) represent sets of vertices that could be repeated in adjacent triangle strips. Traditional triangle strips have operated with a buffer size of 2 vertices. If a large buffer size is allowed the repetition is reduced because older vertices can now be re-used. By Euler's theorem on graphs, the number of triangles in a triangulation is at most twice the number of vertices, and on average we will have to send each vertex twice to the renderer using sequential triangle strips and a buffer of size 2.

The decomposition of a triangular mesh into a triangle-strip data-structure that back-references the previous k vertices, $k \geq 2$ is referred to as a *generalized triangle mesh* [9]. Deering has proposed the use of generalized triangle meshes for compressing connectivity information in geometric polygonal models [9]. He has proposed maintaining a stack of size $k = 16$ to store 16 previous vertices. A vertex for a new triangle is specified either through back-referencing one of the existing vertices on the stack, or by reading-in a new vertex and replacing an existing vertex on the stack. Although a novel idea, no algorithms have been proposed there to suggest how one can decompose polygonal models into generalized triangle meshes for a given buffer size k. Chow [7] presents local and global algorithms for converting triangular and polygonal meshes into generalized meshes. The local algorithm proceeds by starting from the mesh boundary and successively peeling off triangle strips outwards in. The length of the strip is based on buffer size k. The global algorithm presented there is based on our global algorithm for patchification [14] and detailed in Section 3. More recently Hoppe [24] has proposed a greedy triangle-strip growing algorithm that performs vertex caching in a transparent manner. It creates greedy triangle strips in a manner similar to Akeley et al. [1] but then cleverly uses the cache size information to break strips before they become too large and make vertex reuse difficult.

Mitra and Chiueh [28] have studied the effect of increasing the buffer size on reducing the repetition of vertices. Mitra and Chiueh however use an encoding scheme that differs from the triangle-strip specification scheme. They encode the mesh using two buffers – *current frontier* and *next frontier* and use a breadth-first traversal scheme to update these buffers. A triangle is rendered using vertices from the two buffers. Their results show that they can use a 64-vertex buffer and have less than 8% repetition of vertices. Their empirical results on several meshes ranging from $3,000$ to $40,000$ triangles suggest that the size of the vertex buffer required is independent of the mesh size.

Bar-Yehuda and Gotsman [4] studied the extent to which we can increase the stack (buffer) size to reduce this duplication of vertices. This yields a time-versus-space tradeoff; as we increase memory usage, rendering time will decrease. Bar-Yehuda and Gotsman have shown that a buffer of size $13.35\sqrt{n}$ is sufficient to render any mesh on n vertices in the optimal time n, and that a buffer size of $1.649\sqrt{n}$ is necessary for optimal rendering in the worst-case. They show the problem of minimizing the buffer-size for a given mesh is NP-hard, using a reduction from the problem of finding minimum separators of a planar graph.

Triangle strips can be viewed as a method to compress the connectivity information in a triangle mesh. Several papers have been published that provide better alternatives than the approach of converting polyg-

onal models to triangle strips [22, 37, 20, 34]. For a good survey of such techniques, the interested reader should refer the survey by Rossignac [34].

Much less work has been done in integrating triangle strips with multiresolution rendering. Duchaineau *et al.* [10] have proposed performing incremental view-dependent sequential triangle strips for terrains. Their underlying mesh representation is a triangle bintree and as triangles are split or merged, these changes are reflected in the triangle strips for the previous frame. These updated triangle strips are then re-linked to generate longer strips. Velho *et al.* [38] have proposed an elegant technique that generates a recursive hierarchy of triangle strips for uniform or adaptive tessellations of implicit and parametric patches. Their method generates triangle-strips that cover the original patch using a variant of space-filling curves.

3 Constructing Efficient Triangle Strips

In this section, we propose several heuristics for constructing triangle strips from polygonal models. There are at least three different objectives such heuristics might reasonably seek to achieve:

- *Maximize the length of each strip* – since each strip of length s represents $s-2$ triangles, maximizing strip length minimizes this overhead.
- *Minimize swaps* – since each swap costs one additional vertex in the OpenGL cost model.
- *Minimize the number of singleton strips* – since each triangle left isolated after removing a strip creates a singleton strip, we should seek to begin and end our strips on low-degree faces of the triangulation.

We have experimented with several variants of local and global algorithms, as discussed in the following two sub-sections.

3.1 Local Algorithms

Our class of local heuristics starts from the same basic idea as [1] – to use least adjacencies as the basis for choosing the next face in a strip. However, we have tried to improve upon their algorithm by dynamic triangulation and alternate tie-breaking procedures. We have considered three different approaches to triangulating faces:

- *Static triangulation* – In this approach, we triangulate all quads and larger faces in our model as a pre-processing step before we begin finding strips. We use alternate left-right turns, as shown in Figure 3(b) because such a triangulation is inherently sequential, as opposed to the simpler and more conventional fan triangulation.

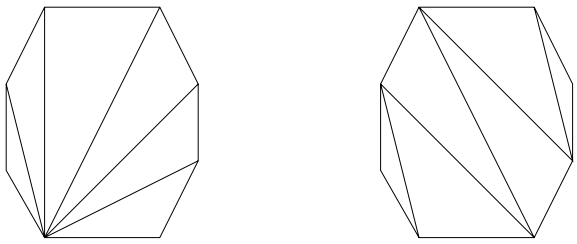


Figure 3: Fan versus sequential triangulation of a convex polygonal face.

- *Dynamic whole-face triangulation* – A second approach completely triangulates each face when we first enter it via some edge on a strip. After using one of the tie-breaking procedures described below to determine the exit edge e, we can triangulate the face as sequentially as possible while exiting at e. If the surface normals do not vary across a face, then whole face triangulation has the additional advantage of encoding fewer normal transitions.

- *Dynamic partial-face triangulation* – Partial-face triangulation provides the freedom to triangulate and walk only part of a face before exiting it. This approach can under certain conditions provably perform better than the whole-face triangulation, as is seen in the example where we represent a cube using a single sequential triangle strip. After identifying the exit edge e of the face with the minimum number of adjacencies, we sequentially triangulate the smallest portion possible of the face from the input edge to exit at e. This is illustrated in Figure 4.

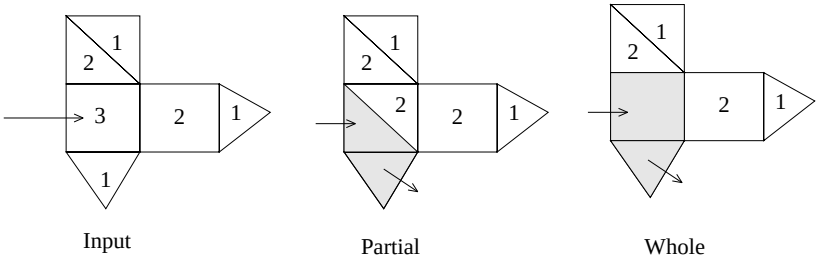


Figure 4: Examples of partial and whole-face triangulation.

We have considered several different approaches in breaking ties when there is more than one polygon that has the least number of adjacencies to the current face. Such ties often occur since the possible number of adjacencies ranges only over 1, 2, and 3. In particular, we tried:

- *Arbitrary* – meaning that we use the first face found among the low-adjacency faces.

- *Look-ahead* – this is the same approach that algorithm presented in [1] takes, as described above.

- *Alternate* – this rule tries to alternate directions in choosing the next polygonal face. To motivate this option, note that sequential strips alternate directions.

- *Random* – chooses the next face randomly from those that were tied.

- *Sequential* – chooses the next face that will not produce a swap, and picks randomly if there is no such face.

To quickly identify the lowest adjacency face to start from, we maintain a priority queue ordered by the number of adjacent polygons to each face. The faces in the priority queue are linked to the adjacency list data structure representing the dual graph of the triangulation. This enables fast lookup to find and delete faces when forming the triangle strips.

3.2 Global Algorithms

Although the problem of finding the strip-minimal triangulation is NP-complete, we perform a global analysis of the structure of a polygonal model using a technique we call *patchification*, which we believe is of independent interest. In typical polyhedral models, there are many quadrilateral faces, often arranged in

large connected regions. We attempt to find large "patches", rectangular regions consisting only of quadrilaterals, as illustrated in Figure 5. These patches can be triangulated sequentially along each row or column, although there is a cost of either 3 swaps per turn or 2 vertices to stop and restart each strip at the end of a row or column.

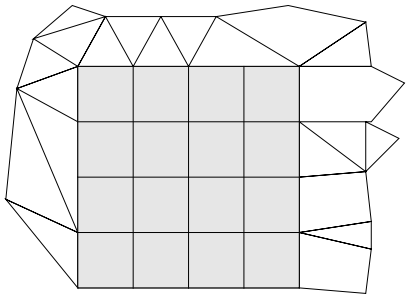


Figure 5: Patchification finds large rectangular patches of quadrilaterals.

Efficient patchification requires computing the number of polygons to the east, west, north, and south of each face, and making sure that when forming the patches, the polygons in the patch are all adjacent. Hence, we have to "walk" through the faces and calculate the number of adjacent polygons to them in each orientation. Each "walk" only visits each face exactly 2 times: once for the north-south direction and once for the east-west direction; once we visit a face in a walk, that face does not require visiting again. To avoid generating too many small patches, we keep a *patch cutoff size* which is the area of the smallest patch we would like to generate. Since we generate patches in decreasing order of size, we can conveniently stop the process once the areas of the patches being generated falls below this cutoff size. This approach takes us time $O(pn)$ where p is the number of patches found. In our studies p was much smaller than n and therefore this approach demonstrated a linear behavior. We tried two different approaches for exploiting the coherence identified in large patches:

- *Row or column strips* – After selecting all patches whose size was greater than a specified cutoff size, we partitioned the patches into sequential strips along rows or columns (whichever direction yielded larger strips) and deleted them from the model. Next, a local algorithm (using whole-face triangulation) was used on the remaining model. By generating one strip along each row or column, we minimize the number of swaps needed.
- *Full-patch strips* – Each patch larger than the cutoff size was converted into one strip, at a cost of 3 swaps per turn. Further, every such strip was extended backwards from the starting quadrilateral and forwards from the ending quadrilateral of the patch to the extent possible. As before, the local algorithm was used on the model left after removing the patches and their forward and backward extensions.

The results of our experiences with these heuristics are discussed in Section 6.1. In our experiments we found that row or column strips for patchification performed better than full-patch strips and partial-face triangulation better than whole-face triangulation. Also, sequential method of breaking ties was found to be the best for minimizing the number of vertices to be sent.

4 Issues in Integrating Triangle Strips with Multiresolution Hierarchies

Thus far we have considered strategies for generating triangle strips in environments where each object is represented at a single resolution; the geometry and the connectivity for the object is static. However, use of

multiresolution hierarchies in level-of-detail-based rendering schemes for real-time graphics environments is becoming increasingly common these days. There are two kinds of multiresolution hierarchies – *discrete* and *continuous*. The discrete multiresolution hierarchies involve computing a fixed number (usually under 10) levels of detail for an object. Perceptually important objects in a visual scene are then rendered using higher levels of detail and perceptually unimportant objects are rendered using lower levels of detail [16]. Such schemes are useful for applications where the perceptual importance varies significantly from one object to another, but not across the same object. However, in several applications, the perceptual importance varies significantly across the same object. Examples include flying over a single terrain object, visualizing a connected iso-surface, and examining a single protein molecule. In such cases it is important to vary the detail across different regions of the same object. Such changes in detail have to be continuous across an object and require multiresolution hierarchies that can generate them on the fly based on view- and scene-parameters. For discrete multiresolution hierarchies, triangle strips can be computed once per level of detail as a pre-process using techniques outlined above in Section 3. However, such solutions do not extend to continuous multiresolution hierarchies. In Section 5 we present a solution for efficiently updating triangle strips in presence of connectivity changes that accompany view-dependent modifications based on continuous multiresolution hierarchies. To better explain our approach, we next give a brief overview of view-dependent simplification and efficient traversal of pointers in data-structures.

4.1 View-Dependent Simplifications

Most of the previous work on generating multiresolution hierarchies for level-of-detail-based rendering has concentrated on computing a fixed set of view-independent levels of detail. At runtime an appropriate level of detail is selected based on viewing parameters. Such methods are overly restrictive and do not take into account finer image-space feedback such as light position, visual acuity, silhouettes, and view direction. Recent advances to address some of these issues in a view-dependent manner take advantage of the temporal coherence to adaptively refine or simplify the polygonal environment from one frame to the next. In particular, adaptive levels of detail have been used in terrains by Gross *et al.* [18] and Lindstrom *et al.* [26]. A number of techniques for conducting view-dependent simplifications of generalized polygonal meshes rely on the primitive operations of vertex-split and edge collapse as shown in Figure 6. The edge (pc) in the mesh on the left collapses to the vertex p and the resulting mesh is shown on the right. Conversely, the vertex p in the mesh on the right can split to the edge (pc) to generate the mesh on the left. Let the vertex p be considered the parent of the vertex c (as c is created from p through a vertex split). The primitives of vertex split and edge collapse were proposed in the context of progressive meshes [22].

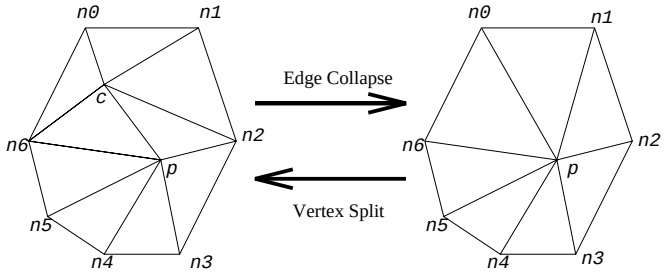


Figure 6: Edge collapse and vertex split

View-dependent simplifications using the edge-collapse/vertex-split primitives include work by Xia *et al.* [39], Hoppe [23], and Gueziec *et al.* [19]. View-dependent simplifications by Luebke and Erikson [27], and De Floriani *et al.* [8] do not rely on the edge-collapse primitive. Our work is most directly applicable to view-dependent simplifications that are based upon the vertex-split/edge-collapse primitive; its extension

to more general view-dependent simplifications is a part of our planned future work. We next overview a representative view-dependent simplification algorithm that is based on *Merge Trees*. In Section 5, we use this algorithm and associated data-structures to explain updating of triangle strips in view-dependent environments.

4.2 Merge Trees

Merge trees have been introduced by Xia *et al.* [39] as a data-structure built upon progressive meshes [22] to enable real-time view-dependent rendering of an object. As discussed earlier, let the vertex p in Figure 6 be considered the parent of the vertex c. The *neighborhood* of a vertex v is defined as the set of triangles that are adjacent to v. The neighborhood of an edge (v_a, v_b) is defined as the union of neighborhoods of v_a and v_b. The merge tree is constructed in a bottom-up fashion from a high-detail mesh to a low-detail mesh by storing these parent-child relationships (representing edge collapses) in a hierarchical manner over the surface of an object. At each level l of the tree a maximal set of edge-collapses is selected in the shortest-edge-first order and with the constraint that their neighborhoods do not overlap. The vertices remaining after these edge collapses are promoted to level $l + 1$.

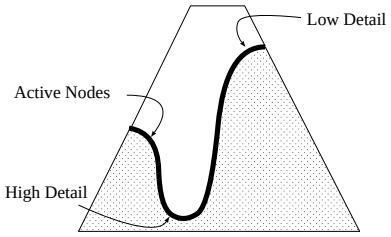


Figure 7: Varying detail in a Merge Tree

View-dependent simplification is achieved by performing edge-collapses and vertex-splits on the triangulation used for display depending upon view-dependent parameters such as lighting (detail is directly proportional to intensity gradient), polygon orientation, (high detail for silhouettes and low detail for back-facing regions) and screen-space projection. This is shown in Figure 7. Since there is a high temporal coherence the selected levels in the merge tree change only gradually from frame to frame. Unconstrained edge-collapses and vertex-splits during runtime can be shown to result in mesh foldovers resulting in visual artifacts such as shading discontinuities. To avoid these artifacts Xia *et al.* propose the concept of dependencies or constraints that necessitate the presence of the entire neighborhood of an edge before it is collapsed (or its parent vertex is split). Thus, for the example shown in Figure 6, the neighborhood of edge pc should consist exactly of vertices $n_0 \ldots n_6$ for c to collapse to p. Similarly, for the vertex p to split to c, the vertices adjacent to p should be exactly the set $n_0 \ldots n_6$. Our current implementation of merge trees can construct the merge tree for 69K triangles bunny model in 10.3 seconds on an SGI Onyx 2.

4.3 Efficient Link Traversal

Let us study what happens when an edge collapses in a triangle strip. Figure 8 shows such a situation. As can be seen, the results of an edge collapse can be represented by replacing all occurrences of the child vertex c with the parent vertex p. In this example, $c = 2$ and $p = 4$.

The above example illustrates that to maintain triangle strips under view-dependent changes to the triangle mesh connectivity, we should replace each vertex in a triangle strip by its nearest uncollapsed ancestor. In an arbitrarily long sequence of such edge collapses, it is easy to see why efficient traversal of links to a vertex's ancestors becomes important.

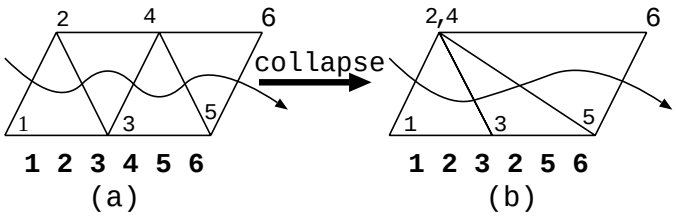


Figure 8: Edge Collapse in a Triangle Strip

Skip list [33] has been proposed as an efficient probabilistic data-structure to store and retrieve data. Skip lists can also be used for efficient compression of pointer paths. Consider a simple linked list as shown in Figure 9(a). Reaching the n-th node on this list requires O(n) pointer hops. Consider next a data-structure that resembles a binary tree and has O(n) additional pointers that connect linked-list nodes that are 2 away, 4 away, . . ., $2^{\log n}$ away (refer Figure 9(b)). Using these additional pointers, any node on the linked list can be accessed in O($\log n$) hops. Skip lists generate such additional pointers in a probabilistic manner to provide the same O($\log n$) access time (refer Figure 9(c)), but in practice have been shown to be faster.

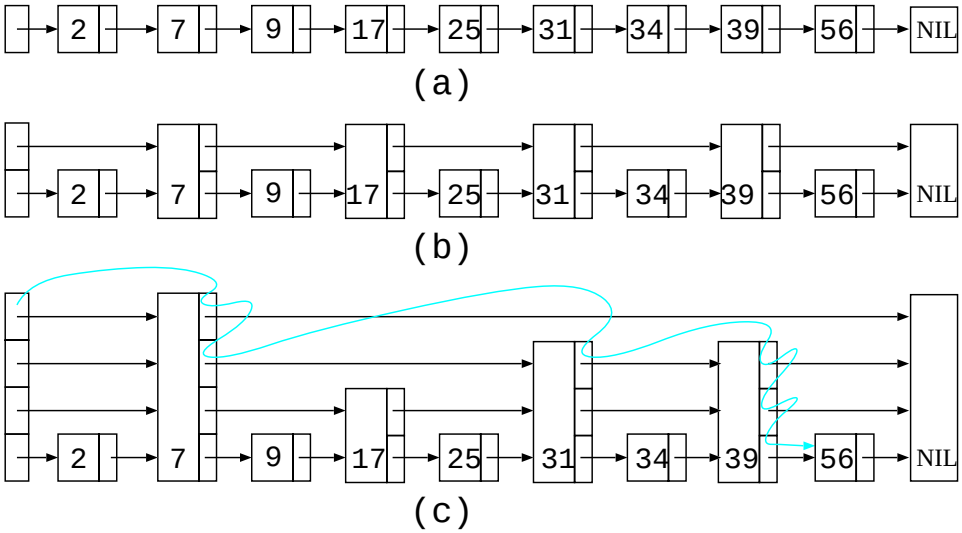


Figure 9: Skip list example

In a skip list, a node that has k forward pointers is a level k node. The level of a node is determined in a probabilistic manner. The search for an element is done by traversing forward pointers that do not overshoot the required element. When no more progress is possible, the search moves down to the next level. This is shown by the gray path in Figure 9(c) that searches for element with id-number 56. To accomplish insertion or deletion of an element in a skip list, a search is carried out for that element using the above method. A vector of pointers is set up during this search that represents the set of pointers that are changed to implement the insert or delete operation.

5 Skip Strips

In our approach we first generate a merge tree file as overviewed above in Section 4.2 and described in [39]. This file contains the parent-child relationships for each node of the tree. Even though our implementation uses merge trees, the concept of Skip Strips is quite general and can be used in conjunction with other vertex-collapse-based simplification schemes as well. We next generate the triangle strip representation of the

original polygonal model using any of the techniques for generating triangle strips as described in Section 3. At run-time we load the merge tree and the triangle strip representations generated during preprocessing and build the *Skip Strip* data-structure on the fly. Then, depending on scene parameters such as eye position, local illumination, front/back-facing regions, we perform vertex-split and edge-collapse operations directly on the Skip Strips. The information from Skip Strips is then used to generate triangle strips for display.

5.1 Skip Strip Data-Structure

A Skip Strip is an array of Skip Strip nodes. Each Skip Strip node contains vertex information, a list of child pointers and a parent pointer. We shall see in Section 5.3 how to generalize this data-structure to support a list of parent pointers to accelerate access in a edge-collapse hierarchy. Exactly one of the child pointers is marked as an *active* child pointer. This can be seen in Figure 10 where the parent pointers are shown on the right and the list of child pointers is shown on the left of each Skip Strip node. Also, the parent pointer of the node is marked *active* if this node has collapsed to its parent at a given stage of simplification; otherwise it is marked *inactive*.

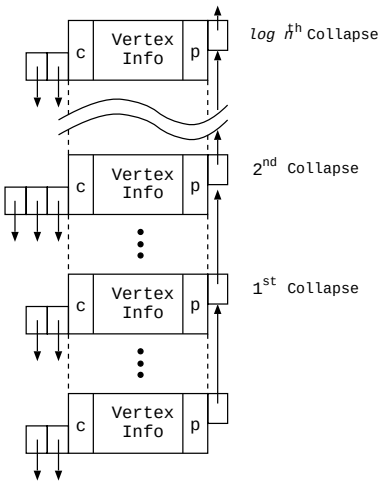


Figure 10: A Skip Strip node

A Skip Strip is constructed at run time from the merge tree and the triangle strip representations. A Skip Strip node is allocated for every merge tree node and then parent-child pointers are set up to mimic the merge tree structure. In our current implementation we are assuming that a child vertex c collapses to a parent vertex p. For this case, a Skip Strip node corresponding to a vertex p will have child pointers to all its children, including c, that collapse to it at different stages of simplification. In general, if there are n vertices then the height of the merge tree is $O(\log n)$. Thus, the length of this child-pointer list for a Skip Strip node could be $O(\log n)$. At a given time only one of these child pointers is flagged *active* and represents the node that will result from the most imminent split. Each Skip Strip node points to its immediate parent via the parent pointer.

To illustrate the Skip Strip data-structure, let us see how it is built from a merge tree. Figure 11(a) shows a hypothetical merge tree over four vertices 1 to 4. As in all the merge tree diagrams in this paper, the right node is the child node and the left node is the parent node (as defined by Figure 6). The equivalent Skip Strip data-structure will have four nodes representing the leaves of the merge tree (the highest detail vertices in the original model). Since according to the merge tree vertex 2 can merge to vertex 1, the parent pointer for the Skip Strip node 2 will point to Skip Strip node 1 and the child pointer for the node 1 will point to node 2. Similarly the parent and child pointers of Skip Strip nodes 3 and 4 will be set. This stage is shown

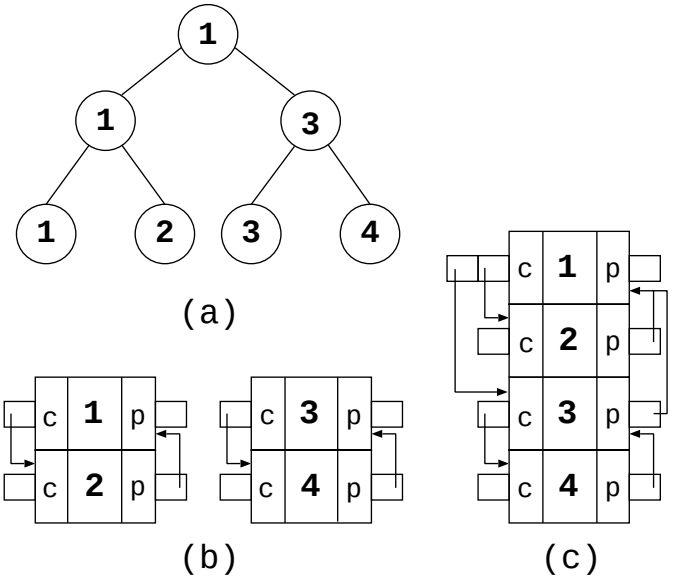


Figure 11: Building a Simple Skip Strip

in Figure 11(b). The final structure in which node 3 merges with node 1 can be represented in the Skip Strip as a parent pointer from node 3 to node 1 and a child pointer from node 1 to node 3. The completed Skip Strip structure is shown in Figure 11(c).

The method that we have outlined above assumes that in an edge collapse from c to p, the new vertex is p. However, several other researchers have pointed out the advantage of creating new vertices during edge collapses. These new vertices could be created for accomplishing geomorphs [22] or for better placement of approximating vertices using sophisticated error metrics [17, 25]. For incorporating such simplification metrics into the framework of Skip Strips we suggest storing multiple coordinate sets, once per approximating vertex, in the child pointer of the Skip Strip node.

5.2 Real-Time Adaptive Representation

Once the Skip Strip has been constructed it is easy to construct an adaptive level-of-detail mesh representation during run-time. Real-time adaptive mesh representation involves the determination of the vertices and the triangle strips at the current level of detail. We shall refer to the vertices and triangle strips selected for display at a given frame as *display vertices* and *display strips*.

The algorithm proceeds by first generating triangle strips for the input mesh at the highest resolution. The Skip Strip data-structure generation outlined in Section 5.1 is independent of the generation of triangle strips. At run-time the display vertices are determined using view- and illumination-dependent parameters. This step is discussed in Section 5.2.1. The appropriate edge-collapses are performed on the set of original triangle strips to generate a set of display strips. This is outlined in Section 5.2.2. The display strips are determined incrementally and efficiently (Section 5.3) and are filtered (Section 5.4) before sending them to the graphics processor.

5.2.1 Determination of display vertices

Determination of display vertices proceeds along the same lines as proposed in earlier work on view-dependent simplification [39, 23] where image-space feedback is used to guide the selection of the level

of detail for the mesh. We determine which region of an object to simplify more and which to simplify less using several parameters such as viewer location and orientation, local illumination, and front/back-facing regions of an object. Similar to merge tree nodes, Skip Strips nodes also store a *switch value* to determine whether to refine, merge, or leave a Skip Strip node in its current level. If the computed value of the view-dependent error at a given node v is less than the *switch value* stored at node v, then node v splits. If the computed value is larger than the *switch value* stored at the parent of node v, then v merges.

In addition to the above criteria, each collapse and split also depends on the validity of the operation as determined during the preprocessing to avoid artifacts such as mesh foldovers as explained earlier in Section 4.2. One way to avoid such artifacts is to use dependencies [26, 39]. In [12], we have introduced the concept of implicit dependencies that can test validity of edge collapse or vertex split in constant time. However, implicit dependencies rely on the existence of independent triangles that can be individually tagged. Since in the Skip Strip data-structure we do not store triangles explicitly it is difficult to use implicit dependencies. For Skip Strips we can use the traditional method of storing dependencies explicitly as a set of adjacent nodes [39]. Instead, we have chosen to optimize the explicit dependencies by storing only that subset of adjacent nodes that do not participate in an ancestor-child relationship, i.e. we do not include an adjacent node in the dependency list if any of its ancestors is already in the list.

The execution of edge collapse and split operation is done in a small constant time (only integer increment and flag change or integer decrement and flag change) as follows. To perform a merge on the Skip Strip we activate the parent pointer and increment the child index of the merged node by one, followed by removing the merged node from the active nodes list. Split is done by deactivating the parent pointer and decrementing child index of the split node by one. Then we insert the node pointed to by the previous child index into the active nodes list. We have discovered that these simpler operations have reduced the time for checking and performing a vertex split or edge collapse from around 60μseconds to 6μseconds.

5.2.2 Determination of display strips

Our scene is represented as a set of triangle strips. Each triangle strip has two representations – the original highest resolution triangle strip that was generated using pre-processing and the Skip-Strip-derived run-time representation of it that represents a triangle strip suitable for the current level of detail. We refer to the former as a *original triangle strip* and the latter as a *display strip*. At each frame we first perform view dependent edge collapses/vertex splits as outlined in Section 5.2.1. Each time an edge collapses or vertex splits, all display strips that contain that edge are flagged as modified. At the end of these simplifications if a display strip remains unmodified it is used for rendering. However, if a display strip is modified we discard it and begin generating its replacement by scanning each vertex in the corresponding original triangle strip. Each vertex of the original triangle strip has a pointer to a corresponding node in a Skip Strip. For each vertex's node in the Skip Strip we check if its parent pointer is active or not. If the parent pointer is active we follow the sequence of active parent pointers until we reach a node that has an inactive parent pointer. The vertex information stored with the first node that has an inactive parent pointer is added to the new display strip. After the new display strip has been completely generated it is sent to the graphics system for display.

Let us next illustrate how the Skip Strips are used to split and collapse vertices of a triangle strip to generate the display strips. Figure 12 shows the original mesh with vertices numbered $1..10$. The two triangle strips representing this mesh are labeled a and b. Since no edges have collapsed, the display strips are the same as the original triangle strips. Figure 13 shows the merge tree and the skip strip with one parent pointer per node, constructed for the mesh in Figure 12 at the highest detail. Figure 14 shows the same after two edge collapses ($6 \rightarrow 5$, and $8 \rightarrow 7$) to the mesh of Figure 12. In Figure 13 none of the parent pointers is active (since there have been no edge collapses). In Figure 14, the parent pointers for nodes 6 and 8 that

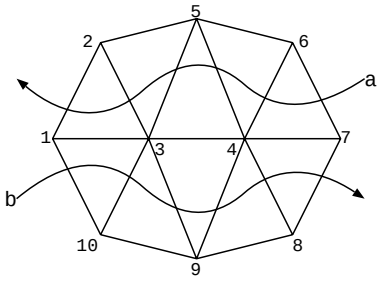


Figure 12: Original triangle mesh

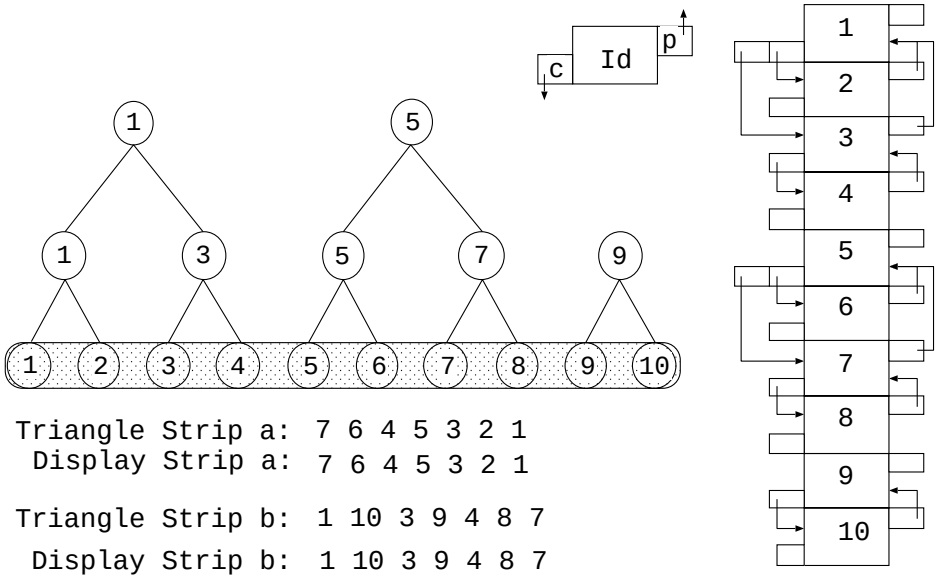


Figure 13: Skip Strip for Triangle Mesh in Figure 12

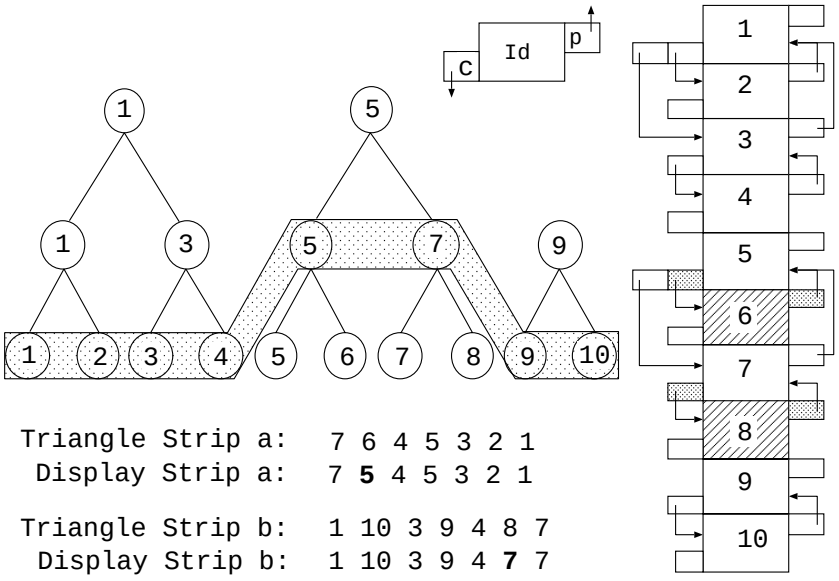


Figure 14: Skip Strip for Triangle Mesh in Figure 12 after two edge collapses

point to 5 and 7 respectively, are active and appear shaded.

5.3 Efficient Skipping for Parent pointers

As the object moves to a coarser representation, the time spent in following the active parent pointers increases. The maximum number of active parent pointers that one might need to traverse is $O(\log n)$ – the height of the vertex hierarchy. To reduce this time we trade off memory for speed. To accomplish this we use ideas from path compression [36] and skip lists [33] to build a list of parent pointers for each Skip Strip node that point to ancestors of this node that are $1, 2, 4, 8, \ldots, \log n$ away in the edge collapse hierarchy. By using an efficient, skip-list-like pointer hopping scheme we can reduce this to $O(\log \log n)$. Although reducing a $O(\log n)$ factor might seem minor, in practice this results in an appreciable difference, especially when we note that the merge tree height is generally a logarithm to the base $5/4$ [39]. Thus, even if the edge-collapse-based vertex hierarchy tree is balanced (which often is not), the height for a tree over one million vertices (and therefore the worst-case pointer hopping) will be 62 ($\sim \log_{1.25}(10^6)$) while a skip-list-like pointer hopping scheme will only need to traverse 6 ($\sim \log_2 62$) pointers, an order of magnitude improvement for present-day datasets.

In this scheme each Skip Strip node has an active parent field to indicate which pointer in the parent list to follow to get closest to, without overshooting, the first active ancestor. We use a lazy update scheme to modify the active parent field for each Skip Strip node. For this we make use of the fact that the vertex hierarchy nodes are collapsed in an accordion-style fashion from high-detail to low-detail. In other words if a vertex i collapses to vertex j, then it means that *all* vertices that lie in the sub-tree rooted at vertex i have already collapsed to vertex i. If the triangle strips reference one of the vertices in this sub-tree rooted at i and if their active parent pointer overshoots j, then we need to decrement the active parent pointer till it points to a node that is below j (in other words has already collapsed). Because of a high temporal coherence, these updates are few and each requires only one or two ancestor checks to find the "correct" ancestor that does not overshoot the first active ancestor. Similarly, when a vertex j splits to vertices i and j we update all pointers from triangle strips that point to j as the first active ancestor to point to a lower level ancestor. We would like to point out that in this application, traversal of triangle strips requires that we access each vertex of the triangle strip and therefore the overhead of such lazy updates of pointers to reflect split and collapse in Skip Strips is minimal. Figure 13 shows the Skip Strip representation with multiple parent pointers, as described above, for the mesh of Figure 12. The active parent pointers appear in bold lines.

5.4 Filtering Triangle Strips

As the model moves to coarser levels the triangle strips begin to accumulate identical vertices. Sending such vertices multiple times is equivalent to sending degenerate triangles that do not contribute to the final scene but add an overhead to the graphics rendering. To address this we filter the triangle strips while sending them to the graphics engine. We have implemented a simple triangle strip scanner that detects and replaces patterns of vertices of the regular expression form $(aa)^+$ and $(ab)^+$ from the sequence of vertices sent for rendering and replaces them with (aa) or (ab) as appropriate.

6 Results

6.1 Triangle Strips

We have exhaustively tested our local and global algorithms on several datasets and compared them with the best-known public-domain triangle strip code that we shall refer to as the SGI code [1]. For our local

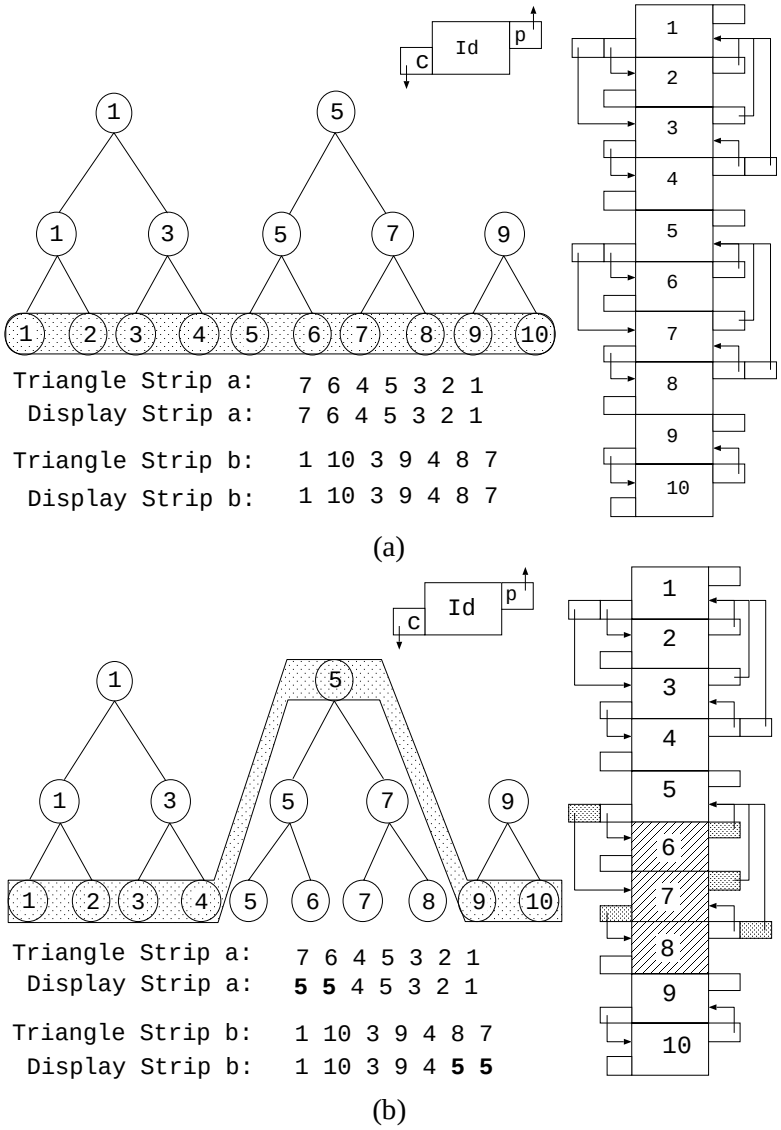


Figure 15: More efficient Skip Strip representations for Figures 13 and 14

approaches there were ten different options for each data file that we ran our experiments on: (a) whole-face triangulation and (b) partial-face triangulation, for each of the five tie breaking methods – (i) arbitrary, (ii) look-ahead, (iii) alternate, (iv) random, and (v) sequential. For our global approaches there were ten different options for each data file that we ran our experiments on: (a) row/column strips and (b) full-patch strips, for each of five different patch cutoff sizes of – 5, 10, 15, 20, and 25.

After comparing the results we had from the above-mentioned 20 different approaches on several datasets, we found that the best option was to use the the global row or column strips with a patch cutoff size of 5. We have implemented this option in our tool, *Stripe* Version 2.1. Table 1 compares the results for Stripe Version 2.1 against the SGI algorithm. For these results we are using the whole-face triangulation with sequential tie-breaking method for the regions of the mesh that are not covered by the global patchification approach. The cost columns show the total number of vertices required to represent the dataset in a generalized triangle strip representation under the OpenGL cost model (where each swap costs one vertex). We should point out that in our current implementation we are assuming that each polygon is convex. One can deal with

non-convex polygons by triangulating them first using one of several public-domain utilities [30, 21] or by interleaving the triangulation with the generation of triangle strips as proposed in this paper.

Data File	Vertices	Triangles	SGI Cost	Stripe Cost	% Savings
Air-Plane	1508	2992	4005	3571	10.8%
Skyscraper	2022	3692	5621	4849	13.8%
Triceratops	2832	5660	8267	7248	12.3%
Power lines	4091	8966	12147	10417	14.2%
Porsche	5247	10425	14227	12529	11.9%
Honda	7106	13594	16599	15060	9.3%
Bell ranger	7105	14168	19941	16892	15.3%
Dodge	8477	16646	20561	18920	8.0%
General	11361	22262	31652	28346	10.4%

Table 1: Comparison of triangle strip algorithms on representative models.

Figures 16, 17, and 18 show the performance comparisons between our best local and best global algorithms against the SGI algorithm for (a) GL and (b) OpenGL cost models. In the GL cost model each swap costs 1 bit (which we ignore) while in the OpenGL cost model each swap costs 1 vertex (since a vertex is repeated). In these figures the models are sorted by number of triangles are along the x-axis and the cost of generalized triangle strip representation is along the y-axis. Observations from these graphs include:

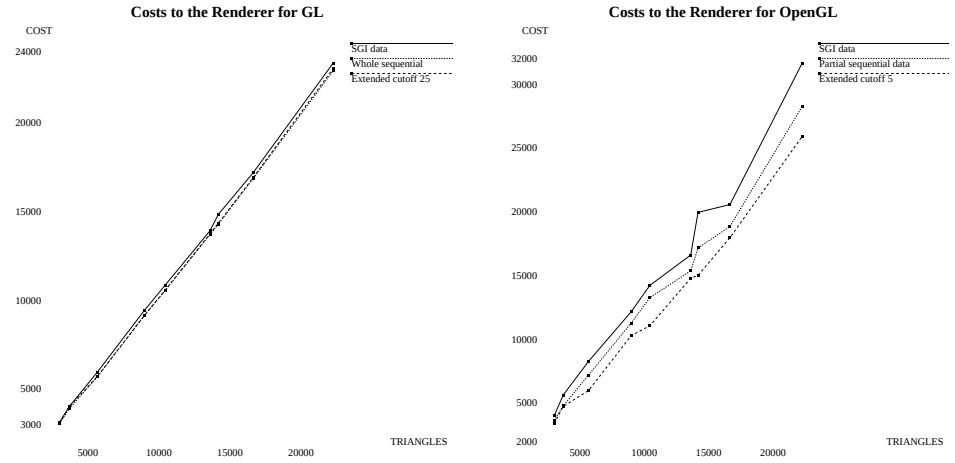


Figure 16: Overall cost comparisons for GL and OpenGL cost models.

- Little if any savings seems possible by sophisticated algorithms under the GL model. However, under the more realistic model the combined local/global algorithm can save up to 15% over the SGI algorithm.
- Our results are close to the theoretical lower bound of the number of triangles + 2, so there is limited potential for better algorithms.
- Although the number of strips and number of swaps required is sensitive to the composition of the model, the total cost grows linear in the size of the model.

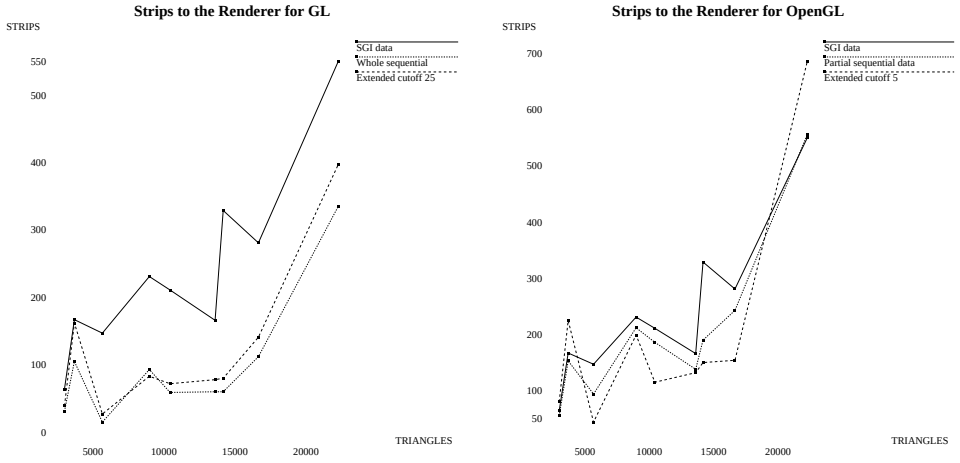


Figure 17: Number of strips produced for GL and OpenGL cost models.

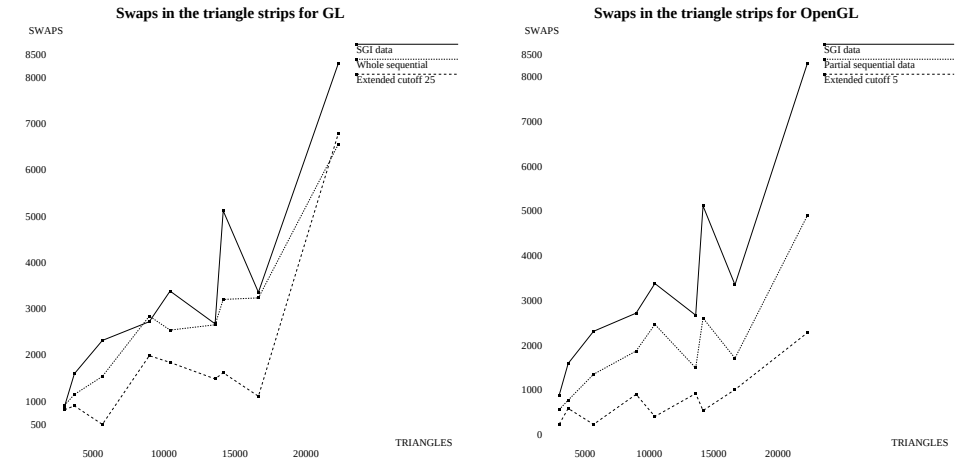


Figure 18: Total swaps produced for GL and OpenGL cost models.

Our times for execution of these algorithms behaved linearly with respect to the input size. *Stripe* Version 2.1 [13] converts the 69K triangles bunny model into triangle strips in 6 seconds on an SGI Onyx 2. When rendering the models with the triangle strips that were produced by each algorithm, the savings in transmission time to the renderer did prove to be a significant savings in rendering time. The triangle strips produced by our code were on average 30% faster to draw than those produced by the SGI algorithm, and were about 60% faster to draw than without using triangle strips at all. These savings increased as the size of the model increased, as shown in Table 2.

Machine	File	Triangles	Raw Triangles	SGI Tri-strips	Stripe
SGI Indigo2	Triceratops	5660	0.50	0.3	0.27
	Bell Ranger	14168	1.62	0.8	0.59
	General	22262	2.52	1.2	0.88
PC- 150MHz	Triceratops	5660	0.86	0.6	0.56
	Bell Ranger	14168	1.93	1.2	1.00
	General	22262	3.13	2.4	1.89

Table 2: Comparison of rendering times in seconds.

For local algorithms under the GL cost model whole-face triangulations worked better than those with partial-face triangulations; under the OpenGL cost model the reverse was true. Partial-face triangulations produce less swaps than whole-face triangulations because the former have a greater choice in selecting the next face in a strip, and are therefore more likely to be able to select faces that do not require a swap. For global algorithms, full-patch strips with cutoff size of 25 have the best performance under the GL cost model whereas full-patch strips with a cutoff size of 5 have the best performance under the OpenGL cost model. This is because a cutoff size of 5 generates more patches than a cutoff size of 25 and more patches means lesser number of swaps. Figure 19 provides visual comparison of the results obtained by our tool Stripe and those obtained by the earlier algorithm being used by SGI.

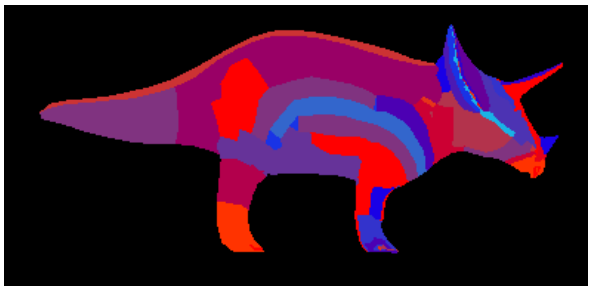
6.2 Impact of Buffer Size

The benefits realized by using triangle strips could be further enhanced by special-purpose hardware that has additional buffer space (beyond the usual storage for two vertices) and alternate queuing disciplines. In this section, we study the impact of such resources on performance, to provide guidance for future hardware design. Increasing the buffer size from a capacity of two vertices naturally decreases the cost of transmission, since we can now specify which of the previous k vertices in the buffer defines the next triangle. The cost of specification becomes $\lceil \lg k \rceil$ bits, instead of number of bits representing one vertex, thus enabling us to potentially represent polygonal models at a cost of less than one vertex per triangle. We ignore the costs of these index bits, since we only seek to determine an upper bound potential improvement in rendering time to assess whether it might be worth the increase in hardware costs. We considered two different queuing disciplines for maintaining the buffer:

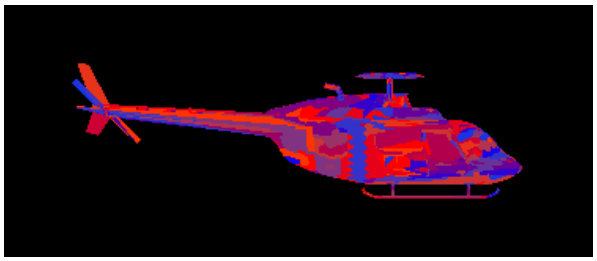
- *First-in, first-out (FIFO)* – This implies that there is no rearrangement of the vertices in the buffer, excluding swaps. FIFO is easiest to implement in hardware, and would thus be preferable if performance is comparable.

- *Least recently used (LRU)* – LRU dynamically rearranges the vertices in the buffer, by placing a vertex that was used most recently into the spot in the buffer that holds the most recently admitted vertex. The least recently used vertex is replaced by the new vertex. LRU provides the benefit that popular vertices are held in the buffer in the hope that they will likely be re-used.

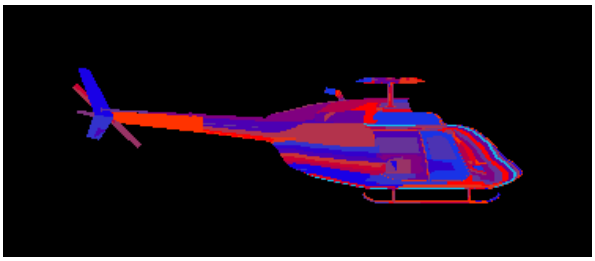
SGI



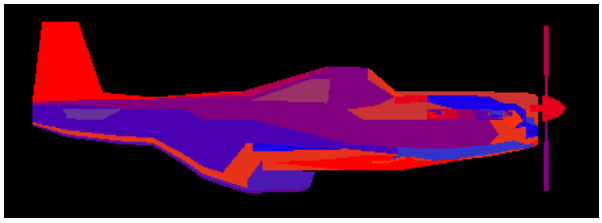
Stripe



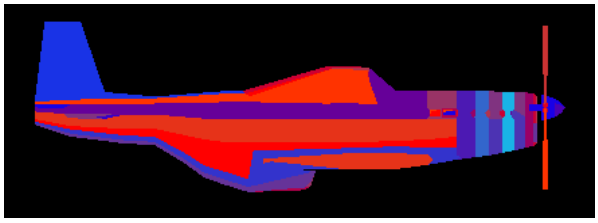
SGI



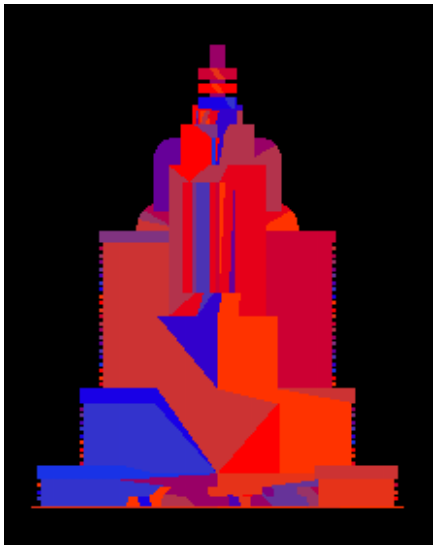
Stripe



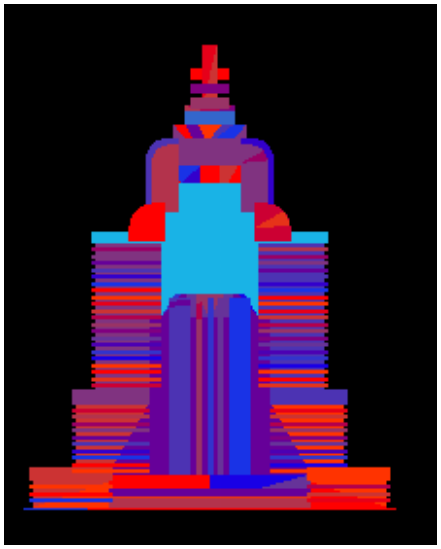
SGI



Stripe



SGI



Stripe

Figure 19: Visual Comparison of Triangle Strips Generated by SGI and Stripe

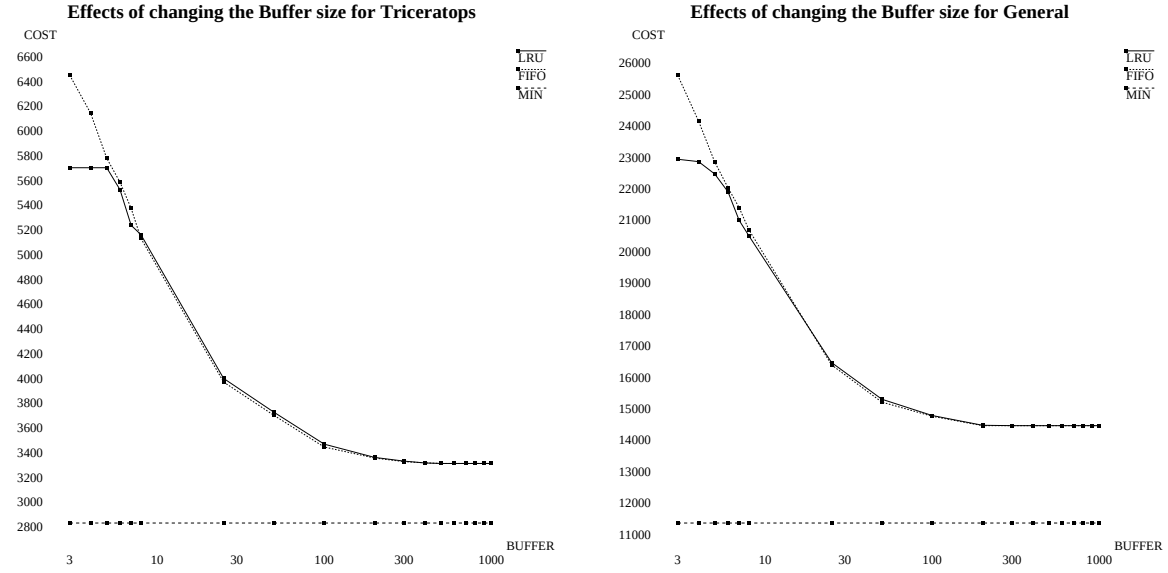


Figure 20: Cost versus buffer size for two representative models.

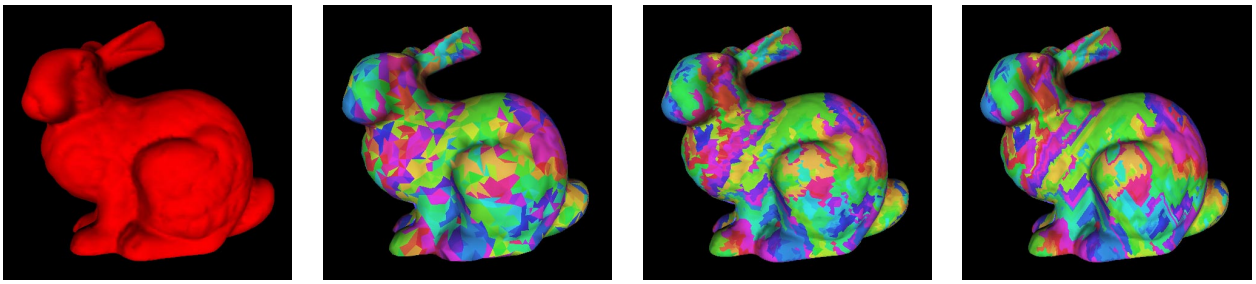
We did our tests on several datasets. Two representative results are are presented in Figure 20. These figures show the cost of the LRU and FIFO queuing disciplines (measured in number of vertices transmitted) versus the dataset sizes (represented by the MIN curve). As can be seen the advantages to be gained from larger buffer sizes diminish rapidly beyond a buffer size of about 32. For this range of buffer sizes, LRU performs better than the FIFO scheme by about 10%.

6.3 Skip Strips

We have implemented Skip Strips and have obtained the results shown in Tables 3 and 4. All of these results have been obtained on an SGI Onyx 2 with four R10000 processors, 1 GB RAM. Timings reported here do not assume parallelization. Table 3 shows the comparison between rendering datasets using triangle representation as in the conventional view-dependent rendering and rendering using Skip Strips. As can be seen the number of vertices sent is substantially less for Skip Strips since they incrementally maintain triangle strips that are used for final rendering.

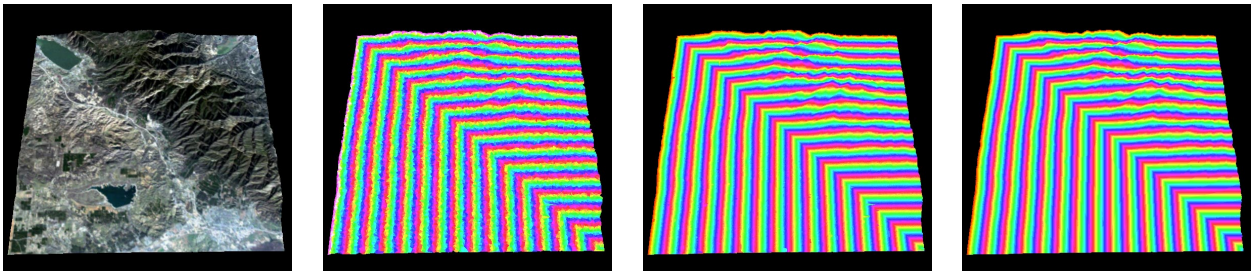
Table 4 shows the advantage of using Skip Strips over recomputing triangle strips at every frame. As can be seen from this table, recomputing triangle strips results in fewer vertices being sent as compared to those using Skip Strips. However, the total cost of recomputing triangle strips along with the time for their display far exceeds the cost of maintaining, updating, and displaying triangle strips using Skip Strips. All times reported below are wall clock times (not CPU times).

The datasets used for the above results appear in Figures 21, 22, and 23. In these figures, parts (a) show an intermediate level of view-dependent simplification, while parts (b), (c), and (d) show how the triangle strips are maintained across different levels of detail using Skip Strips. Colors in parts (a) depict object colors whereas colors in parts (b), (c), and (d) denote different triangle strips.



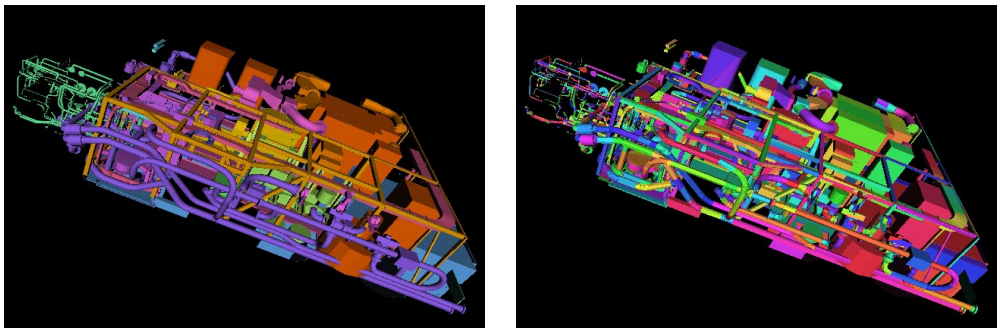
(a) 30K triangles (b) 5K triangles (c) 30K triangles (d) 65K triangles

Figure 21: Skip strips across varying resolutions for the Stanford Bunny model

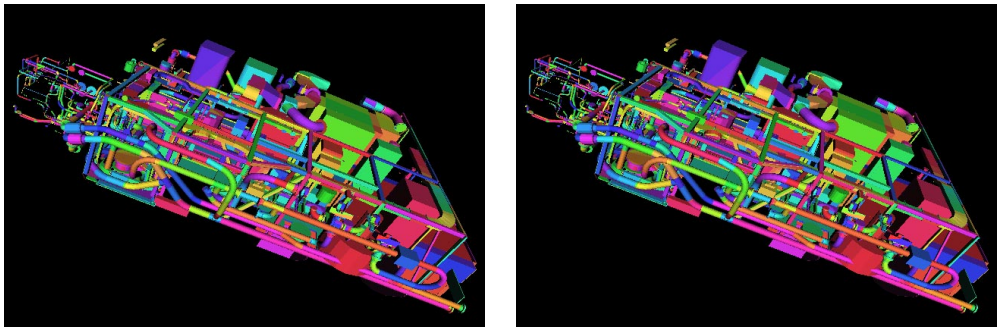


(a) 255K triangles (b) 32K triangles (c) 255K triangles (d) 522K triangles

Figure 22: Skip strips across varying resolutions for the Terrain dataset



(a) 170K triangles (b) 65K triangles



(c) 170K triangles (d) 340K triangles

Figure 23: Skip strips across varying resolutions for the Auxilliary Machine Room dataset

Dataset	Frame Tris	Triangles		Skip Strips	
		Display (msec)	Verts Sent	Display (msec)	Verts Sent
Bunny	10.2K	27.1	30.6K	16.0	22K
	50.8K	120.1	152.4K	78.1	83K
	69.4K	166.1	208.2K	101.0	92K
Terrain	71.4K	129.0	214.2K	103.0	160K
	255.5K	651.2	766.5K	401.2	480K
	522.2K	1573.5	1566.6K	991.3	550K
AMR	90.4K	134.1	271.2K	103.1	181K
	180.6K	385.4	541.8K	213.7	320K
	340.2K	573.1	1020.6K	297.4	491K
Dragon	107.8K	256.2	323.4K	204.1	205K
	445.5K	850.3	1336.5K	611.2	800K
	871.3K	1908.2	2613.9K	1184.5	1280K

Table 3: Comparison between Raw Triangles and Skip Strips for Rendering

Dataset	Triangles in Frame	Strips			Skip Strips		
		Construction (msec)	Display (msec)	Vertices Sent	Triangle Strip Updates (msec)	Display (msec)	Vertices Sent
Bunny	5K	310	4.6	8K	5.3	6.4	10K
	30K	2120	23.1	35K	13.1	30.3	42K
	65K	5310	84.6	90K	25.2	85.2	90K
Terrain	32K	2190	54.3	60K	15.2	78.4	90K
	255K	24630	370.5	411K	150.3	401.2	480K
	522K	131070	450.3	550K	230.0	460.3	550K
AMR	65K	4100	72.4	80K	32.3	101.3	130K
	170K	14300	173.2	268K	60.8	201.4	300K
	340K	35810	291.5	491K	101.2	297.4	491K

Table 4: Computing Triangle Strips per frame versus Skip Strips

7 Conclusions

We have explored a total of twenty different local and global algorithms on over two hundred data models in our quest for an effective triangle strip generation algorithm that can perform well under the prevalent OpenGL cost model. Our conclusion is that the best approach for the OpenGL cost model is global row or column strips with a patch cutoff size of 5. Source code for Stripe 2.1 is freely available for non-commercial use from `http://www.cs.sunysb.edu/$\sim$stripe.html`. As can be seen from the results of Table 1, we are able to outperform the SGI algorithm significantly. We typically produce a significantly lower number of strips than they do (usually 60%-80% less using the local whole-triangulation algorithm), resulting in an average cost savings of about 15% less than SGI algorithm under the OpenGL model. Further, our cost averages just 10% more than the theoretical minimum of using one sequential strip with no swaps, when using the global row or column strips algorithm with a patch cutoff size of 5, as shown in Figure 16. We have found that using global algorithms for detecting large strips of quads proves very effective for reducing swaps. This has proved to be quite useful for generating efficient triangle strips for the OpenGL

cost model where every swap costs one vertex.

All our algorithms run in linear time. Although the SGI algorithm does have a slightly better running time, we do not believe this to be a serious drawback of our approach since the triangle-strip generation phase is typically done off-line before interactive visualization. Also, our algorithm can take as input a polygonal model, while the SGI algorithm cannot handle polygonal data. Therefore to use their algorithm, the user needs to first pre-triangulate the data model, which is an extra step not added into the SGI running time.

The results of our experiments with larger buffer sizes offer only limited room for optimism. As we increase the buffer-size the savings do increase, however the improvements diminish very quickly. LRU seems to work much better than FIFO in the smaller buffers, although this must be contrasted with the time and hardware needed to maintain a LRU buffer. As indicated by our results, making a choice of a small buffer size, say around 32 seems attractive.

We have shown how Skip Strips can provide a convenient and simple representation to integrate retained-mode data-structures such as triangle strips with immediate-mode view-dependent simplifications. The Skip Strips offer two main advantages. First, they make pointer hopping along parent links in any hierarchical vertex collapse scheme efficient. Second, they simplify the execution of the vertex split and edge collapse operations to be as simple as two integer increment or decrement operations.

Skip Strips provide the advantage of hardware-assisted acceleration to view-dependent simplifications. However, they also suffer from some of the same limitations that afflict triangle strips. Thus, Skip Strip performance will not be very good for datasets that have several discontinuities in surfaces (cracks, holes, T-junctions), normals, colors, and textures. For such datasets the triangle strips that are generated have to be split across such surface attribute discontinuities thereby limiting their efficacy in succinctly representing the polygonal mesh. Although this does affect overall performance, the results will likely still be better than rendering raw triangles.

Another issue to consider is the performance of Skip Strips over genus-reducing simplifications. Our preliminary results indicate that Skip Strips are also applicable to view-dependent genus-reducing simplifications; we need to test this further.

Acknowledgements

This work has been supported in part by the NSF grants: CCR-9502239, CCR-9625669, DMI-9800690, ACR-9812572, ONR Awards: 400x116yip01, N00149710589, and a DURIP award N00014970362. Jihad El-Sana has been supported in part by the Fulbright/Israeli Arab Scholarship Program and the Catacosinos Fellowship for Excellence in Computer Science. Figure 23 shows the Auxiliary Machine Room part from the dataset of a notional submarine provided to us by the Electric Boat Division of General Dynamics. We would like to thank the reviewers for their insightful comments which led to several improvements in the presentation of this paper.

References

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

[2] E. Arkin, M. Held, J. Mitchell, and S. Skiena. Hamiltonian triangulations for fast rendering. In *Second Annual European Symposium on Algorithms*, volume 855, pages 36–47. Springer-Verlag Lecture Notes in Computer Science, 1994.

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

[4] R. Bar-Yehuda and C. Gotsman. Time/space tradeoffs for polygon mesh rendering. *ACM Transactions on Graphics*, 15, no. 2:141–152, 1996.

[5] P. Bhattacharya and A. Rosenfeld. Polygonal ribbons in two and three dimensions. Technical report, Department of Computer Science, University of Maryland, 1994.

[6] J. Bose and G. Toussaint. No quadrangulation is extremely odd. Technical Report 95-03, Department of Computer Science, University of British Columbia, 1995.

[7] M. Chow. Optimized geomerty compression for real-time rendering. In *IEEE Visualization '97 Proceedings*, pages 403 – 410. ACM/SIGGRAPH Press, October 1997.

[8] L. De Floriani, P. Magillo, and E. Puppo. Efficient implementation of multi-triangulation. In H. Rushmeier D. Elbert and H. Hagen, editors, *Proceedings Visualization '98*, pages 43–50, October 1998.

[9] M. F. Deering. Geometry compression. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 13–20. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[10] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, and C. Aldrich an M. Mineev-Weinstein. Roaming terrain: Real-time optimally adapting meshes. In *Proceedings of the IEEE Visualization '97*, pages 81 – 88. ACM/SIGGRAPH Press, October 1997.

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

[12] J. El-Sana and A. Varshney. Generalized view-dependent simplification. *Computer Graphics Forum*, 18, No. 6, 1999.

[13] F. Evans, E. Azanli, S. Skiena, and A. Varshney. Stripe Version 2.0, http://www.cs.sunysb.edu/∼stripe.

[14] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In *IEEE Visualization '96 Proceedings*, pages 319 – 326. ACM/SIGGRAPH Press, October 1996.

[15] F. Evans, S. Skiena, and A. Varshney. Efficiently generating triangle strips for fast rendering. Technical report, Department of Computer Science, State University of New York at Stonyt Brook, Stony Brook, NY 11794-4400, USA, March 1997.

[16] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of SIGGRAPH 93 (Anaheim, California, August 1–6, 1993)*, Computer Graphics Proceedings, Annual Conference Series, pages 247–254. ACM SIGGRAPH, August 1993.

[17] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH '97 (Los Angeles, CA)*, Computer Graphics Proceedings, Annual Conference Series, pages 209 – 216. ACM SIGGRAPH, ACM Press, August 1997.

[18] M. H. Gross, R. Gatti, and O. Staadt. Fast multiresolution surface meshing. In G. M. Nielson and D. Silver, editors, *IEEE Visualization '95 Proceedings*, pages 135–142, 1995.

[19] A. Gueziec, F. Lazarus, G. Taubin, and W. Horn. Surface partitions for progressive transmission and display, and dynamic simplification of polygonal surfaces. In S. N. Spencer, editor, *Proceedings VRML 98: third Symposium on the Virtual Reality Modeling Language, Monterey, California, February 16–19, 1998*, pages 25–32, New York, NY, USA, 1998. ACM Press.

[20] S. Gumhold and W. Straßer. Real time compression of triangle mesh connectivity. In *SIGGRAPH 98 Conference proceedings*, Annual Conference Series, pages 133–140. ACM SIGGRAPH, 1998.

[21] M. Held. Efficient and reliable triangulation of polygons. In *Proceedings of Computer Graphics International*, pages 633–643, 1998.

[22] H. Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, Computer Graphics Proceedings, Annual Conference Series, pages 99 – 108. ACM SIGGRAPH, ACM Press, August 1996.

[23] H. Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of SIGGRAPH '97 (Los Angeles, CA)*, Computer Graphics Proceedings, Annual Conference Series, pages 189 – 197. ACM SIGGRAPH, ACM Press, August 1997.

[24] H. Hoppe. Optimization of mesh locality for transparent vertex caching. In *Proceedings of SIGGRAPH '99 (Los Angeles, CA, August 8–13, 1999)*, Computer Graphics Proceedings, Annual Conference Series, pages 269 – 276. ACM Siggraph, ACM Press, August 1999.

[25] P. Lindstrom and G. Turk. Fast and memory efficient polygonal simplification. In D. Ebert, H. Rushmeier, and H. Hagen, editors, *Proceedings Visualization '98*, pages 279–286, October 1998.

[26] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hughes, Nick Faust, and Gregory Turner. Real-Time, continuous level of detail rendering of height fields. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 109–118. ACM SIGGRAPH, Addison Wesley, August 1996.

[27] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of SIGGRAPH '97 (Los Angeles, CA)*, Computer Graphics Proceedings, Annual Conference Series, pages 198 – 208. ACM SIGGRAPH, ACM Press, August 1997.

[28] T. Mitra and T. Chiueh. A breadth-first approach to efficient mesh traversal. In *1998 Eurographics/Siggraph Workshop on Graphics Hardware*, pages 31–38, 1998.

[29] G. Narasimhan. On Hamiltonian triangulations in simple polygons. In *Proceedings of the Fifth MSI-Stony Brook Workshop on Computational Geometry*, page 15, October 1995.

[30] A. Narkhede and D. Manocha. Fast polygon triangulation based on seidel's algorithm. *Graphics Gems 5*, pages 394–397, 1995.

[31] Open GL Architecture Review Board. *OpenGL Reference Manual*. Addison-Wesley Publishing Company, Reading, MA, 1993.

[32] Open GL Architecture Review Board, J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison-Wesley Publishing Company, Reading, MA, 1993.

[33] W. Pugh. Skip lists: A probabilitics alternative to balanced trees. *Communications of the ACM*, 33(6):668–678, 1990.

[34] J. Rossignac. Edgebreaker: Connnectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1), January 1999.

[35] B. Speckmann and J. Snoeyink. Easy triangle strips for tin terrain models. In *Proceedings of the Canadian Conference on Computational Geometry*, pages 239 – 244, 1997.

[36] R. E. Tarjan. Data structures and network algorithms. In *Regional Conference Series in Applied Mathematics*, volume 44 of *CBMS-NFS*. SIAM, 1983.

[37] G. Taubin, A. Guéziec, W. Horn, and F. Lazarus. Progressive forest split compression. In *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 123–132. ACM SIGGRAPH, 1998.

[38] L. Velho, L. de Figueiredo, and J. Gomes. Hierarchical generalized triangle strips. *The Visual Computer*, 15(1):21 – 35, 1999.

[39] J. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3, No. 2:171 – 183, June 1997.

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