Skip Strips: Maintaining Triangle Strips for View-dependent Rendering

Jihad El-Sana[1,2] Elvir Azanli[2] Amitabh Varshney[2]

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

[2] Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400

Abstract

View-dependent simplification has emerged as a powerful tool for graphics acceleration in visualization of complex environments. However, view-dependent simplification techniques have not been able to take full advantage of the underlying graphics hardware. Specifically, triangle strips are a widely used hardware-supported mechanism to compactly represent and efficiently render static triangle meshes. 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.

View-dependent simplifications have been recently 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 fail to take advantage of hardware-supported mechanisms for graphics acceleration, such as triangle strips. Luebke and Erikson [15] 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 [12] has proposed a solution to compute triangle strips per frame for the view-dependent simplification specific to that frame. In this paper we introduce Skip Strips as a solution to this dichotomy of immediate-mode simplifications and retained-mode hardware-supported acceleration.

A Skip Strip stores the vertex hierarchy nodes in a skip-list-like manner with path compression. Our approach combines the advantages of the two methods – selection of varied level of detail at different regions of the surface from view-dependent simplification and faster rendering from triangle strip representations. In addition, Skip Strips perform edge collapse and vertex split in constant time per operation, and the test to prevent foldovers at run time is done much faster as a result of using compact dependency lists. As other view-dependent simplification approaches, Skip Strips also take advantage of coherence between frames and incrementally update the displayed triangle strips. By using triangle strips, our algorithm is able to display the same number of triangles faster and uses less memory to store the active set of triangles.

2 Previous Work

In this section we give an overview of previous work done in the areas of view-dependent simplifications, triangle strip generation, and path compression data-structures.

2.1 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* [8] and Lindstrom *et al* [13]. 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 1. 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. We refer to vertex p as the parent of 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 [11].

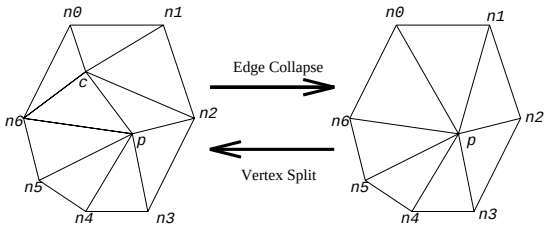

Figure 1: Edge collapse and vertex split

View-dependent simplifications using the edge-collapse/vertex-split primitives include work by Xia *et al* [20], Hoppe [12], Guéziec *et al* [9], and El-Sana and Varshney [4]. View-dependent simplifications by Luebke and Erikson [15], and De Floriani *et al* [3] 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.

2.2 Triangle Strips

Triangle strips provide a compact representation of triangular meshes and are supported by several graphics APIs including OpenGL. Triangle strips enable fast rendering and transmission of triangular meshes. An example triangle strip in the model of a cow is shown in Figure 2. The set of triangles shown in Figure 3(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 saving 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 3(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 3(b) can be represented as $(1, 2, 3, 4, 5, 4, 6, 7)$.

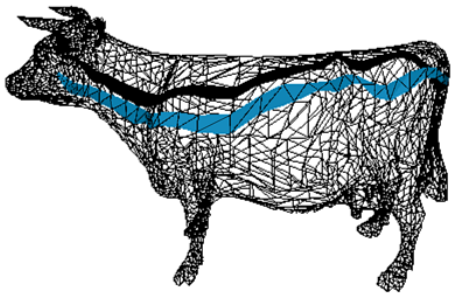

Figure 2: A triangle strip in a cow model

Akeley *et al* [1] have developed a program that constructs

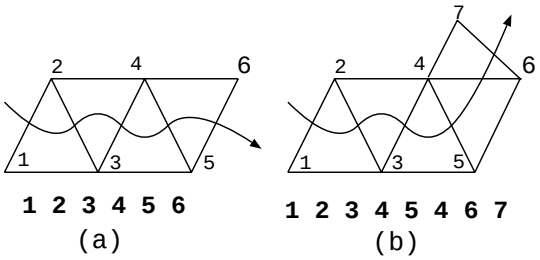

Figure 3: A triangle strip example

generalized triangle strips for a given triangle mesh model [1]. The algorithm tries to generate strips which minimize the number of one-triangle strips. This algorithm chooses the triangle which is adjacent to the least number of neighbors as the next triangle in a strip. Evans *et al* [6] use global adjacency information in conjunction with several heuristics such as maximizing the length of each strip, minimizing swaps, and minimizing the number of single-triangle strips. Speckmann and Snoeyink [17] have computed the triangle strips for triangulated irregular networks by creating a spanning tree of the dual graph, and then traversing the tree in a modified depth-first fashion. Chow [2], Taubin *at al* [19], and Gumhold and Straßer [10] have used strips to efficiently compress polygonal meshes.

2.3 Efficient Link Traversal

Let us study what happens when an edge collapses in a triangle strip. Figure 4 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, $c = 4$ and $p = 2$.

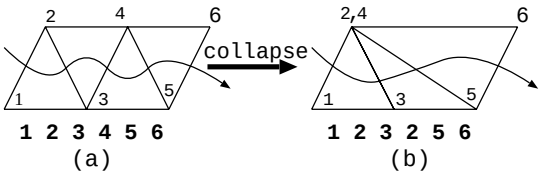

Figure 4: Edge Collapse in a Triangle Strip

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.

Skip list [16] 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 5(a). Reaching the n^{th} node on this list requires O(n) pointer hops. Consider next a data-structure that has $n/2$ additional pointers that connect linked list nodes that are 2 away (refer Figure 5(b)). Using these additional pointers, any node on the list can be accessed in O($n/2$) time. Skip lists generate O(n) such additional pointers in a probabilistic manner to provide O($\log n$) time access in the average case (refer Figure 5(c)).

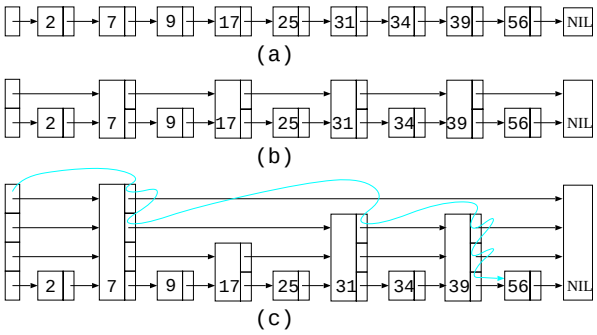

Figure 5: A Skip list example

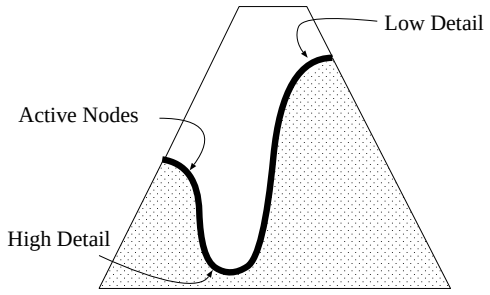

Figure 6: Varying detail in a Merge Tree

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 5(c). 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 to represent the set of pointers that are changed to implement the insert or delete operation.

3 Technical Background

In this paper we build upon two previous algorithms – construction of vertex hierarchy for view-dependent simplifications [20, 12] and construction of efficient triangle strips [6]. Let us overview these two algorithms next.

3.1 Construction of Merge Trees

Merge trees have been introduced by Xia *et al* [20] as a data-structure built upon progressive meshes [11] to enable real-time view-dependent rendering of an object. As discussed earlier, let the vertex p in Figure 1 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 $l + 1$.

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 backfacing regions) and screen-space projection. This is shown in Figure 6. 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* [20] 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 1, 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.

3.2 Generating Triangle Strips

We use the *Stripe* program by Evans *et al* [6] to generate high quality triangle strips. This approach considers the problem of constructing good triangle strips from polygonal models. Often such models are not fully triangulated, and contain quadrilaterals and other non-triangular faces, which must be triangulated prior to rendering. The choice of triangulation can significantly impact the cost of the resulting strips. Evans *et al* have experimented with several variants of local and global algorithms; the details are available in [6]. After comparing the results from 20 different local and global approaches on over 200 datasets, the best option has been empirically observed to use the global row or column strips with a patch cutoff size of 5. In this approach the model is first partitioned into regions that have collections of $m \times n$ quadrilaterals arranged in m rows and n columns, which is referred to as a *patch*. Each patch whose number of quadrilaterals, mn, is greater than a specified cutoff, in this case 5, is converted into one strip at a cost of three swaps per turn. Further, every such strip is extended backwards from the starting quadrilateral and forwards from the ending quadrilateral of the patch to the extent possible. On triangulated models like the ones we consider in this paper, *Stripe* has been found to work as well as other public-domain triangle strip converters. *Stripe* Version 2.0 [5] converts the 69K triangles bunny model into triangle strips with a total of 90K vertices in 6 seconds on an SGI Onyx2.

Stripe generates efficient triangle strips but requires more time than simplistic methods such as the greedy method [12]. Since we wanted to do comparisons with an on-line algorithm to convert polygonal meshes into triangle strips we also decided to implement the greedy method. The greedy method proceeds as follows. From a given triangle we extend a triangle strip as far as possible. Once it is no longer possible to extend the triangle strip, we stop and begin a new triangle strip. In our current implementation of the greedy method we

are working only with sequential triangle strips as discussed in Section 2.2. We found that for an on-line method, the greedy method is a better choice than *Stripe* since the former takes much less time, even though it generates about $15 - 20\%$ more vertices. The greedy method takes 0.2 seconds on an SGI Onyx2 to convert a 69K triangle bunny model to triangle strips and generates 96K vertices.

4 Our Approach

In our approach we generate a merge tree and the triangle strip representation of the original polygonal model off-line. The merge tree file, which contains the parent-child relationships for each node of the tree, is constructed as overviewed in Section 3.1 and described in [20]. 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. The triangle strip representation is generated using the *Stripe* program as overviewed above in Section 3.2 and described in [6]. At run-time we load the merge tree and 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.

4.1 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. This can be seen in Figure 7 where the parent pointer is shown on the right and the list of child pointers is shown on the left of each Skip Strip node. We shall see in Section 4.3 how to generalize this data-structure to support a list of parent pointers to accelerate access in an edge-collapse hierarchy.

A Skip Strip is constructed at run time from the merge tree and triangle strip representations. A Skip Strip node is allocated for every merge tree leaf (terminal node) and 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 average height of the merge tree is $O(\log n)$. Thus, the average length of this child-pointer list for a Skip Strip node is $O(\log n)$. 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. 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*.

To illustrate the Skip Strip data-structure, let us see how it is built from a merge tree. Figure 8(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 is the parent node (as defined by Figure 1). Let us assume that we are dealing with edge collapses in which one vertex collapses to another (i.e. no new vertices are created). The equivalent Skip Strip data-structure will have four nodes

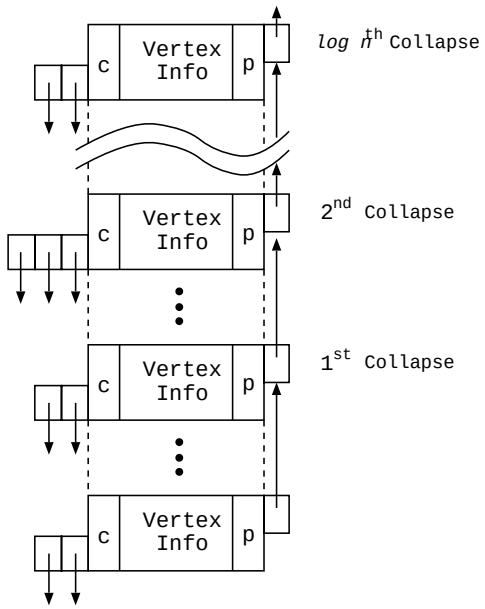

Figure 7: A Skip Strip node

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 in Figure 8(b). The edge collapse $3 \rightarrow 1$ 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 8(c).

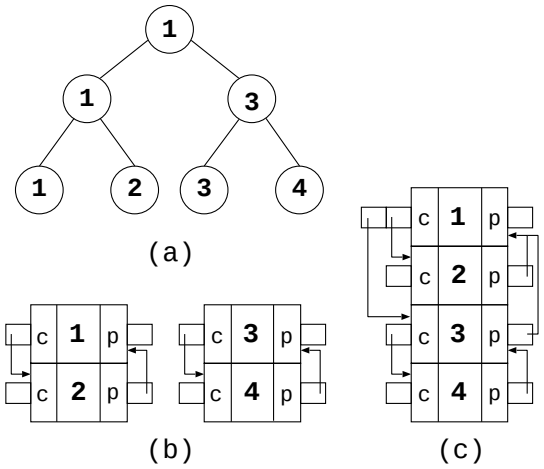

Figure 8: Building a Simple Skip Strip

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 [11] or for

better placement of approximating vertices using sophisticated error metrics [7, 14, 4]. 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.

4.2 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 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*.

4.2.1 Determination of display vertices

Determination of display vertices proceeds along the same lines as proposed in earlier work on view-dependent simplification [20, 12] 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 3.1. One way to avoid such artifacts is to use dependencies [13, 20]. In [4], 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 [20]. 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.

4.2.2 Determination of display strips

The graphics dataset is represented as a set of triangle strips. Each triangle strip has two representations – the original highest resolution triangle strip that was generated using preprocessing, 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 4.2.1. 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 whether 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 9 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 11 shows the same after two edge collapses ($6 \rightarrow 5$, and $8 \rightarrow 7$) to the mesh of Figure 9. In Figure 10 none of the parent pointers is active (since there have been no edge collapses). Figure 10 shows the merge tree and the skip strip with one parent pointer per node, constructed for the mesh in Figure 9 at the highest detail. In Figure 11, the parent pointers for nodes 6 and 8 pointing to 5 and 7 respectively, are active and appear dot shaded. The nodes 6 and 8 are inactive and appear with hatched shading.

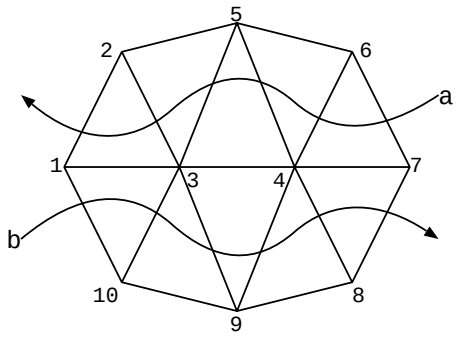

Figure 9: Original triangle mesh

4.3 Efficient Skipping for Parent pointers

As the object moves to a coarse representation, the time spent in following the active parent pointers increases. On the average, the maximum number of active parent pointers that one might need to traverse is $O(\log n)$ – the height of the vertex

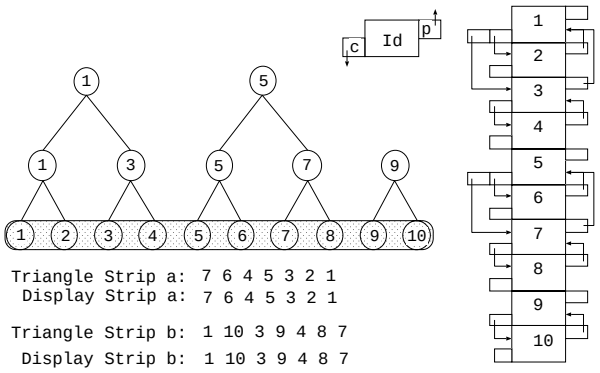

Figure 10: Skip Strip for Triangle Mesh in Figure 9

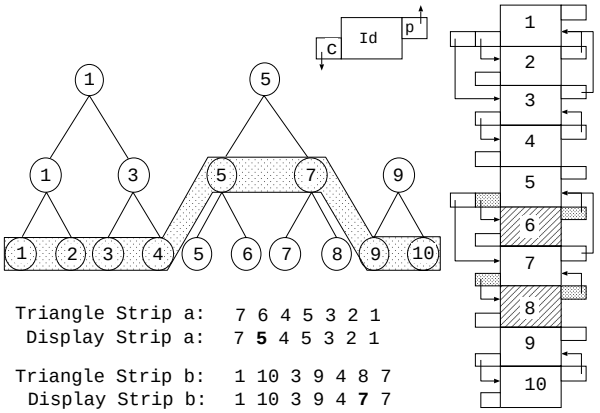

Figure 11: Skip Strip for Triangle Mesh in Figure 9 after two edge collapses

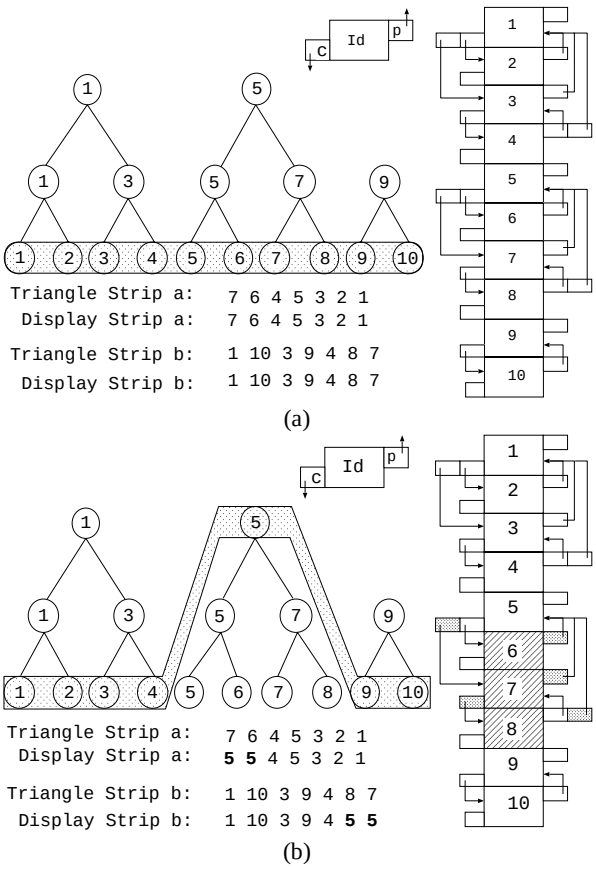

Figure 12: More efficient Skip Strip representations for Figures 10 and 11

hierarchy. To reduce this time we trade off memory for speed. To accomplish this we use ideas from path compression [18] and skip lists [16] to build a list of parent pointers for each Skip Strip node. The parent pointers of each node point to its ancestors that are $1, 2, 4, \ldots, \log n$ hops 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 $O(\log n)$ to $O(\log \log n)$ 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 $5/4$ [20]. Thus, even if the edge-collapse-based vertex hierarchy tree is balanced (which it 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.

To efficiently implement traversal of parent pointers, 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 until 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. Likewise, when a vertex j splits 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 note 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 12 shows the Skip Strip representation with multiple parent pointers for each node for the mesh in Figure 9. Note that the active parent and child pointers appear shaded.

4.4 Further Optimizations

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 imple-

mented a simple triangle strip scanner that detects and replaces patterns of vertices of the regular expression form $(aa)^+$ by (aa) and $(ab)^+$ by (ab) in the sequence of vertices sent for rendering.

Figure 13 shows the relationship between the triangle strip *a* (top half of the mesh from Figure 9) and how the display strip relates to it. As can be seen, a display strip is simply a linked list of pointers to the triangle strip. At the beginning of each frame the display strip is updated from the triangle strip. As the underlying mesh is simplified and vertex repetitions (as detected by triangle strip filtering) increase, it pays to do two further optimizations: (a) skip over the repetitions, and (b) change the display strip incrementally from frame to frame instead of constructing it from the original triangle strip per frame. The first optimization can be easily accomplished by using a skip-list-like structure instead of a linked list for the triangle strip (refer Figure 13(b)). The second optimization is accomplished by storing two pointers with each collapsible edge. These pointers point to the two triangle strips to which the two triangle sharing that edge belong. Since the triangle strips are computed statically, these pointers are generated only once during the pre-processing stage. For non-manifold meshes that can have more than two triangles sharing an edge, one can accordingly store one triangle strip pointer per additional triangle. Whenever an edge collapses at run-time, the (at most) two triangle strips that are affected have their headers flagged as *modified*. When a display strip is considered for rendering, we first check to see if its corresponding triangle strip has been modified since the last frame. If it has, we update the display strip, otherwise use it as is.

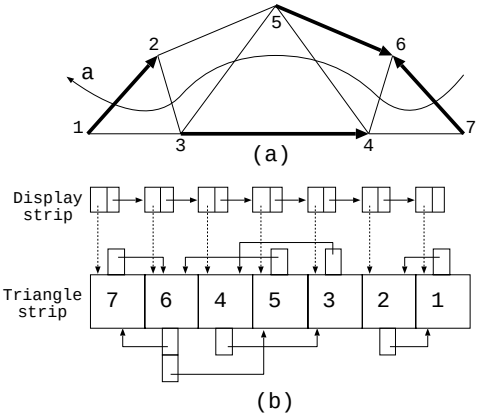

Figure 13: (a) Arrows mark edge collapses. (b) Efficient local skipping over triangle strip.

We note again that the Skip Strips are computed only once, at the pre-processing stage. Therefore, as the simplification increases there comes a stage when triangle strips computed from a Skip Strip representation are highly fragmented (they each represent a small number of triangles). To address this issue, we have added a stage in our current implementation that performs merging of display strips. This proceeds as follows. We check the triangle strip pointers on the last edge of a display strip to see if a new display strip is beginning at that edge. If it is, we extend the current strip effectively merging the two strips.

5 Results

We have implemented Skip Strips and have obtained the results shown in Table 1 and Figure 14. 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 of the view-dependent simplifications.

Table 1 shows the comparison between rendering datasets using three modes of view-dependent renderings. The three modes differ in how triangles are sent for rendering. Identical parameters of view-dependent simplifications are used across the three modes resulting in identical sets of triangles rendered. In the first mode triangles are sent independently without taking advantage of any adjacency information. In the second mode triangles determined for display in one frame are converted into triangle strips using the greedy method for generating sequential triangle strips. This is the current state-of-the-art method for using triangle strips with view-dependent simplifications. The third mode involves using Skip Strips to generate display strips for rendering triangles. The comparisons for the three modes are shown in Table 1 for four datasets across representative flythroughs (as shown in the video). The *Frame Count* row indicates the number of frames in the flythrough path. The *Adapt Count* row indicates the total number of edge collapse/vertex split operations performed for the given flythrough path. The *Tris Count* and *Verts* rows represent the total number of triangles and vertices sent for rendering, respectively, over the entire flythrough path. Within each mode *Adapt*, *Display*, and *Total* indicate the cumulative times spent over the flythrough paths in changing the view-dependent detail, rendering, and the total time, respectively. In online stripping *Strip* is the time to generate triangle strips whereas for Skip-Strips, the time to maintain the strips is part of the *Display* time. As can be seen from these numbers, Skip Strips result in a $35\% - 95\%$ improvement over sending raw triangles and $50\% - 63\%$ improvement over computing triangle strips on-the-fly from scratch.

As the simplification levels increase and mesh sizes reduce, it becomes more attractive to perform on-the-fly greedy triangle strip computation than to maintain Skip Strips since the fragmentation amongst triangle strips increases as mentioned in Section 4.4. Merging of triangle strips on the fly addresses this problem to a certain extent, but it is inevitable that at some stage of simplification it becomes less attractive to maintain Skip Strips. Figure 14 shows our results in determining the threshold above which we found it better to use Skip Strips for the bunny model. We found similar performance curves for other datasets. Rather than clutter the graph with several curves, we have simply reported the crossover points for the other datasets on the same graph.

The datasets used for the above results appear in Figures 15, 16, and 17. 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.

6 Conclusions

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-

Dataset		Bunny	Buddha	AMR	Terrain
Frame count		215	152	101	150
Adapt Count		61.5K	55.2K	65.8K	83.5K
Tris count		9.6M	10.5M	6.5M	12.5M
Send Triangles	Verts	28.8M	31.5M	19.5M	37.5M
	Adapt	7.4s	7.1s	8.1s	10.5s
	Display	36.5s	41.6s	22.8s	48.6s
	Total	43.9s	48.7s	30.9s	59.1s
Online Stripping	Verts	15.9M	16.8M	18.0M	14.3M
	Adapt	10.5s	9.8s	11.7s	14.5s
	Strip	12.5s	13.2s	9.1s	16.3s
	Display	20.1s	22.2s	16.4s	18.5s
	Total	43.1s	45.2s	37.2s	49.3s
Skip-Strips	Verts	17.3M	18.8M	16.1M	16.5M
	Adapt	3.1s	2.6s	3.2s	4.7s
	Display	24.1s	26.1s	19.6s	26.3s
	Total	27.2s	28.7s	22.8s	31.0s

Table 1: Performance of view-dependent triangle, triangle strips on the-fly, and Skip Strips

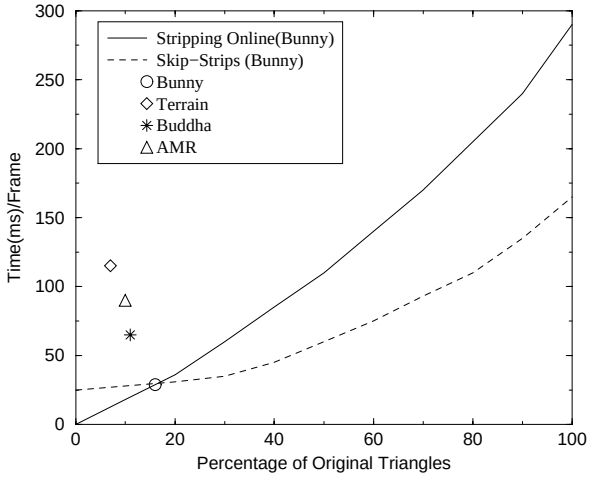

Figure 14: Skip-Strips versus stripping online

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 such as cracks, 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, DMI-9800690, ACR-9812572 and a DURIP instrumentation 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 17 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] M. Chow. Optimized geomerty compression for real-time rendering. In *IEEE Visualization '97 Proceedings*, pages 403 – 410. ACM/SIGGRAPH Press, October 1997.

[3] L. De Floriani, P. Magillo, and E. Puppo. Efficient implementation of multi-triangulation. In *Proceedings Visualization '98*, pages 43–50, October 1998.

[4] J. El-Sana and A. Varshney. Generalized view-dependent simplification. In *Eurographics '99 (to appear)*, Milano, Italy, 1999.

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

[6] 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.

[7] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH '97*, pages 209 – 216, August 1997.

[8] 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.

[9] A. Guéziec, G. Taubin, B. Horn, and F. Lazarus. A framework for streaming geometry in VRML. *IEEE Computer Graphics and Applications*, 19(2):68–78, 1999.

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

[11] H. Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH '96*, pages 99 – 108. ACM SIGGRAPH, ACM Press, August 1996.

[12] 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.

[13] P. Lindstrom, D. Koller, W. Ribarsky, L. Hughes, N. Faust, and G. Turner. Real-Time, continuous level of detail rendering of height fields. In *SIGGRAPH 96 Conference Proceedings*, pages 109–118. ACM SIGGRAPH, 1996.

[14] 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.

[15] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of SIGGRAPH '97*, pages 198 – 208. ACM Press, August 1997.

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

[17] B. Speckmann and J. Snoeyink. Easy triangle for TIN terrain models. In *Canadian Conference on Computational Geomerty 97*, pages 239–244, 1997.

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

[19] 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.

[20] 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*, pages 171 – 183, June 1997.