# An Improved Adjacency Data Structure for Fast Triangle Stripping

Patrick Reuter
LaBRI - CNRS - INRIA Futurs
University Bordeaux 1
France
preuter@labri.fr

Johannes Behr
Department of Visual Computing
Computer Graphics Center Darmstadt
Germany
jbehr@zgdv.de

Marc Alexa
Department of Computer Science
Darmstadt University of Technology
Germany
alexa@informatik.tu-darmstadt.de

September 18, 2004

**Abstract**

To speed up the rendering of polygonal meshes, triangle strips are commonly used to reduce the number of vertices sent to the graphics subsystem by exploiting the fact that adjacent triangles share an edge. In this paper, we present an improved adjacency data structure for fast triangle stripping algorithms. There are three major contributions: first, the data structure can be created quickly and robustly from any indexed face set; second, its cache-friendly layout is specifically designed to efficiently answer common stripping queries, such as neighbor finding and least-degree triangle finding, in constant time; third, the stripping algorithm operates in-place, since strips are created by simply relinking pointers. An implementation of a stripping algorithm shows a significant speed-up compared to other implementations. Our implementation is publicly available as part of OpenSG [9].

## 1 Motivation

Rendering triangular meshes requires the vertices of each triangle to be sent to the graphics subsystem. In order to minimize the necessary bandwidth, it is advantageous to order the triangles so that consecutive triangles share an edge. Triangle sequences of this type are called *strips*. Finding an optimal partition of a triangular mesh into triangle strips is an NP-hard problem [4]. However, several heuristic algorithms achieve partitions close to the theoretical bounds [5, 7, 8, 11].

Lately, the idea of strips has been generalized to a *vertex cache* [6]. Instead of storing only the last two vertices for the possible definition of a new triangle, the vertex cache stores more, which would theoretically allow each triangle to be specified with less than one vertex [2, 3]. However, triangle strips remain one of the most important rendering primitives.

Most practical file formats for meshes store faces individually; thus, a conversion to strips should occur when loading the mesh. Existing work on stripping concentrates on generating a good set of strips, that is, a set that minimizes the number of vertices sent to the graphics subsystem, and considers the generation to be an offline process. We have found that most algorithms have similar queries in the triangle adjacency graph and that a suitable data structure speeds up the generation significantly. Using this data structure, stripping can be done during loading, which we found more useful in practice than storing the strip information.

We present a simple extension of the half-edge data structure [10]. It is the result of understanding what operations are typically needed for stripping algorithms and minimizing their execution times. Furthermore, we present an implementation of a variant of the algorithm developed by Evans et al. [5], commonly known as STRIPE. Our implementation is publicly available as part of OpenSG [9].

## 2 An Alternative Half-edge Data Structure

Generating triangle strips requires the creation and traversal of a triangle adjacency graph followed by the processing of every triangle. Creating the triangle adjacency graph involves locating triangles that share an edges. This time-consuming operation should ideally be performed in constant time. To get good performance, a cache-friendly data structure that tries to avoid following pointers is desirable. During the triangle strip generation process, triangles have *states*.Before being processed, they have a degree which is the number of unprocessed, adjacent triangles. After being processed they are part of a fan, a strip, or the set of isolated triangles. Apart from traversing the triangle adjacency graph, the most frequent operations during stripping are changing the state of a triangle and finding arbitrary triangles with a given state.

The half-edge data structure [10] accommodates the requirement for quick traversal of the triangle adjacency graph. We have adapted this data structure to our needs by organizing triangles in doubly-linked lists reflecting states and by removing elements that are not needed for stripping. Consequently, a half-edge is represented as follows:

```
class HalfEdge {
  Triangle *triangle;
  HalfEdge *twin;
  HalfEdge *next;
  int vertexIndex;
  /* and some methods ... */
};
```

where `triangle` points to the triangle containing the edge, `next` points to the next half-edge of the same triangle, and `twin` points to the half-edge in reverse direction or is null at boundary edges. The start vertex of the half-edge (`vertexIndex`) and the end vertex of the half-edge (`next->vertexIndex`) are represented by indices, since they are not needed as topological elements for stripping.

A triangle is represented as follows:

```
class Triangle {
  HalfEdge halfEdgeVec[3];
  int state;
  Triangle *next;
  Triangle *prev;
  /* and some methods ... */
};
```

We store the three edges incident upon a triangle as a vector, instead of storing only one pointer to a single edge. Although an obvious choice, this storage scheme has two major advantages over existing implementations of the half-edge data structure. First, the access is faster on average as it requires dereferencing only once. Second, allocating memory for triangles and their edges at the same time aligns them nicely in memory, which minimizes cache misses for the most common access operations (especially `halfEdge->next`) and yields a significant speed-up.

The triangles themselves are stored in doubly-linked lists maintained with the `next` and `prev` pointers. Triangles with the same state are stored in the same list:

```
class TriangleList {
  Triangle *first;
  Triangle *last;
  /* and some methods ... */
};
```

If a triangle changes its state, it moves to another triangle list by simply relinking the pointers.

# 3 Creating the Triangle Adjacency Graph

We assume that the data structures are generated from an indexed face set. Faces with more than three vertices are triangulated so that only triangles must be processed. For every triangle, a new `Triangle` object, including its three `HalfEdge` objects, is created. Pointers within the triangle and the three edges are simple and fast to generate, as demonstrated in the following:

```
void init(void) {
  /* ... */
  state = 0;
  halfEdgeVec[0].next = &(edgeVec[1]);
  halfEdgeVec[0].triangle = this;
  halfEdgeVec[1].next = &(edgeVec[2]);
  halfEdgeVec[1].triangle = this;
  halfEdgeVec[2].next = &(edgeVec[0]);
  halfEdgeVec[2].triangle = this;
}
```

The main computational burden is to link corresponding half-edges with each other and to handle non-manifold cases (when more than two triangles join at an edge). To link half-edges, we use a temporary vector data structure that stores outgoing half-edges for each vertex. Experiments have shown that the fastest implementation is a vector, indexed by the start vertex indices, that points to vectors of pairs. Each pair consists of an end vertex index and a pointer to the corresponding half edge. The vector implementation is faster than a map implementation because insertion and retrieval are dominated by the search of the start vertex index. Resolving the end vertex requires constant time per vertex because each vertex has 6 outgoing half-edges on average. Consequently, we represent the temporary vector data structure as follows:

```
vector < vector < pair < int, HalfEdge *> > > temporaryVector;
```

When a triangle is added to the triangle adjacency graph (`addTriangle`), its half-edges are checked against the existing edges using the temporary vector data structure (`getHalfEdge`). If any half-edge does not already exist, it is added to the temporary vector data structure (`addHalfEdge`). Otherwise, the respective edge is non-manifold and so the triangle is labeled as isolated and stored in an extra list of isolated triangles. The core of the three required methods is outlined in Figure 1.

The temporary vector data structure implicitly contains the number of edges for every vertex. This number can be used to identify candidates for triangle fans. Triangle fans are generated if the degree of a vertex is larger than a user-specified threshold. The associated triangles are then linked into a separate list of triangle fans while removing them from the list of triangles to be processed. All remaining triangles are stored in the *valid triangle list* and are candidates for stripping.

# 4 A Simple Stripping Algorithm

Most stripping strategies try to avoid the generation of isolated triangles. During stripping, triangles are connected to strips. This process removes triangles from the set of candidates for a strip. Thus, a triangle with few neighbors is more likely to become isolated, as it 'loses' neighbors to other strips. A common strategy to avoid this is to start a strip with triangles of low degree and grow it in the direction of lowest degree [1, 5].

Consequently, all triangles from the valid triangle list are relinked into four lists of `TriangleList`, one for each degree from 0 to 3. As an invariant throughout the process, each list contains all triangles of a certain degree. As triangles are added to a strip, they are removed from the degree lists, and all adjacent triangles are dropped to the list of lower degree by simply re-linking their pointers without requiring additional memory. Consequently, a triangle strip is simply a list of pointers to triangles. After generation, strips can be traversed in either direction.

The STRIPE algorithm introduced by Evans et al. [5] adheres to the simple heuristic of starting at triangles with lowest degree and then traversing the triangle adjacency graph while generating strips. In the following, we present the stripping algorithm we have implemented which is a slight variation of the STRIPE algorithm.

For each strip, a random triangle is fetched from the list representing the lowest degree. Next, a random lowest-degree adjacent triangle is used to grow the strip in one direction. This strip is then extended by trying to build a sequence of left-right turns. If two adjacent triangles are available, the appropriate triangle is chosen. If only one adjacent triangle is available, and this triangle introduces a swap (i.e. a left-left or right-right turn), a degenerate (zero-area) triangle is inserted.

By randomly picking the starting triangle from the lowest degree list, a random stripping pattern from the set of possible partitions is generated. While according to the heuristic, all triangles of lowest degree are equally suited, computing several random partitions reveals slightly varying results. This leads to the idea of sampling several partitions and choosing the best. Random sampling has proven to be an effective method for generating approximate solutions to NP-type problems. In our setting, it also allows the use of an arbitrary cost function, because the partition could be evaluated with an external function (e.g. the rendering framerate). However, we simply use the vertex count.

A C++ implementation of this stripping algorithm, including random sampling and fanning, is available as part of the OpenSG open source scene graph project (http://www.opensg.org - OSGNodeGraph.*).

```
HalfEdge * getHalfEdge(int startVertexIndex, int endVertexIndex)
{
  /* ... */
  HalfEdge *halfEdge = 0;
  for (i = 0; i < _temporaryVector[startVertexIndex].size(); i++)
    if (temporaryVector[startVertexIndex][i].first == endVertexIndex)
    {
      halfEdge = temporaryVector[startVertexIndex][i].second;
      break;
    }
  return halfEdge;
}

void addHalfEdge(HalfEdge &halfEdge, int startVertexIndex, int endVertexIndex)
{
  /* ... */
  _temporaryVector[startVertexIndex].push_back
    (std::pair<int,HalfEdge*>(endVertexIndex,&halfEdge));

  HalfEdge *twin = getHalfEdge(endVertexIndex, startVertexIndex);
  halfEdge.vertexIndex = startVertexIndex;
  halfEdge.twin = twin;

  if (twin)
  {
    twin->twin = &halfEdge;
    halfEdge.triangle->state++;
    twin->triangle->state++;
  }
}

void addTriangle(int v0, int v1, int v2)
{
  /* ... */
  init();

  if (!getHalfEdge(v0, v1) && !getHalfEdge(v1, v2) && !getHalfEdge(v2, v0))
  {
    addHalfEdge(triangle->halfEdgeVec[0], v0, v1);
    addHalfEdge(triangle->halfEdgeVec[1], v1, v2);
    addHalfEdge(triangle->halfEdgeVec[2], v2, v0);
    /* manifold case: */
    /* add the triangle to the list of triangles to be processed */
  }
  else
    /* non-manifold case:  */
    /* add the triangle to the list of isolated triangles */
}
```

Figure 1: Adding a triangle to the triangle adjacency graph.

# 5 Programmer's Interface

Clients of our implementation need only use the `TriangleAdjacencyGraph` class, which provides five public methods to be used for stripping:

- `void reserve(int vertexNum, int triangleNum)` - an optional method to use before stripping . This method improves performance by preallocating all memory for the stripping data structures. Using this method results in a slightly better performance.
- `void addTriangle(int v0, int v1, int v2)` - inserts a triangle described by the three vertex indices `v0`, `v1`, and `v2` into the data structures before stripping.
- `int calcOptPrim(int iteration, bool doStrip, bool doFan, int minFanFaces)` - performs the stripping by finding triangle fans with minimum `minFanFaces` triangles and triangle strips when the corresponding flags `doStrip` and `doFan` are set. The method returns the number of vertices to be sent to the renderer.
- `int primitiveCount(void)` - returns the number of generated primitives after stripping.
- `int getPrimitive(vector<int> &primIndex)` - returns the primitive type (strips, fans, triangles) and the vertex indices stored in a vector after stripping.

# 6 Results

We compared the execution times and number of triangle strip vertices of our OpenSG implementation to those of the publicly available version of STRIPE [5], the FTSG algorithm [11], and the Game Programming Gems Algorithm [7] using the recommended options. NVIDIA's tristrip algorithm [8] turned out to be so slow that we did not run any tests on it. Results measured on a Pentium IV at 1.7 GHz with 512 MB are depicted in Table 1; they consist of the required time for triangle stripping and the number of vertices in the final stripped model that must be sent to the graphics subsystem. We divided the time of our OpenSG implementation into the triangle adjacency graph creation time and the stripping time.

| Model | Triangles | STRIPE V2.0 | | FTSG V1.2 | |
|---|---|---|---|---|---|
| | | sec. | vertices | sec. | vertices |
| bunny | 69,451 | 1.8 | 82,182 | 0.33 | 81,412 |
| dragon | 871,414 | 19.5 | 1,140,991 | 4.60 | 1,121,151 |
| buddha | 1,087,716 | 24.2 | 1,423,102 | 5.92 | 1,398,464 |

| Model | Triangles | GameGems | | **OpenSG** | | |
|---|---|---|---|---|---|---|
| | | sec. | vertices | **sec.** (graph set up + stripping) | | **vertices** |
| bunny | 69,451 | 6.50 | 88,330 | **0.12** | (0.08 + 0.04) | **81,676** |
| dragon | 871,414 | 1384 | 1,233,809 | **1.45** | (0.81 + 0.64) | **1,129,330** |
| buddha | 1,087,716 | 2096 | 1,538,041 | **1.80** | (1.05 + 0.75) | **1,408,325** |

Table 1: Comparison of stripping time and quality (number of vertices to be sent to the graphics subsystem).

# 7 Conclusions

We have presented an improved adjacency data structure for fast triangle stripping. Stripping with our algorithm is so fast that it can be done on the fly while loading a mesh. This is due to the following reasons: first, the adjacency data structure can be created quickly and robustly from any indexed face set; second, its cache-friendly layout is specifically designed to efficiently answer common stripping queries, such as neighbor finding and least-degree triangle finding, in constant time; third, the stripping algorithm operates in-place, since strips are created by simply relinking pointers. Our implementation of the stripping algorithm is part of the new version of OpenSG. We believe that the improved adjacency data structure can be used in other applications besides triangle stripping, such as, for example, subdivision surfaces.


## Acknowledgements

The bunny, dragon, and buddha models are from the Stanford Graphics Repository. We thank the anonymous reviewers for their valuable comments and Dave Burke for proofreading the paper.


## Web Information

Source code for the adjacency data structure and the stripping algorithm, as well as the test results and sample images, are available at `http://www.acm.org/jgt/papers/ReuterBehrAlexa05` The OpenSG implementation is available at `http://www.opensg.org`

# References


[1] Kurt Akeley, Paul Haeberli, and Derrick Burns. tomesh.c : C program on SGI developer's toolbox CD, 1990.

[2] Mike M. Chow. Optimized geometry compression for real-time rendering. *IEEE Visualization '97*, pages 346–354, 1997.

[3] Michael F. Deering. Geometry compression. *Proceedings of SIGGRAPH 95*, pages 13–20, 1995.

[4] Francine Evans, Steven Skiena, and Amitabh Varshney. Completing sequential triangulations is hard. Technical report, State University of New York at Stony Brook, 1996.

[5] Francine Evans, Steven Skiena, and Amitabh Varshney. Optimizing triangle strips for fast rendering. *IEEE Visualization '96*, pages 319–326, 1996.

[6] Hugues Hoppe. Optimization of mesh locality for transparent vertex caching. *Proceedings of SIGGRAPH 99*, pages 269–276, 1999.

[7] Carl S. Marshall. Triangle strip creation, optimizations, and rendering. In *Game Programming Gems 3*, pages 359–366. Charles River Media, 2002.

[8] NVIDIA. *NvTriStrip Library*. www.nvidia.com, 2003.

[9] OpenSG. *An Open Source SceneGraph*. www.opensg.org, 2003.

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

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