Controllable Single-Strip Generation for Triangulated Surfaces

M. Gopi
Department of Computer Science,
University of California, Irvine
gopi@ics.uci.edu

Abstract

In this paper we introduce a method to represent a given triangular model using a single triangle strip. Since this problem is NP-complete, we break the limitation by splitting adjacent triangles when necessary. The common edge is split at the mid-point, and the newly formed triangles are coplanar with their parent triangles. Hence the resulting geometry of the model is visually and topologically identical to the original triangular model. Our method can develop any edge-connected oriented 2-manifold of arbitrary topology, with or without boundary, into a single strip. Our stripification method can be controlled to start and end at triangles incident on specific vertices. Further, an acyclic set of edges of the input model can be marked as "constraint edges" and our method can generate a single strip that does not cross over these edges, but still cover the whole model.
Keywords:*Triangulation, Stripification, Hamiltonian paths and cycles, Path planning, Constrained path planning, Fundamental cycles.*

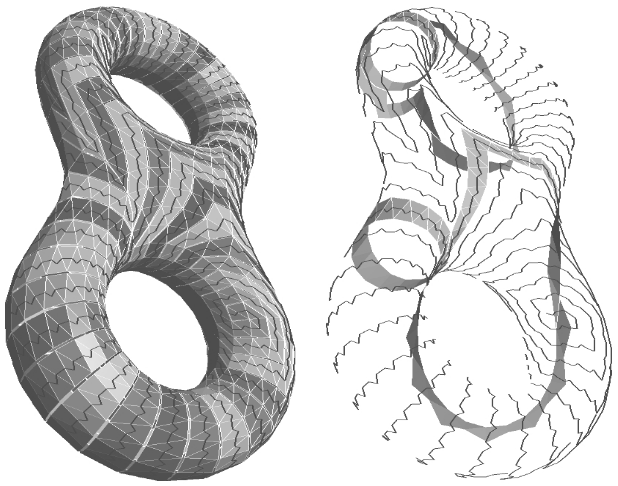
Figure 1. We can find a single strip, and using that fundamental cycles of surfaces with arbitrary topology and boundaries. Left: A double torus traversed in a single strip. Right: Its four fundamental cycles.

1. Introduction

Constructing strips from the input set of triangles has been an active field of research in computer graphics and computational geometry, primarily motivated by the need for efficient rendering in the former and by traveling salesman and Hamiltonian path problems in the latter. Traditionally, triangle stripification research has been pursued along two extreme problem statements. At one end, the input triangulation is considered to be unchangeable and algorithms have been designed to analyze if there is a Hamiltonian path in the triangulation or to get as few number of strips as possible from the given triangulation (for example, [6, 4]). At the other end, the input triangulation is completely ignored, and a new triangulation is imposed on the input set of vertices in order to arrive at a single strip triangulation even if it requires addition of new vertices (for example, [10, 2]). The work presented in this paper tries to bridge the gap by combining the strengths of both extremes. It finds a single strip triangulation from the input triangulation by splitting the input triangles if necessary. At the same time, the geometry of the input model is also retained. Our motivation goes beyond the rendering requirement and theoretical aspects of Hamiltonian path. The advantages of having a single triangle strip representation of a model enables a plethora of other geometric and topological algorithms to be applied on the model. In this paper, we show a by-product of our algorithm to compute fundamental cycles using discrete steepest gradient method on the single strip on manifolds of arbitrary topology using total linear ordering of the triangulation.

Main Contributions: The following are the main contributions of the paper.

- We present a method to convert an edge-connected triangulated model into a single edge-connected triangle strip, without any preprocessing.
- We present a variation of the above method to control the stripification process to avoid crossing over

given acyclic edge sequences. This has applications in constrained path planning problems, in which edge sequences are given as constraints that robots cannot pass through.

- We extend our algorithm to control the stripification process by forcing the strip to start at a triangle incident on a given start vertex, cover the entire model, and end at another triangle incident on a given destination vertex.
- We present a method to find fundamental cycles of the given model using the single strip representation.

In the following sections we briefly describe the related work in this field (Section 2), explain our algorithm for single strip generation (Section 3) and finding fundamental cycles (Section 4). We also improve on our stripification algorithm to include constraints based on edges and source-destinations (Section 5).

2. Related Work

There has been a phenomenal interest in the stripification algorithms trying to find longer and longer triangle strips from the given model, primarily motivated by the need for efficient rendering. These stripification algorithms can be categorized based on their input requirements. The first category of algorithms takes only the vertices of the model as input and finds a triangulation that would generate a single strip (e.g. [2]). The second category takes edges of a polygon as input and triangulates the interior of the polygon, with or without the addition of Steiner vertices, to create triangle strip(s) that covers the polygon (e.g. [10]). Typically, these two categories work only with data sets on a plane or a height field. The third category takes triangles of the model as input and tries to build long triangle strips, not necessarily a single strip, only using the input set of triangles (e.g. [6]). The third category works with 2D surfaces embedded in 3D.

The algorithm presented in this paper is a combination of all the above three categories and create a single triangle strip from the model. In the triangle strip generated by our method, all the input vertices are used as in the first category, Steiner vertices and hence more triangles are added as in the second category, and finally, the geometry of the input triangulation is retained as in the third category.

Dillencourt [5] showed that finding Hamiltonian cycles in Delaunay triangulation is NP-complete. Since finding a single strip is equivalent to finding a Hamiltonian path, conceding to the hardness of the problem, many algorithms just try to find as few a strips as possible from the input triangulation. A program that SGI developed [1] produces generalized triangle strips. Its heuristic tends to create strips that begin and end on faces with fewest number of neighbors in the triangulation, so as to reduce the number of isolated triangles. The classic STRIPE algorithm [6] makes a global analysis of the input triangle mesh, trying to find patches that can be efficiently striped. Velho et al. [17] build and maintain their triangle strips incrementally while creating the triangle mesh simultaneously. [4] builds strips by reusing the points added to previous strips as often as possible. Snoeyink and Speckmann [14] propose a stripification algorithm specially designed for triangulated irregular network (TIN) models using the spanning trees of the dual graphs of TIN models. [18] decompose spanning trees of the dual graph into triangle strips. Tunneling method for triangle strips in continuous level of detail meshes is proposed by [13]. Taking this a step further, [12] propose dynamic triangle strip management for view-dependent mesh simplification and rendering algorithms. Hoppe [8], and Bogomjakov and Gotsman [3] investigate the ordering of triangles in order to reduce the number of vertex cache misses and develop methods to find triangle sequences that preserves the property of locality. Triangle strips are also important in geometric compression and transmission and is a by-product of these algorithms [11, 16]. Here again, the input triangulation is usually not modified.
The hardness of the problem of finding a Hamiltonian path can be eased with minor variations of the problem statement. For example, algorithms presented in [2] avoid Delaunay triangulation of the planar point set and create a Hamiltonian path triangulation. Further, they also take a (planar) simple polygon, check using visibility graph if there exists a single strip triangulation of the interior of the polygon. If not, such a triangulation is produced using Steiner vertices. They also prove that computing a Hamiltonian triangulation for planar polygons with holes is NP-hard. The QuadTIN method [10] triangulates an irregular terrain point data set by adding Steiner vertices at the quadtree corners to produce a dynamic view-dependent terrain triangulation that can be traversed as a single strip. Given a quadrilateral mesh of a manifold, Taubin [15] splits each quadrilateral into triangles and orders them into a single strip. Gopi and Eppstein [7] construct single strip from a triangulated manifold by adding new vertices and triangles using perfect matching graph algorithm. But this algorithm works for only manifold surfaces without boundaries and the stripification is not controllable. Unlike the above methods that take points on a plane or a height field or a quadragulation as input, our method uses the triangulation, not just the point set, of manifolds of arbitrary topology including the ones with boundaries. Further, even by adding new triangles, our algorithm does not change the input geometry (in terms of visual fidelity); whereas the above methods prescribe a completely new triangulation that includes the input point set. Our stripification is completely controllable

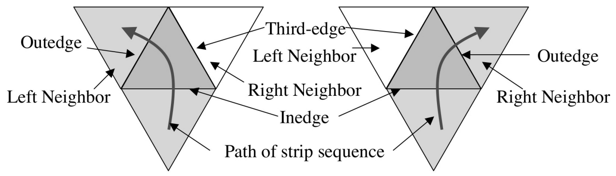

Figure 2. Left turn and right turn.

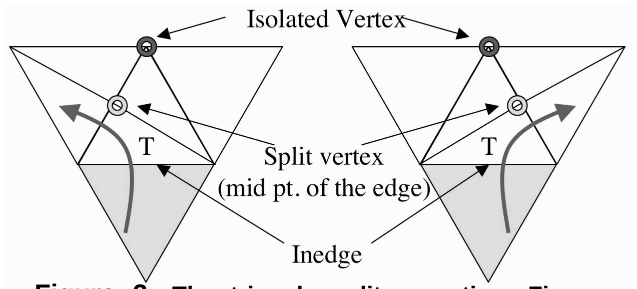

Figure 3. The triangle split operation. Figure shows left and right splits. A split operation is used to isolate a vertex. To ensure such isolation, the strip grows in the same direction as the split direction.

and can follow the prescribed direction that satisfies certain conditions.

3. Single Strip

In this section we describe a few terms we use in this paper followed by a detailed description of our algorithm to create a single strip out of the input model.

3.1. Definitions

Triangle Strip: A triangle strip is a linear sequence of *unique* triangles each of which share a common edge with its neighbor in the sequence. During rendering, after issuing the first three vertices to render the first triangle of the strip, rendering of every subsequent triangle in a specific orientation requires just one additional vertex information. “Swap” commands are used to change this orientation. A sequential triangle strip is one that does not require a “swap” operation during the traversal of vertices for rendering; non-sequential strips might need “swap” operations. In this paper, the triangle strips we generate might be non-sequential.

Adjacent triangle: We consider two triangles to be adjacent if they share a common edge, not just a common vertex.

Striped triangle: A *striped* triangle is one that has already been included in the triangle strip; otherwise it is *unstriped*.

Boundary edge: Since we consider only manifolds with or without boundary, every edge has either one or two triangles incident on it. A boundary edge is one that has only one incident triangle, or has two incident triangles out of which one is striped and the other is unstriped.

Boundary triangle: A boundary triangle is an *unstriped* triangle with at least one boundary edge.

Third vertex of an edge: The vertex that is opposite to an edge of a triangle is called the third vertex of that edge.

Right and left turns: *Inedge* of a triangle in a triangle strip sequence is the edge shared by the previous neighbor in the sequence; *outedge* is the one that is shared by the next neighbor. In the counter-clockwise ordering of edges with respect to the normal of the triangle, if the *outedge* is the next edge to *inedge* then the triangle strip is said to have taken a *right turn*; otherwise a *left turn* (refer Figure 2). The corresponding adjacent triangles are called right neighbor and left neighbor. The edge of a triangle that is neither its inedge nor its outedge is referred to as the *third edge*.

Triangle-Split Operation: The fundamental and the only geometric operation we do is a *triangle-split*. Given two adjacent triangles, we split these triangles by connecting the third vertices of the common edge to the mid-point of the common edge (refer Figure 3). The rest of the stripification algorithm is about when the triangle-split operation is used.

Split-candidate: An *unstriped* triangle is a candidate for splitting if exactly one of its edges is a boundary edge and the third vertex of the boundary edge is also in a boundary (refer Figure 4). If a triangle is a split-candidate then a decision to split it or not is made based on the following classification of split-candidates.

Bridge triangle: (Figure 4) A split-candidate T is said to be a bridge triangle if any path through the unstriped triangles from the left neighbor of T to the right neighbor has to pass through T. In other words, removal of T splits the unstriped region into two disconnected islands and hence T forms a bridge between these islands. Bridge triangles, when encountered while strip-growth, have to be split. Not splitting a bridge triangle would leave a portion of the model unstriped.

Fundamental triangle: If a split-candidate is not a bridge triangle, it is called a fundamental triangle. In other words, from the left neighbor of T to its right neighbor, there exists a path, which we call a *parallel path*, that does not pass through T (refer Figure 4). We call these triangles as fundamental triangles since they are the seed triangle to trace fundamental cycles of the model; and the *parallel path* is homotopic to the fundamental cycle of the model. Presence of a fundamental cycle implies presence of fundamental triangles; although, the converse need not be true. Fundamental triangles, when considered to be included in the strip, should *not* be split. Consistently splitting fundamental triangles would lead to infinitely long triangle strips. More details are given in the following sections.

3.2. Algorithm for Single Strip Representation

Here we describe the basic algorithm for single strip generation, followed by the cases in which it will not work and the

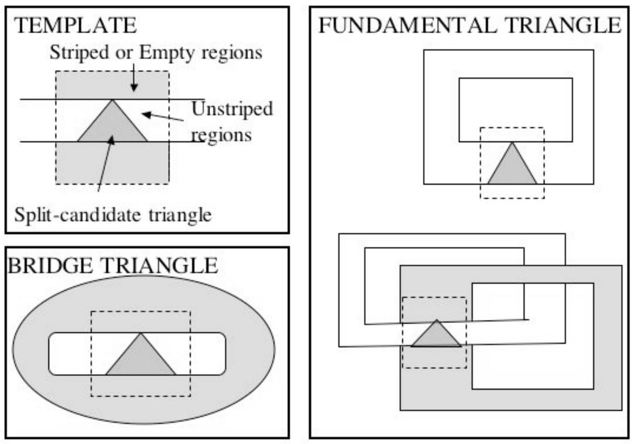

Figure 4. Top Left: The template of the local neighborhood of a split-candidate triangle: only one edge and its third vertex are on the boundary. Split candidate is further classified into bridge-triangle (bottom-left) and fundamental-triangle (right) depending on whether the unstriped region (white) is connected around the triangle or not. Notice the identical local neighborhood (denoted by dashed lines) but different global topologies of bridge and fundamental triangles. The unstriped (white) path connecting either side of the fundamental triangle is the *parallel path*. Notice that the triangle connecting the boundaries of an one annulus (top-right) is a fundamental triangle.

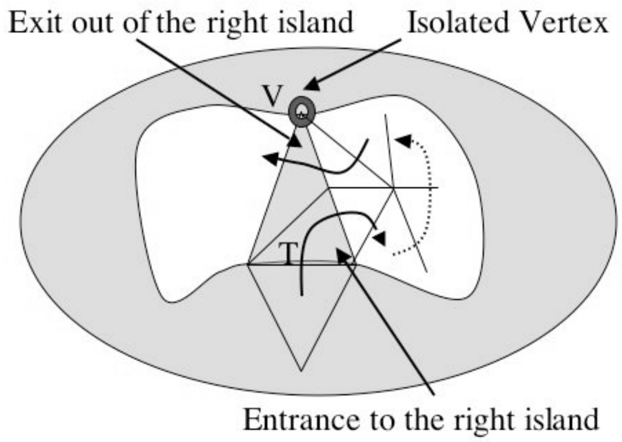

Figure 5. T **is a bridge triangle and is split to isolate the third vertex** V **which is also on a boundary.** T **forms the bridge between two islands of unstriped regions. After the split, one of the children of T acts as an entry to the island and the other acts as an exit. The stripification grows in the same direction of the split to effectively isolate** V**.**

amendments to this basic algorithm to handle these cases.

Basic algorithm: The basic stripification algorithm chooses the first triangle and collects a non-empty and unstriped adjacent neighbor as the next triangle in the sequence. We have two choices for the next triangle in the sequence – right neighbor and left neighbor. If triangle i in the strip sequence is the right neighbor of triangle $i-1$, then the right neighbor of i is chosen as the next triangle $(i+1)$ of the sequence; otherwise left neighbor is chosen. If the edge in the direction of the strip is a boundary edge then the direction is reversed. This process continues till all triangles are striped.

Handling islands: Refer to Figure 5. Consider a bridge triangle T. Striping T might create two islands out of which only one island can be possibly traversed by the strip. Triangle split operation is applied on T and one of its right or left adjacent neighbors to *isolate* V. The triangle split operation creates two paths, one to enter and the other to exit an island once the stripification of that island is complete. The stripification continues in the same direction as the split adjacent neighbor of T. Note that going in the opposite direction would create infinite triangle split operations.

Handling non-zero genus models: There are situations when a split-candidate T need not create islands. Such split-candidates, as defined earlier, are called fundamental triangles. Fundamental triangles exist under one of the two cases – first, if the model is not a genus zero model; second, if the model is a manifold with boundaries and the three vertices of a triangle belong to two or more different boundary loops as shown in bottom-left of Figure 4. If T is a fundamental triangle then it is included in the strip without any split and the stripification continues in the same direction as before. If the fundamental triangle T is split and the strip follows the *parallel path* to reach the other side of T, this will trigger another split of T eventually leading to infinite number of splits.

The actual pseudo-code of our algorithm is presented in Algorithm 1. This algorithm is guaranteed to generate single strip surfaces. There are no other special cases to handle other than that is described above. In Algorithm 1, the *third edge* of a triangle is the edge that is neither an inedge or an outedge (refer to Section 3.1).

Greedy turns: Every split adds two new triangles. If while growing the strip, the right neighbor requires a split and the left neighbor does not require a split, or vice-versa, we can take a greedy turn to choose the appropriate neighbor and avoid the split. Such greedy turns have proved useful in handling triangulations where two islands of triangles are connected by a single strip of triangles. Greedy turns are used only to reduce the number of splits and not required for the

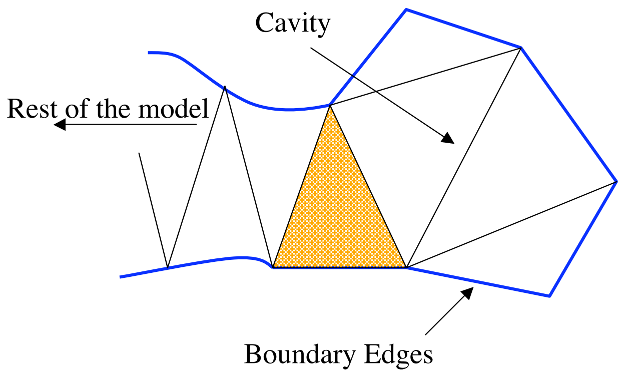

Figure 6. The split-candidate has to be classified as a bridge or fundamental triangle. If the search for the left neighbor starts at the right neighbor, the search would end immediately. On the other hand, if the search starts in the left neigbor, then the time taken is proportional to the complexity of the unstriped model region. So a local search for cavities is encouraged to improve the efficiency of the stripfication.

Algorithm 1 Generate Single Strip

```
Choose a seed triangle $T$.
T.SetStriped()
Set T.outedge to be a non-boundary edge.
Set default direction.
while there exists an unstriped triangle do
    if IsBoundaryEdge(T.outedge) then
        T.outedge = T.thirdedge;
    else
        GreedyTurnCheck(T) {Optional}
    end if
    if IsBridgeTriangle(T.neighbor(outedge) then
        splitNeighbor = T.neighbor(outedge).neighbor(outedge).
        Split(T, splitNeighbor)
    end if
    T = T.neighbor(outedge)
    T.SetStriped()
end while
```

correctness of the stripification algorithm. Pseudo-code of this algorithm is given in Algorithm 2.

Implementation insights: In the above algorithm, the implementation to check if the given triangle is a bridge triangle is the important step. We have implemented this using depth first search among only the unstriped triangles with the left neighbor as the root and the right neighbor as the search target. In certain straight forward cases, this search can be avoided by a local search for “cavities” formed by boundary edges (Figure 6).

Algorithm 2 GreedyTurnCheck(T)

```
if IsBridgeTriangle(T.neighbor(outedge)) and
not IsBoundaryEdge(T.thirdedge) and
not IsBridgeTriangle(T.neighbor(thirdedge)) then
    T.outedge = T.thirdedge;
end if
```

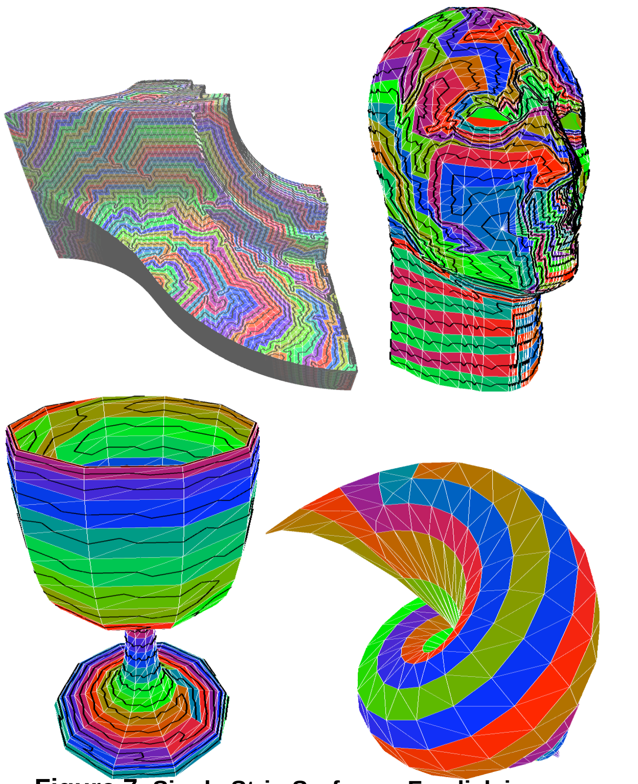

Figure 7. Single Strip Surfaces: Fandisk is genus 0 with 12946 input and 16630 output triangles; Face is a three boundary, genus 0 object with 2918 input and 3668 output triangles; Goblet is genus 0 with 1000 input and output triangles; Shell is a one boundary, genus 0 object with 1680 input and 1710 output triangles. The double torus used in Figure 1 has genus 2 with 1536 input and 1642 output triangles.

Though we do not take any clues about the topology of the object as input in our implementation, we suggest using such information to avoid the above search in the following cases:
(a) The search is not necessary for an object with genus zero, since all split-candidate triangles are bridge triangles. In other words, there are no fundamental triangles and hence no fundamental cycles for a genus zero object.
(b) This search is also not necessary for a mesh homeomorphic to a disk if the first seed triangle of the strip is a boundary triangle. On the other hand, if the seed triangle is an in-

Algorithm 3 InitiateFundCycle(Triangle T) { T is a fundamental triangle}

```
Add T to cycle C.
Let D be the adjacent neighbor of T with the largest index in the linear ordering.
Let S be the adjacent neighbor of T with the second largest index in the linear ordering. {S is the source of the fundamental path and D is the destination}
Add S to cycle C.
SteepestGradientAscent(S, D, C)
return C;
```

Algorithm 4 SteepestGradientAscent(Triangle S, Triangle D, Cycle C)

```
if S==D then
    return; {Reached the destination}
end if
Let M be the adjacent neighbor of S with the largest index but less than that of D's index.
Add M to C {Steepest gradient ascent to reach D}
SteepestGradientAscent(M, D, C)
return;
```

terior triangle, the edges of this triangle form a boundary edge cycle in addition to the original boundary cycle of the given model (a case similar to one-annulus in Figure 4 except that the interior of the annulus is "striped" instead of empty). This is one case when there might be fundamental triangles but need not translate to the presence of fundamental cycles.

3.3. Analysis of the algorithm

The algorithm presented in the paper has time complexity of $O(nlogn)$ average time complexity and $O(n^2)$ in the worst case. If the topology of the object is a sphere topology and is known apriori, then the complexity of the method can be brought down to $O(n)$.

A triangle is split only if it is a bridge triangle. Once a bridge triangle is split, none of the new triangles is a bridge triangle. Every split creates two new triangles. If the number of input triangles is n then at most n triangles can be bridge triangles and the number of new triangles is at most $2n$. This analysis assumes algorithm with the greedy approach in strip traversal.

Our implementation takes less than half a second for the fandisk model which has 12946 input triangles and 16630 output triangles, a 28% increase in the number of triangles. The genus two double torus model has a 6% increase in triangle count, genus one torus has 19% increase, and the genus zero head model with three boundaries has 25% increase in the triangle count (Figure 7). All these models have less than 4000 output triangles and took less than 0.3 seconds each. The timing measurements were taken on a 667MHz PowerPC G4 with 512MB memory. Though our motivation for generating a single strip is not just rendering, [12] have shown that anything less than 33% increase in the triangle count can be compensated by the improvement in the rendering performance achieved by a single strip.

4. Computing the Fundamental Cycles

An interesting application of our basic single stripification algorithm is computing the fundamental cycles. Fundamental cycles are used in finding the topology of the object. Fundamental cycles are generators of any closed loop on the surface. Hence an algebraic combination of these loops can be "morphed" without cutting the cycle to any closed loop on the surface. Finding fundamental cycles have many applications in graphics including texture synthesis on manifolds, conformal texture mapping, 3D morphing, geometry images, etc.

As mentioned in Section 3.1, the fundamental triangles that are identified during the stripification process are chosen as seed triangles to find the fundamental cycles. The only constraint that we impose on the stripification algorithm in order to find the correct fundamental triangles is to start the stripification with a boundary triangle, if any, as the seed triangle. The most important aspect of single strip representation is that there is a total linear ordering of triangles. We use this linear ordering to find the shortest path along the strip to get the fundamental cycle. We number all the triangles in the order of their appearance in the single strip representation. Any triangle in the strip will have the index one more than that of its adjacent triangle along the *inedge* and one less than that of its neighbor along its *outedge*.

By definition, for every fundamental triangle T there is a path from one of its side to the other not passing through T. Hence the single strip representation of the model consists of a strip segment from the *outedge* neighbor of T to the *third-edge* neighbor of T and this segment does not contain T. Along with T, this strip segment becomes a cycle, the fundamental cycle. We use the linear ordering of triangles to follow a steepest gradient ascent approach to reach the destination quickly. This cuts off unnecessary portions of the strip resulting in a "cleaner" fundamental cycle. The pseudo-code of this algorithm is given in Algorithm 3.

For a manifold with genus g and for a manifold with boundary derived from that, there are $2g$ and $2g+h-1$ fundamental cycles respectively, where h is the number of connected boundary loops on the manifold. For a double torus with genus two, there are four cycles and for the "head" model which is derived from a sphere ($g=0$) by creating $h=3$ boundaries, there are two fundamental cycles as shown in Figures 1 and 8.

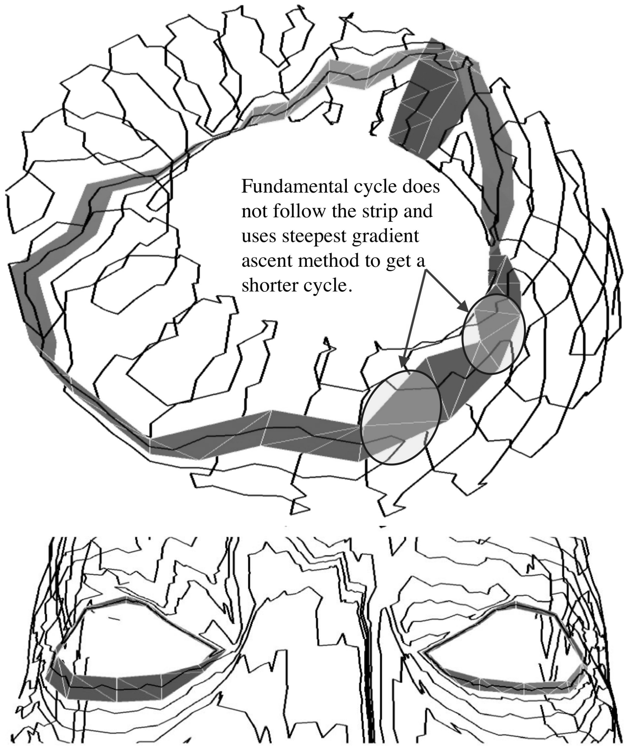
Fundamental cycle does not follow the strip and uses steepest gradient ascent method to get a shorter cycle.

Figure 8. **Fundamental cycles of torus (genus one) and face (genus 0 and three boundaries). They have two fundamental cycles each. Notice the effect of the steepest gradient ascent method by which fundamental path "jumps" across strip boundaries (highlighted in the torus) thus cutting off unnecessary strip triangles to find a shorter path to complete the loop.**

5. Constrained Single Stripification

The stripification algorithm presented in Section 3.2 can be modified to take into account different constraints imposed on the required strip. Here we describe two such constraints and appropriate changes that has to be done to the input and algorithm to handle these constraints.

5.1. Edge Constrained Stripification

In robotic path planning applications, a single strip representation of the space that the robot has to cover can be used to plan the path. Further, there might be obstacles in the space that the robots have to avoid. These obstacles can be represented as a two dimensional region or a one-dimensional edge sequence. If the obstacles are regions, these regions have to be cut out from the input space so that they are not traversed by the strip. This makes the input space manifolds with boundaries. Since our stripification algorithm can handle surfaces with boundaries, we can avoid two dimensional "prohibited" regions during stripification. However, in the case of one dimensional obsta-

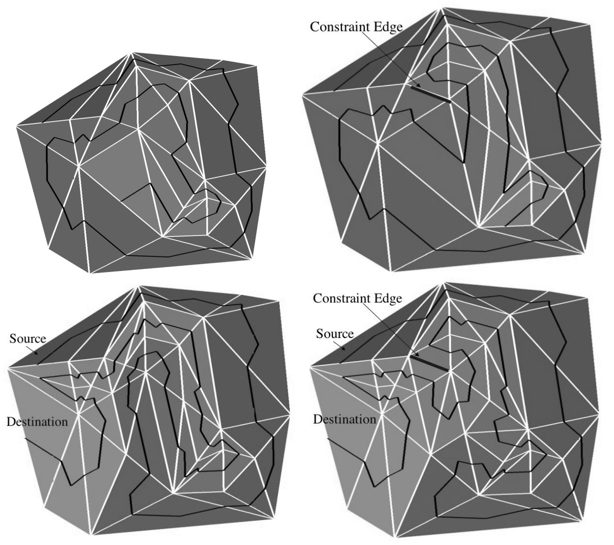

Figure 9. **Top Left: Single strip without the edge constraint. Top Right: Single strip of the same input mesh with the edge constraint (blue). The strip does not cross over this edge. Bottom Left: Single strip of the same input mesh with constraint on destination vertex. Bottom Right: Result with both edge and source-destination constraints.**

cles, we mark those edges as boundary edges and the triangles incident on them as non-adjacent triangles. This makes sure that the crossover of these marked edges by the strip is avoided. An example of the result of imposing such an edge constraint and the resulting stripification is shown in Figure 9.

5.2. Source-Destination Constrained Stripification

Given a source and a destination vertex, the strip can be controlled to start and end in triangles incident on the given vertices. Since we use the triangle split operation, in the course of strip generation, uniqueness of triangles is lost. So to avoid ambiguity, we define source and destination vertices rather than triangles. It is easy to start the strip at any vertex, and our stripification algorithm can take any seed triangle, including a triangle incident on the start vertex. But it is difficult to end the strip at a given vertex and appropriate modifications have to be done to our algorithm to accommodate this change. The idea is to carve out a path from the destination vertex and when the growing strip from the start vertex reaches the "mouth" of this path *after covering all the other triangles*, the strip grows into the path and finally reaches the destination vertex. In other words, the strip can enter the "mouth" of the path only if both the strip and the path have no other adjacent triangle to grow. Let us call the triangles that carve out the path from the destination as *path-triangles*. In every triangle along the path from the destination vertex, we maintain fields to point to the pre-

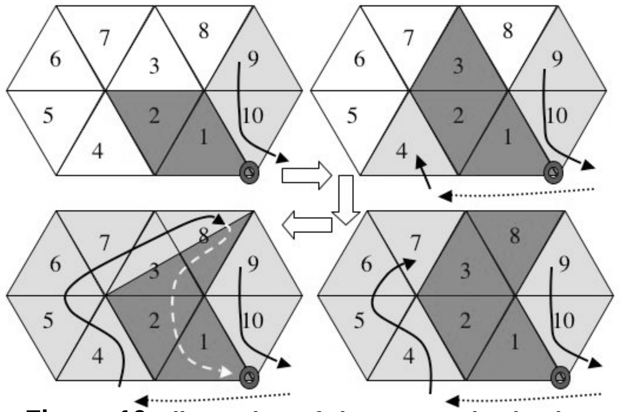

Figure 10. **Illustration of the source-destination constrained algorithm: Marked vertex is the chosen destination vertex. Triangles 1 and 2 are first path triangles. Strip starts from 9, goes to the rest of the mesh and stripes 4. This makes 2 a boundary triangle, and hence 3 becomes the path triangle. Continuing, when 3 becomes the boundary triangle 8 becomes a path triangle. Since 7 does not have any other triangle to grow the strip, 3 and 8 and split. Note 8, and not 2, is chosen for the split since 3's *to-triangle* is 8 and hence is farther from the destination. Once the strip reaches the last triangle 8 both the strip and the path do not have any more triangle to grow and hence the strip enters the path striping the triangles on its way to the destination.**

vious triangle in the path, the *from-triangle* field, and the next triangle in the path, the *to-triangle* field. The first triangle in the path will have a NULL *from-triangle* field and the last triangle will have a NULL *to-triangle* field. In the following description, we refer to the path from the destination as just *path*. Here we describe our algorithm to grow the path and the strip. We suggest referring to Figures 10 and 11 whenever necessary. Choose a triangle T_1 incident on the destination vertex that has an adjacent triangle T_2 that is not incident on the destination vertex. The triangles T_1 and T_2 are our first two *path-triangles* and they are assigned as the first and last triangle of the *path* respectively. When the *path* grows, a triangle T_{n+1}, adjacent to the last triangle T_n of the path, is added to the path if T_n is a boundary triangle and T_{n+1} is unstriped. In other words, T_{n+1} is the only way for the strip to reach T_n and hence, by induction, to the destination vertex. In Figure 10, Triangles 3 and 8 are added to the path when Triangles 2 and 3 become boundary triangles respectively. Thus, after the inclusion of first two triangles, the path from the destination vertex grows only when the strip from the source grows and comes in contact with the last triangle of the path and makes it a boundary triangle. The pseudo-code of the above propagation algorithm is given in Algorithm 5. Now we have to deal with the situation when a growing strip comes in contact with the *path-triangles*. A triangle strip grows when a new triangle is striped. Striping a new triangle might make its adjacent triangles, boundary triangles. If one of its adjacent neighbors is a *path-triangle*, then it would trigger propagation of path (Algorithm 5). In the next round of strip growing, the stripification algorithm has to always choose a non-path triangle to grow. But there is a possibility that the strip has no other triangle adjacent to the last striped triangle, but only its *path-triangle* neighbors, as in the case of Triangle 7 in Figure 10. If both the neighbors of the last triangle in the

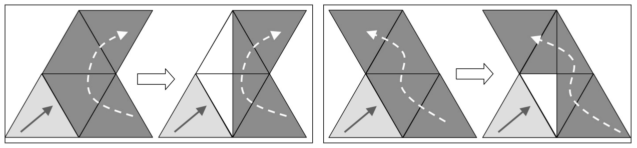

Figure 11. **Triangle on the bottom-left is a striped triangle and the dark triangles are in the path carved from the chosen destination. The dashed arrows show the direction away from the destination. When the strip has no where to grow, the adjacent path triangle and the path-triangle that is farther from the destination are split, and the path is re-routed. Two cases of path-rerouting is shown. The newly formed white triangles are unstriped triangles.**

Algorithm 5 PropagatePath(Triangle T)

```
if T is not a path-triangle then
  return
end if
if T is the last triangle of the path then
  if T is a boundary triangle then
    if there exists an unstriped neighbor $N$ not in the path then
      AddToPath(N) {This should also set to- and from-triangle fields of $T$ and $N$ appropriately}
      PropagatePath(N) {Call this function recursively}
    end if
  else
    return; { T is the last triangle, but not a boundary triangle}
  end if
else
  PropagatePath(T.to-triangle) {T is a path triangle, but not the last triangle}
end if
return;
```

strip are *path-triangles*, then the one closer to the last triangle of the path is chosen as a candidate for growing the strip.

There are two cases based on whether this chosen *path triangle* is the last triangle of the path or not. If this chosen path-triangle has no adjacent triangle to grow the *path* then it means that the strip has reached the “mouth” of the path, and the only direction to grow the strip is through this path to reach the destination vertex (refer to the point when the strip enters the path in Figure 10). If the adjacent triangle is not the last path-triangle, then the strip has to find the “mouth” of the path searching in the direction of to-triangle of this chosen path-triangle (refer to the point when Triangles 3 and 8 are split in Figure 10). In this case, we use the triangle split operation as shown in Figure 11, and we call this a *path-split* operation since it has to adjust the path after the split. Further, after a path-split operation, the last striped triangle will always have a non-path-triangle neighbor, leaving room for the strip to grow. The above algorithm continues till the strip reaches the last triangle and eventually the destination vertex. This algorithm is guaranteed to cover all the triangles in the model and reach the destination vertex. Results of this algorithm for different destination vertices are shown in Figure 9.

6. Conclusion

We have presented algorithms to develop a single triangle strip from the given triangulation by edge split operations when necessary. The resulting triangulation includes original vertices and a few vertices inserted in edge-midpoints during the split operation. Hence the geometry of the single strip surface is exactly the same as the original surface. Our algorithm can also be constrained such that the strip does not cross over the constraint edges and the strip starts and ends at triangles incident on chosen vertices.

Applications of single strip transcends beyond its use in efficient rendering. We showed one of the by-products of our algorithm, the fundamental cycle computation and it used the total linear ordering of all the triangles in a triangle strip of the model. Mesh compression of triangle strips [9] have proved to be more effective than traditional mesh compression techniques. Such techniques can be helped by single strip surfaces, in spite of modest increase in triangle count.

Our stripification is completely controllable to produce strips that satisfy certain properties. The need for a particular type of strip depends on the application. The only invariant that has to be maintained during strip growth process is that the number of boundary edge loops of the striped triangles, that is not a boundary loop of the given model, is exactly one at any time during the stripification process. Any striping pattern that respects this invariant can be achieved using this method.

References

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

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

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

[4] M. M. Chow. Optimized geometry compression for real-time rendering. In *Proc. IEEE Visualization*, pages 347–354, 1997.

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

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

[7] M. Gopi and D. Eppstein. Single strip triangulation of manifolds with arbitrary topology. In *Proc. of EUROGRAPHICS*, 2004.

[8] H. Hoppe. Optimization of mesh locality for transparent vertex caching. In *Proc. SIGGRAPH*, pages 269–276, 1999.

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

[10] R. Pajarola, M. Antonijuan, and R. Lario. QuadTIN: Quadtree based triangulated irregular networks. In *Proc. IEEE Visualization*, pages 395–402, 2002.

[11] J. Rossignac. Edgebreaker: Compressing the incidence graph of triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, January-March 1999.

[12] M. Shafae and R. Pajarola. DStrips: Dynamic triangle strips for real-time mesh simplification and rendering. In *Proc. Pacific Graphics*, pages 271–280, 2003.

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

[14] R. Szeliski and H.-Y. Shum. Creating full view panoramic image mosaics and environment maps. In *Proc. SIGGRAPH*, pages 251–258, 1997.

[15] G. Taubin. Constructing hamiltonian triangle strips on quadrilateral meshes. In *Int. Workshop on Visualization and Mathematics and IBM Research Tech. Rep. RC-22295.*, 2002.

[16] C. Touma and C. Gotsman. Triangle mesh compression. In *Proceedings Graphics Interface 98*, pages 26–34, 1998.

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

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