Fast and Effective Striping

Johannes Behr
ZGDV Darmstadt
jbehr@zgdv.de

Marc Alexa
TU Darmstadt, GRIS
alexa@gris.informatik.tu-darmstadt.de

January 27, 2002

Abstract

An algorithm for striping triangle meshes in the context of OpenSG is presented. The technique is similar to STRIPE, however, a careful yet simple implementation leads to significant faster execution. Furthermore, we propose to randomly sample several partitions and chose the one that minimizes a given cost function. More samples are expected to lead to better solutions, thus, introducing a parameter to scale between speed and quality.

1 Introduction

Rendering triangular meshes requires to send the vertices of each triangle to the graphics subsystem. In order to minimize the necessary bandwidth it is favorably to send the triangles so that consecutive triangles share an edge, which allows to specify each triangle (except the first one) with a single vertex. Such sequences of triangles are called *strips*. Note that finding the minimum number of strips to cover a given polygonal mesh is a NP hard problem. However, several heuristical algorithm achieve partitions close to the theoretical bounds [5, 7].

Lately, the idea of strips has been generalized to a *vertex cache*. Instead of storing only the two last vertices for the possible definition of a new triangle, the vertex cache stores more, which would theoretically allow to specify each triangle with less than one vertex [2, 4]. However, in practice it seems that the perfomance gain over triangle strips on commodity hardware is negligible - as is documented in NVIDIA white papers and stated as a 'hindsight' by Hoppe referring to his work on optimizing vertex locality to exploit the vertex cache [6].

Here, we present a simple implementation of a variant of the algorithm developed by Evans et al [5], whose implementation is publicly available as STRIPE. The execution of this new implementation within OpenSG turns out to be so fast that we decided to add the option of generating several random sample partitionings. This allows to trade in speed for a possible improved partition. In addition, the cost function used to chose a partition could be adapted to suite the needs of application and available graphics hardware.

2 Algorithm

The algorithm adheres to the simple heuristic of starting at faces with lowest degree. The degree of a face refers to the number of adjacent faces which have not been added to a strip.

In the first step of the algorithm faces are triangulated. Then, complex edges (i.e. edges with more than two incident faces) are resolved by cutting the mesh at the edge. This is done automatically when generating the data structures for the process. Note that complex vertices are no problem for striping provided that the right data structures are used. All faces are sorted into bins of their respective degree.

For each strip, a face is fetched from the bin representing the lowest degree. A random neighboring face is used to build the strip into one direction. This strip is extended trying to build a sequence of left-right turns. If only one adjacent face is available, which would introduce a swap (i.e. a left-left or right-right turn), a dummy face is inserted. If no adjacent face is avalable the strip is extended from the starting face into the other direction.

By randomly picking the starting face for the strips from the lowest degree bin a random sample partioning is generated. Accordin to the heuristic, all faces of lowest degree are equally suited. Random sampling has proven to be an effective method for generating approximate solutions to NP-type problems. In our setting, it has the additional benefit of using an arbitrary cost function to chose the best sample. This cost function could, for example, be measured as the actual frame rate in the target application. In our implementation we simply use the vertex count.

3 Implementation

The mesh connectivity is stored in a half-edge type data structure (e.g. double-connected edge list [3] or directed edges [1]). When a face is added to the data structure and one if its oriented edges exists the oriented edge is repeated, thus, effectivily cutting the non-manifold mesh at this edge. To accomodate complex vertices, all edges incident upon a vertex are stored explicitly as an array. In particular, each edge has fields `next`, `twin`, and `face` referring to the respective elements of the data structure. Each face has a `edge` field containing an arbitrary bounding edge and a `processed` field, which is true once the face has been added to a strip.

The degree bins are implemented as doubly-connected lists. Each list has a pointer to the list with the next lower degree. Each face has a pointer to the list it is contained in. This data structure is geared towards the repeated operation of degree reduction of a face while striping. If a face is added to a strip, its degree as well as the degrees of all adjacent faces have do be decremented. Using the pointer structure, reducing the degree of a face works as follows: The face is removed from the current list by redirecting the pointers of its list neighbors and added to the next lower degree bin using the pointer between the lists. For convience, faces have methods `release()`, to reduce the degree and set the `processed` flag as well as `dropNeighbors()` to reduce the degree of adjacent faces.

The generation of a strip consists of several phases, which are stored as the `walkMode`. The `walkMode` is one of {`START`,`LEFT`,`RIGHT`,`FINISH`}, each of which will be explained in detail later. Using a c-like notation the main loop of strip generation looks like this:

```
while(nodesLeft > 0) {
  switch (walkMode) {
    case START: ...
    case LEFT: ...
    case RIGHT: ...
case FINISH: ...
  }
```

To indicate the start of a new strip, `walkMode` is set to `START` and the followin steps are executed:

```
  case START:
    firstDirection = true;
    stripIndex++;
    strip[stripIndex].clear();

    for (lowestDegree = 0; lowestDegree < 4; lowestDegree++)
```

```
    if (faceList[lowestDegree].size() > 0) {
      currentFace = faceList[lowestDegree].first();

  strip[stripIndex].add(currentFace->index);
  currentFace->release();
  currentFace->dropNeighbors();

  walkMode = NEW;
  nodesLeft--;
```

After this, `currentFace` contains a face of lowest possible degree obtained from the degree bins. This face has already been added to the current `strip` with index `stripIndex`. The `currentFace` is marked as `processed` (as part of `release()`) and it is moved to the next lower degree bin. Now, the walking direction has to be chosen.

```
  nextFace = 0;
  lowestDegree = 4;
  edge = currentFace->edge;
  do {
    if (edge->twin->face != 0 && !edge->twin->face->processed)
      if (gateEdge->twin->face->degree < lowestDegree) {
        nextFace = gateEdge->twin->face;
        lowestDegree = nextFace->degree;
        gateEdge = edge;
      }
    edge = edge->next;
  while (edge != currentFace->edge);
  firstEdge = gateEdge;
  if (nextFace) {
    strip[stripIndex].add(gateEdge->index);
    strip[stripIndex].add(nextFace->index);
    nextFace->drop();
    nextFace->dropNeighbors();
    cost++;
    walkMode = RIGHT;
    nodesLeft--;
    firstTurn = START;
  } else {
    walkMode = START;
  }
  break;
```

For a new face, all adjacent faces are inspected. From the set of existent and non-striped neighbors the one with lowest degree is picked and stored in `nextFace`. Throughout the strip generation, `gateEdge` stored the edge between `currentFace` and `nextFace`. In addition, `firstEdge` keeps the first gate edge for later striping in the other direction from the start face.

If a valid `nextFace` has been found, the face and the gate edge are pushed in the strip, face degrees are adjusted, and `walkMode` is set to `RIGHT`.

```
case RIGHT:
  currentFace = gateEdge->twin->face;
  gateEdge = gateEdge->twin->next;
  nextFace = 0;
  if (gateEdge->twin->face != 0 && !gateEdge->twin->face->processed) {
```

```
      nextFace = gateEdge->twin->face;
      walkMode = LEFT;
      if (firstTurn == START)
        firstTurn = RIGHT;
    } else {
      gateEdge = gateEdge->next;
      if (gateEdge->twin->face != 0 && !gateEdge->twin->face->processed) {
        nextFace = gateEdge->twin->face;
        if (firstTurn == START)
          firstTurn = LEFT;
      }
    }
    break;
```

Here, `currentFace` is updated and `gateEdge` is set to the expected gate for right turn (i.e. the next ccw edge from the gate). If the face adjacent to the gate edge exists and is has not been processed `walkMode` and `nextFace` are set accordingly. If this preferred face is not available, the other open edge is tried as a possible gate edge. In this case, `walkMode` stays unchanged. If this turn was the first one, `firstTurn` has store the move.

The `LEFT` case looks quite similar. The only difference is that instead of using the next edge as gate, the previous one should be used. For triangle meshes this is identical to `next->next`.

```
  case LEFT:
    currentFace = gateEdge->twin->face;
    gateEdge = gateEdge->twin->next->next;
    nextFace = 0;
      if (gateEdge->twin->face) {
        nextFace = gateEdge->twin->face;
        walkMode = RIGHT;
        if (firstTurn == START)
          firstTurn = LEFT;
      } else {
        gateEdge = gateEdge->next->next;
        if (gateEdge->twin->face) {
          nextFace = gateEdge->twin->face;
          walkMode = LEFT;
          if (firstTurn == START)
            firstTurn = RIGHT;
        }
      }
      break;
```

If a `nextFace` has been found, it ahs to be added to the strip and the degrees have to be updated. In case the strip cannot be continued either the other direction from start face is tried or the strip is finished.

```
  case LEFT:
  case RIGHT:
    if (nextFace) {
      strip[stripIndex].add(gateEdge->index);
      strip[stripIndex].add(nextFace->index);
      nextFace->drop();
      nextFace->dropNeighbors();
```

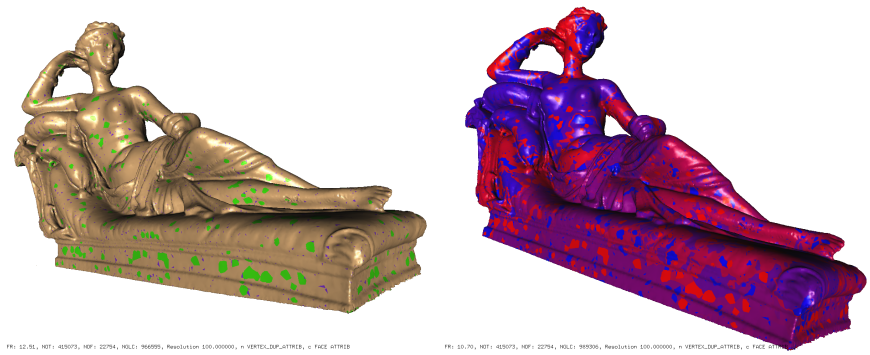

Figure 1: The result of the striping procedure. The left picture shows triangle fans in green and isolated triangles in blue. On the right, strips are color according to when the respectiver triangles have been processed.

```
    cost++;
    nodeLeft--;
  } else {
    if (firstDirection) {
      walkMode = firstTurn;
      gateEdge = firstEdge;
      strip[stripIndex].flipped = true;
      firstDirection = false;
    } else
      walkMode = FINISH;
  }
  if (nodesLeft <= 0)
    walkMode = FINISH;
  break;
```

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

4 Results

First results with the striping algorithm show a significant speed-up for striped vs. non-striped meshes, as expected. A set of strips for a triangle mesh is computed at a speed of more than 500K triangles/sec on commodity hardware (e.g. a 1GHz Pentium PC). Table 1 shows results achieved with our implementation. Since the striping process conatins some randomness, the model is striped 100 times. Visual results of the striping algorithm are depicted in Figures 1.

We compare the execution times and number of strip vertices of our implementation to the those of the publicly available version of STRIPE. For STRIPE we use the the NOSWAP option, which generally produces the best results. For comparison average vertex count results from our implementation are used together with execution times for a single run. Note that this is what one could expect when running the algorithm a single time. The results are depicted in the Table 2.

Model			Vertices in Strips			Time in sec	
Name	# Vertex	# Face	min	avg	max	start-up	striping
Bunny	34834	69451	81390	81684.6	81875	0.12	0.09
Lady	178445	356902	458816	459304	459905	0.48	0.36
Elephant	27015	54026	69026	69196.1	69350	0.05	0.05
Dino	2832	5660	7208	7258.5	7312	0.01	0.01
Horse	48485	96966	116716	116988	117263	0.13	0.11

Table 1: Results of our striping implementation for 100 runs per model. The time for building up data structures has to be spent only once, independent of the number of striping runs.

	STRIPE		OpenSG	
Model	verts	time	verts	time
Bunny	82128	1.78 sec	81684.6	0.26 sec
Lady	463989	7.96 sec	459304	0.85 sec
Elephant	69948	1.19 sec	69196.1	0.11 sec
Dino	7340	0.12 sec	7258.5	0.02 sec
Horse	117538	2.32 sec	116988	0.25 sec

Table 2: Running times and striping results of the STRIPE code and our implementation in OpenSG. Vertex counts for the OpenSG implementation show the *expected* result (i.e. the average of 100 runs). Timings include built up of data structures and allocation of memory. The new implementation is consistently faster and produces slightly better results.

References

[1] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. Directed edges–a scalable representation for triangle meshes. *Journal of Graphics Tools*, 3(4):1–12, 1998. ISSN 1086-7651.

[2] Mike M. Chow. Optimized geometry compression for real-time rendering. *IEEE Visualization '97*, pages 346–354, November 1997. ISBN 0-58113-011-2.

[3] Mark de Berg, Mark van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry – Algorithms and Applications*. Springer-Verlag, Berlin Heidelberg, 1997.

[4] Michael F. Deering. Geometry compression. *Proceedings of SIGGRAPH 95*, pages 13–20, August 1995. ISBN 0-201-84776-0. Held in Los Angeles, California.

[5] Francine Evans, Steven S. Skiena, and Amitabh Varshney. Optimizing triangle strips for fast rendering. *IEEE Visualization '96*, pages 319–326, October 1996. ISBN 0-89791-864-9.

[6] Hugues Hoppe. Optimization of mesh locality for transparent vertex caching. *Proceedings of SIGGRAPH 99*, pages 269–276, August 1999. ISBN 0-20148-560-5. Held in Los Angeles, California.

[7] Xinyu Xiang, Martin Held, and Joseph S. B. Mitchell. Fast and effective stripification of polygonal surface models. *1999 ACM Symposium on Interactive 3D Graphics*, pages 71–78, April 1999. ISBN 1-58113-082-1.