Bremen

CGVR

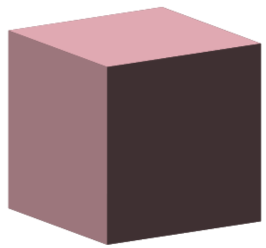
Advanced Computer Graphics
Striping / Stripping

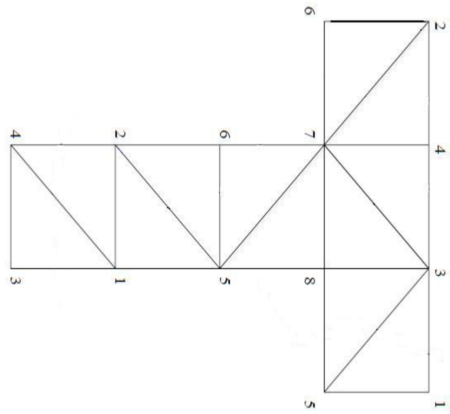
G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

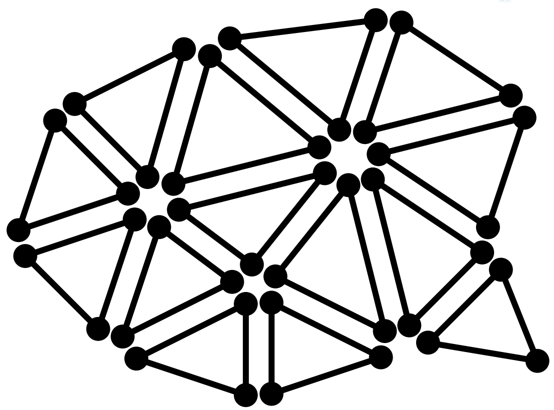
Motivation

- In the following, consider only *triangle* meshes
- Naïve rendering:
  - *N* triangles→ 3*N* vertices have to be sent to the graphics card
- Implementation in OpenGL:

```
glBegin( GL_TRIANGLES );
for ( unsigned int i = 0;
        i < n_tris; i++ )
{
      glVertex3fv( tri[i][0] );
      glVertex3fv( tri[i][1] );
      glVertex3fv( tri[i][2] );
}
glEnd();
```

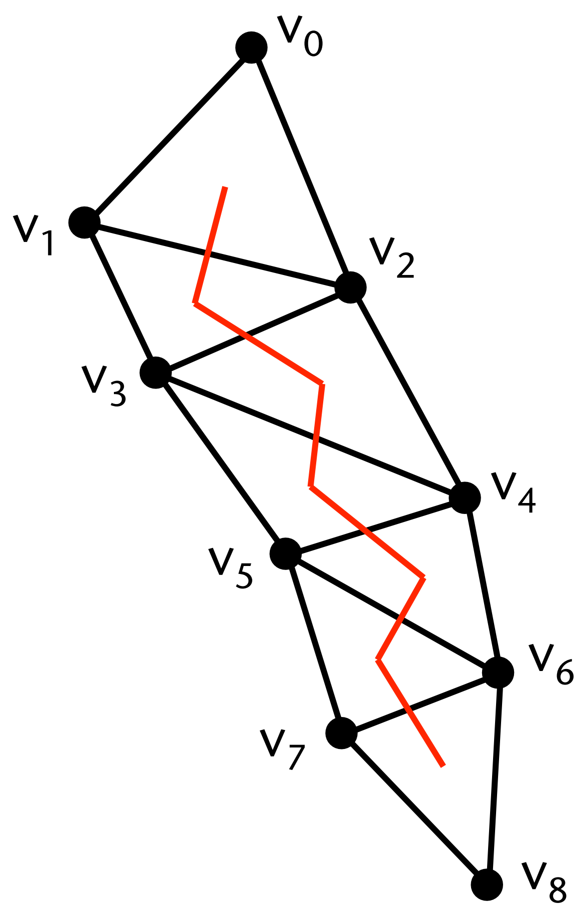






Solution

- Graphics cards offer a special primitive: the triangle strip
- The idea:
  - The graphics card always "remembers" the 2 vertices which it received last
  - With each transmission of a new vertex, the graphics card forms a new triangle out of the new and the 2 "old" vertices
- Example:
  - 9 vertices → 7 triangles
- Advantage: factor 3 less vertex data need to be transmitted and processed!



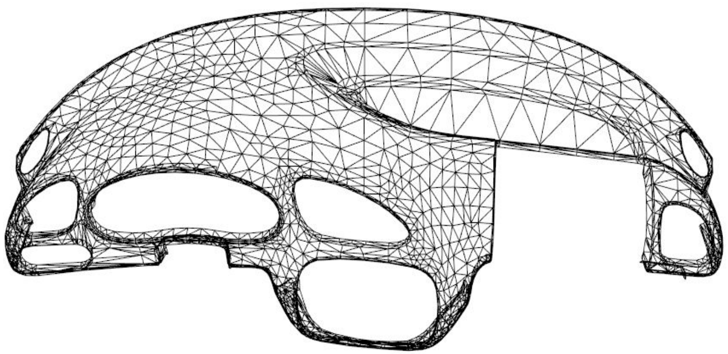
Transmitted vertices:

v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8

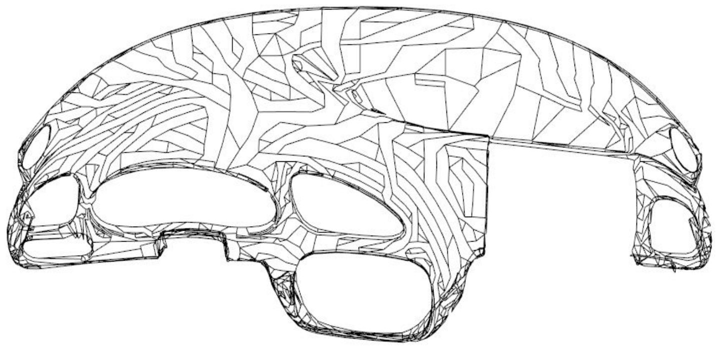
- Implementation in OpenGL:

```
glBegin( GL_TRIANGLE_STRIP );
for ( unsigned int j = 0; j < strip.n_verts; j ++ )
    glVertex3fv( strip.v[j] );
glEnd();
```

Example of a "Striped" Object



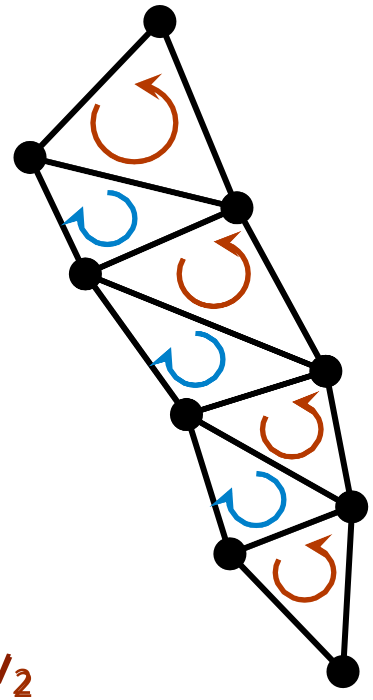
4320 polygons
12960 vertices

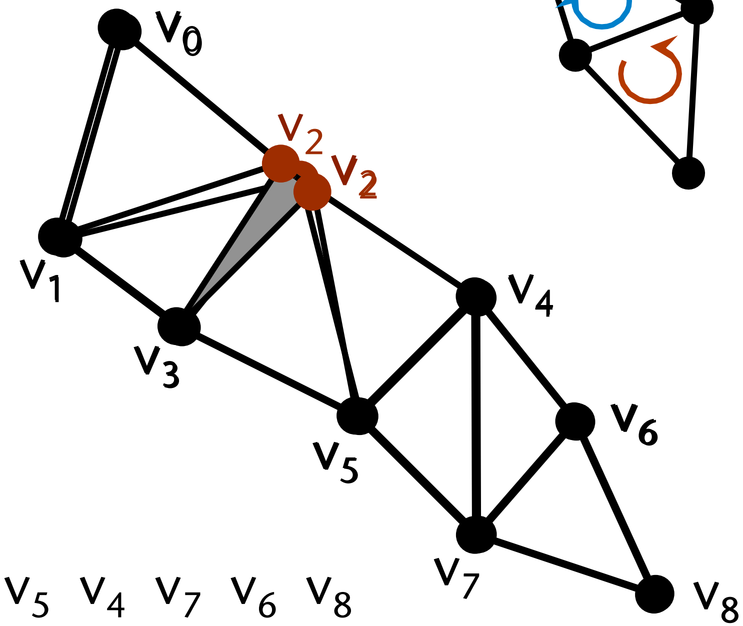


905 stripes
6127 vertices

Some Concepts

- Definition:
  A triangle strip is a sequence of triangles in a mesh, so that two triangles following each other have a common edge.
- What about the orientation of the triangles?
- What can be done in such cases?
- Solution:
  - V_2 has to be transmitted twice
  - V_2 is called a swap-vertex – it creates a degenerated triangle

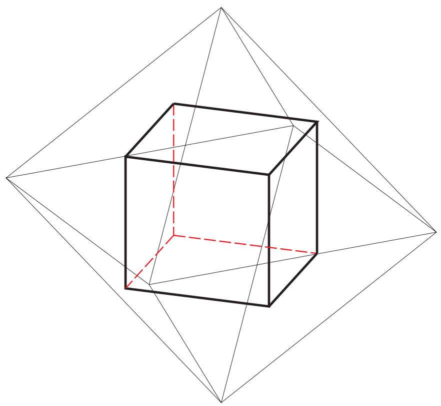


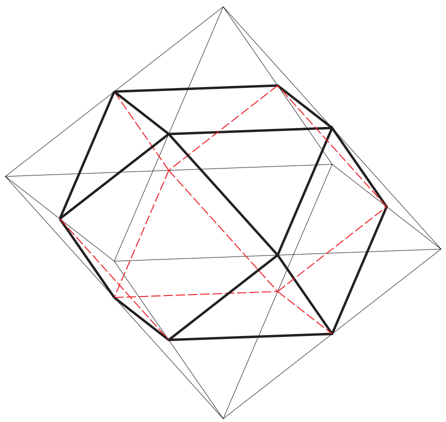


v_0 v_1 v_2 v_3 v_2 v_5 v_4 v_7 v_6 v_8

A Geometric "Denksport-Aufgabe"

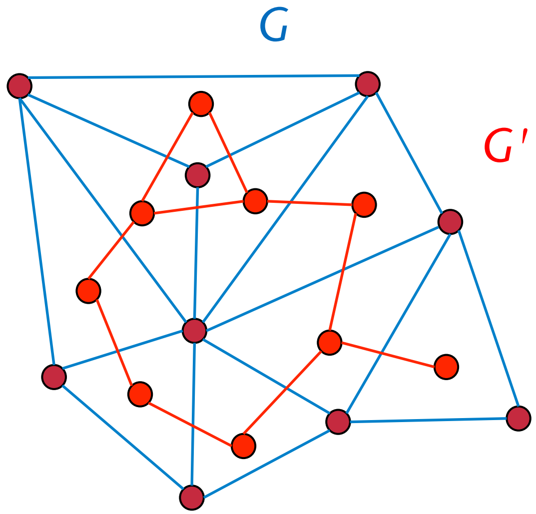
- Take an octahedron; determine the midpoints of its facets; connect the midpoints of adjacent facets ⟶ which polyhedron emerges?
- Take an octahedron; determine the midpoints of its edges; connect the midpoints of adjacent edges ⟶ which polyhedron emerges?
- Take a cube; connect the midpoints of adjacent facets ⟶ which polyhedron do you get?





An NP-Complete Problem

- Questions:
  1. Is it possible to create a single strip out of each mesh?
  2. How can one (or more) strip(s) be created efficiently?



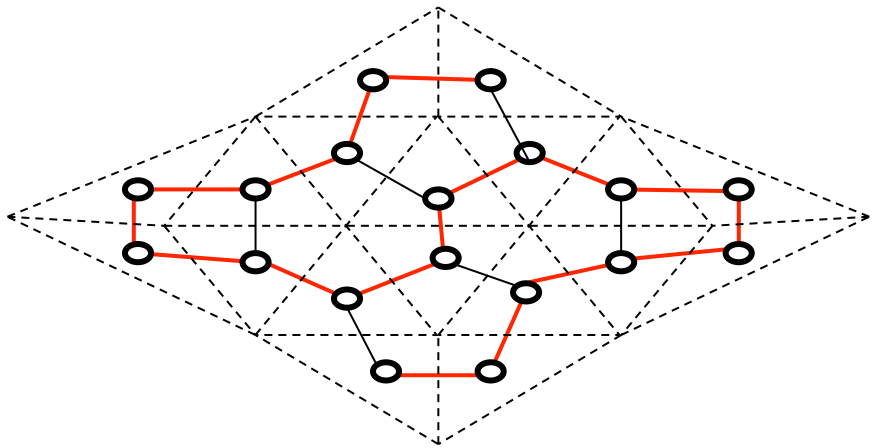
- Reduction of the problem:
  - Utilize the dual graph
- Definition of the dual graph:
  Given a mesh / planar graph $G = (V, E, F)$.
  The *dual graph* $G' = (V', E', F')$ is derived from G as follows: replace each facet of F by a node in V' and connect two nodes in V' by an edge, iff their original facets share a common edge (i.e., are adjacent).

- Proposition:
  The problem "*decide for any triangle mesh M, whether it is possible to turn it into a single tri-strip*" is NP-complete!

- Proof, part 1:
  - To show: the verification of a "candidate" is in the class P
  - Let G' be the dual graph of the original mesh M
  - We have: $|E'| \in O(|V'|) = O(|F|)$
  - For E' create an adjacency matrix (using a hash table, this costs $O(|E'|)$)
  - Let a candidate strip be $(v'_{i_1}, v'_{i_2}, v'_{i_3}, \ldots, v'_{i_n})$
    - Each v'_{i_k} of G' corresponds to a triangle in M
  - Check each pair $(v'_{i_k}, v'_{i_{k+1}})$ of the candidate, whether it is contained in E'

- Proof, part 2: reduction *from* a known NP-complete problem
  - The reduction from the know NP-complete onto our problem must be ∈ P
- Definition Hamilton path :

  Given a graph *G*. A *Hamilton path* is a path through *G*, so that each vertex is visited *exactly once*.



- Example :
- Observations :
  - A mesh/graph *G*
    has a single triangle strip iff
    the dual graph *G'* has a Hamilton path
  - In case all facets within the (closed) mesh *G* are triangles, then all nodes in *G'* have degree 3 (= number of incident edges)

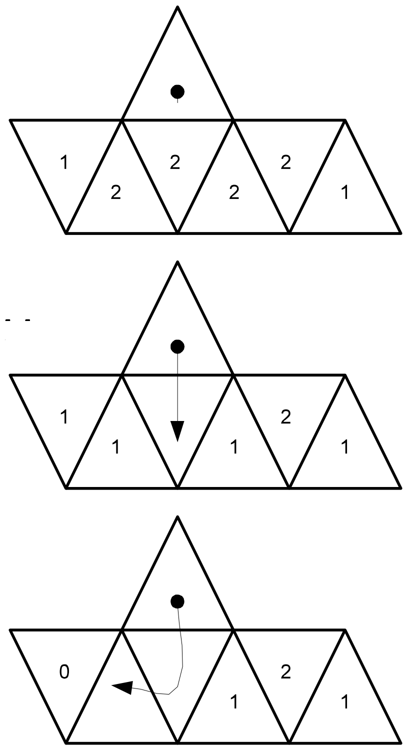
- Theorem from graph theory:
  The problem to decide, whether a given graph possesses a Hamilton path, is NP-complete.
  This is even the case, if all nodes within the graph have degree 3!

- Conclusion: "only" try to create as few strips as possible

Tasks

- Stripification = stripping of a mesh into triangle strips
- Optimization task: only as few strips as possible, and overall as few "double" vertices / swap-vertices as possible
- Definitions:
  Free triangle := triangle that does not yet belong to a strip
  Degree of a triangle := number of *free* neighbor triangles

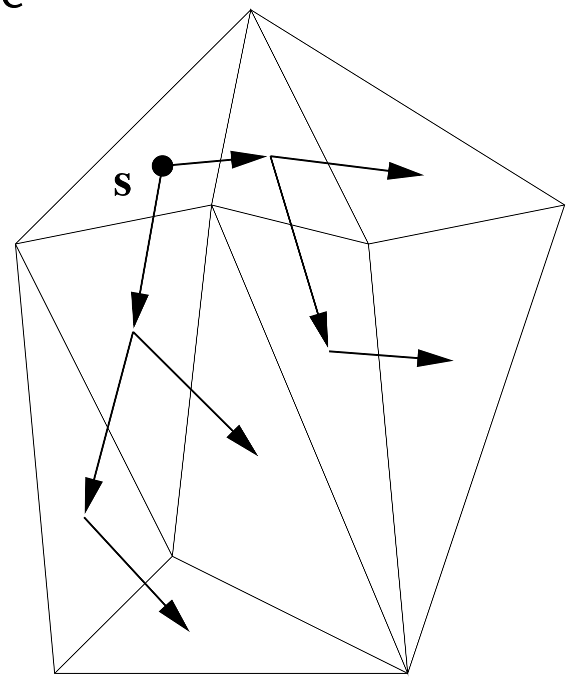
- The SGI-Algorithm [Akeley, 1990]:
  - Use a greedy strategy
  - The local criterion: go to the neighboring triangle with the smallest degree
  - Do look-ahead for tie-breaking



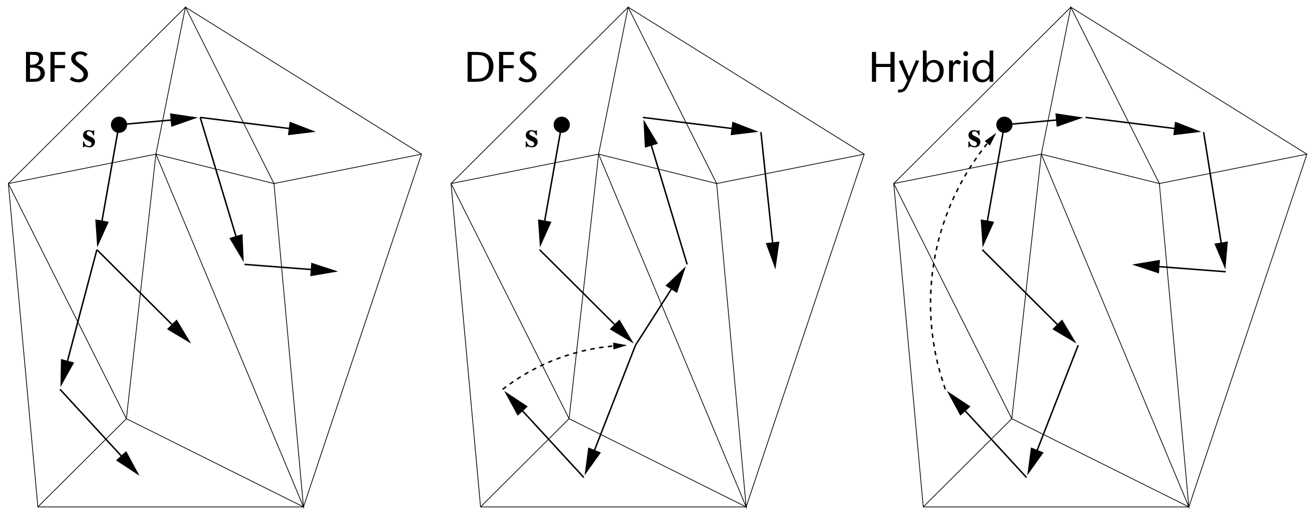

```
while ex. still free triangles:
  choose triangle with smallest degree
  start new strip with this triangle
  while last triangle in current strip has free neighbors:
    choose the neighbor with the lowest degree
    if tie:
      look one step ahead
    add triangle to current strip
```

- The "Fast Triangle Strip Generator" (FTSG) [Xiang et al., 1999]:
  - One of the best algorithms (in the sense of rendering performance and the time of construction)
- Overview of the algorithm:
  1. Create a *spanning tree T* in the dual graph of the tri-mesh
  2. Partition *T* into as few paths as possible
  3. Possibly do post-processing: try to unite short strips
- Definition: spanning tree

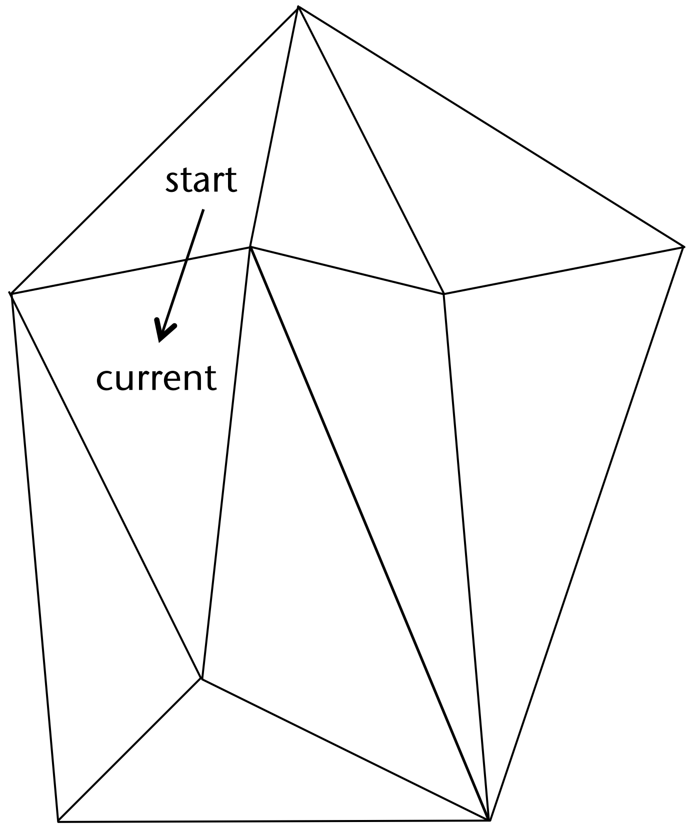
- Regarding step 2 (partitioning the spanning tree *T*):
    - Select a node *v* within *T* with degree 3 (has two children) and a level as deep as possible
    - The sub-tree with root *v* consists of a *single path* from the left to the right side; store this strip; delete the path from *T*
    - Repeat, until no node with degree 3 remains
- Running time: *O*(*n*) (with the suitable data structures)



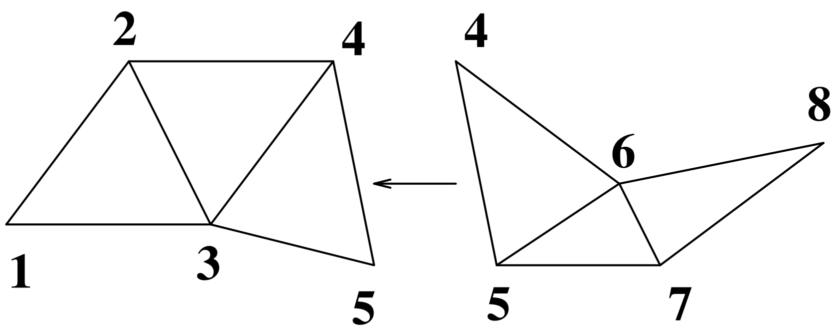
- Regarding step 1 (construction of the spanning tree *T*):
  - Aim: try to construct a spanning tree with a minimal number of nodes that have degree 3 or higher = 2+ children
    - This is a consequence from step 2
  - Typical procedures for the construction of the spanning tree:
    - Breadth-first search (BFS) through the original graph
    - Depth-first search (DFS
  - Practice shows: DFS yields the best results in this case



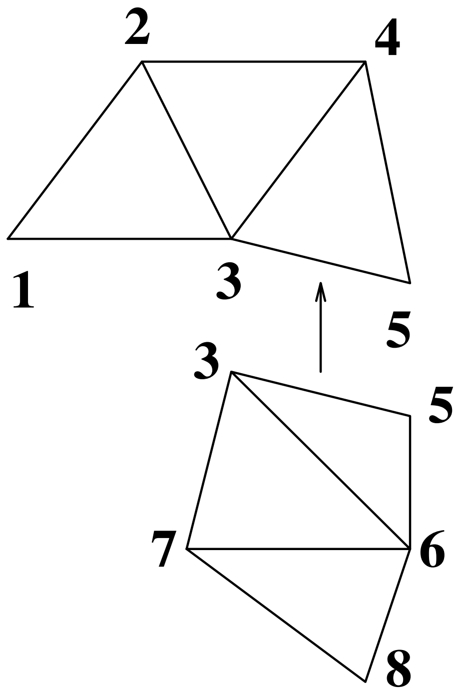
- Additional heuristics (since we need the spanning tree for special purposes):
  - Observation: during DFS through the graph, we have very often a free choice which neighboring node shall be visited next
  - ➢ Heuristic here: choose the neighboring triangle (= neighboring node in spanning tree *T*) such that the current triangle and this neighboring triangle would not produce a *swap vertex*, if they were in a common triangle strip



- Regarding step 3 (post-processing = concatenation of short strips):
  - The setup of strips (on the graphics card) costs more time than to send a vertex
  - Consequence: concatenate several short strips, even if that requires to insert a swap-vertex
  - Cases:



Good case



And 2 more cases … (reduction by 1 vertex)

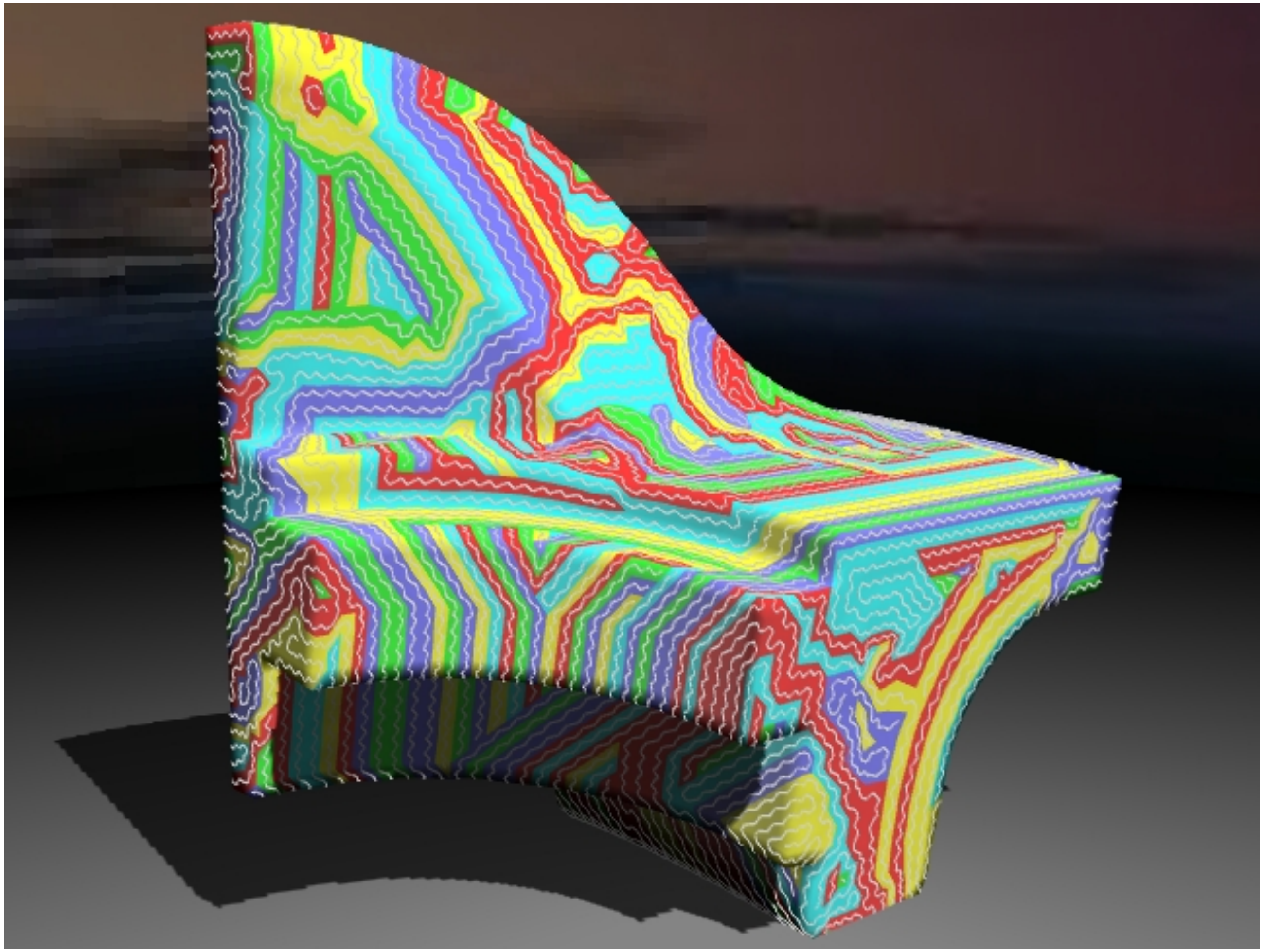
- Result: on average, ca. 1.23 vertices per triangle are sent during rendering
- Data structure: DCEL is a suitable candidate
- Observation: The performance depends very heavily on the DS!
  - With pre-implemented, generic DS (e.g., from a library) one is finished with the job quickly, but the performance is mediocre
  - With specially adapted, "hand-made" DS, the performance (hopefully!) is very good, but the implementation takes much longer

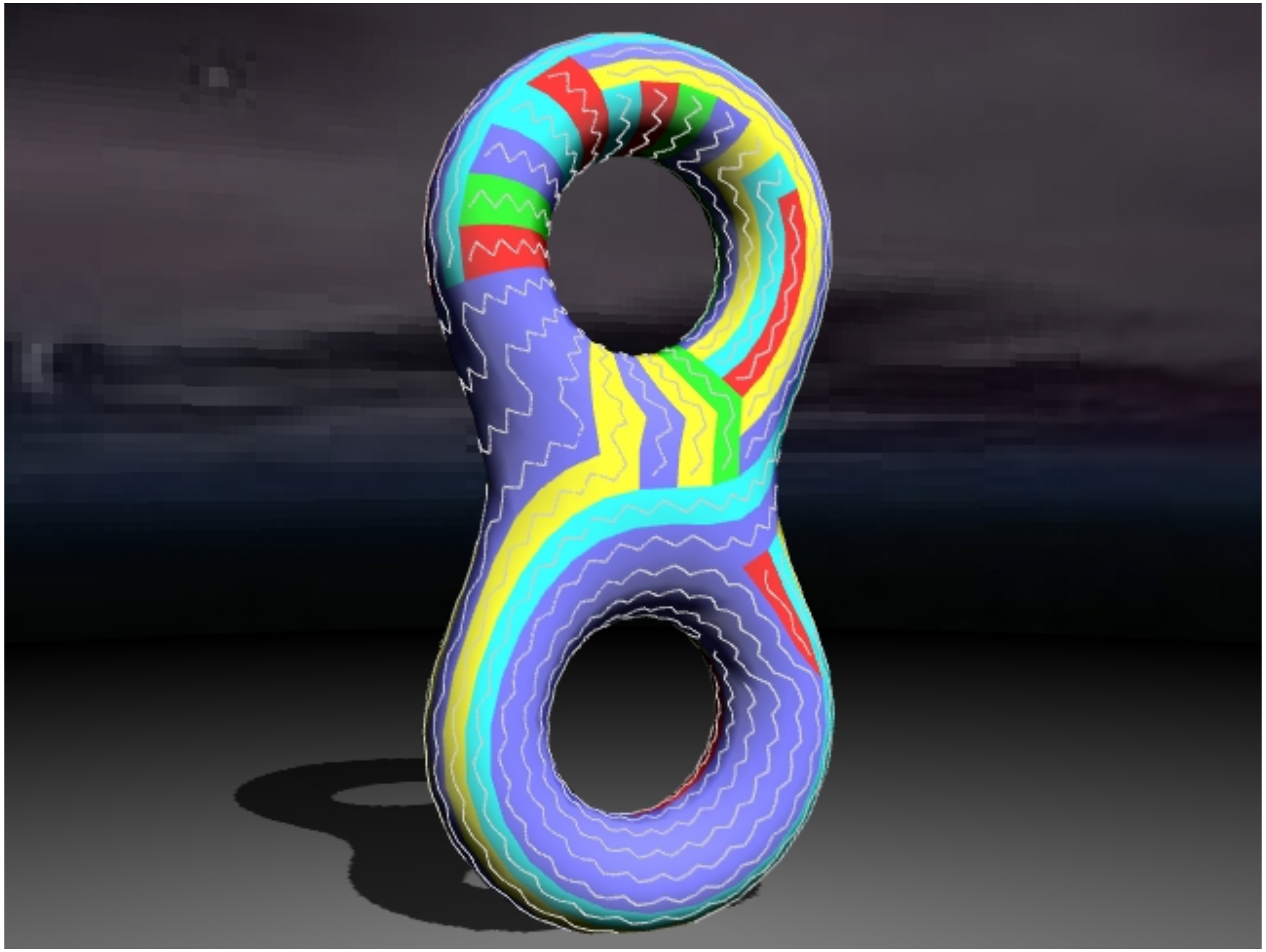
- A special data structure for stripification:
  - Utilize the fact that we only work with triangle meshes
  - Store all three incident edges directly with the triangles

```
class Triangle
{
   HalfEdge halfEdges[3]; // faster access
   uint degree;           // 0, ..., 3
   Triangle *nextInList,  // for d-list
            *prevInList;
   Triangle *child[2];    // for spanning tree
}
```

    - This minimizes "pointer chasing", makes better use of CPU cache
- This is a mix of face-based and edge-based data structures

Examples





Caveats

- Today's graphics cards are so complex and possess so many further features, that it is not clear, whether / how much striping gains us rendering performance!
  - Some GPUs have an integrated index cache (helpful when rendering indexed face sets), so that the pipeline first checks whether a new vertex index is in the cache; if so, then the transformed vertex is retrieved directly from the cache
- In the case where expensive vertex shaders are used, striping surely gains performance
- On mobile graphics chips, striping gains performance for "normal" rendering (as of 2012)
  - Because they have no caches, and much less features
  - But the opinions are mixed ...