A Compact Cell Structure for Scientific Visualization

W.J. Schroeder
Boris Yamrom

GE Corporate Research & Development
Schenectady, NY 12301

Abstract

Well designed data structures and access methods are vital to developing efficient visualization algorithms. The cell structure is a compact, general data structure for representing n-dimensional topological constructs such as unstructured grids, polygonal, or triangle strip representations. The cell structure also provides constant time access methods for a wide variety of visualization algorithms. This paper describes the representation, access methods, and implementation of the cell structure. Sample algorithms such as decimation, triangle strip generation, and streamline propagation are used to illustrate its application.

1.0 Introduction

The bulk of the visualization literature is oriented towards algorithms and graphical representational schemes[1],[2],[3]. Data structures, if described at all, are often presented superficially. However it is only with well designed combinations of both algorithm and data structure that useful visualization techniques can be created[4].

Visualization data tends to have some particular characteristics: the data size is large and the data type is varied. Large data are simply the result of the basic goal of visualization - to transform large data into more comprehensible forms. The data is varied because visualization techniques are general. An iso-surface generation algorithm[5],[5] is just as useful applied to medical data as it is to financial visualization. Hence visual data structures must be both compact (i.e., small memory requirement) and general (i.e., represent a wide variety of data).

The cell structure is a compact and general data structure for representing cell topology. Cell topology consists of points plus and a particular ordering of points (i.e., a cell). One or more cells may share a given point as well as other topological features such as edges and faces. The most important feature of the cell structure is that it represents adjacency, or topological neighborhood information, with minimal memory requirement. The cell structure has also been designed with access methods that support a wide variety of visualization algorithms.

A number of similar data structures have been previously developed[7],[8]. The simplest structures are variations of display lists: lists of points, and polygon/element/cell connectivity. While these data structures are compact, performing operations requiring adjacency information results in algorithms of $O(n^2)$ time complexity since searching is required. A variation of this data structure uses a supplemental list to represent adjacency information[9],[10]. For each cell of particular type and dimension (e.g., hexahedron with six faces), a list of neighbors is maintained (e.g., the six face neighbors of the hexahedron). This structure is particularly useful when the cell type and topology is the same for all cells. However, when mixed cell type and topology is required, the structure becomes unwieldy. Also, in order to build this structure, an $O(n^2)$ search is initially required. More elaborate data structures include the hierarchical winged edge[11] and radial edge[12] structures. Hierarchical structures explicitly represent topology in terms of a hierarchy of increasing topological dimension: vertices, edges, faces, regions. Although extremely powerful constructs, fully elaborated hierarchical structures require large amounts of memory than the simpler ones just described.

The cell structure is a variation of the display list structure with additional hierarchical information. It can simultaneously represent cells of mixed topology, and provides constant time access to adjacency information. Moreover, the cell structure is more compact than hierarchical structures since its hierarchical information is implicitly represented.

2.0 Cell Structure

In this section the mathematical basis, representation, access methods, and implementation of the cell structure is described.

2.1 Mathematical Basis

The cell structure is based on the topological construct called a cell, Ci. A cell is an ordered sequence of

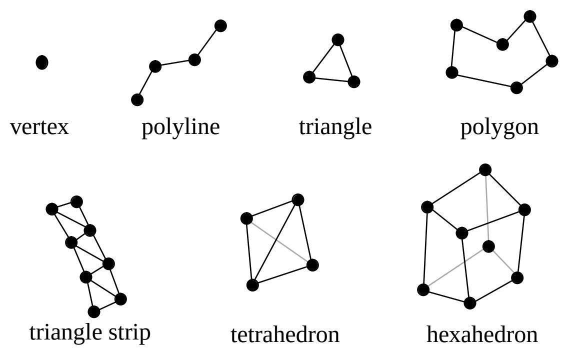

Figure 1. Some example cell types.

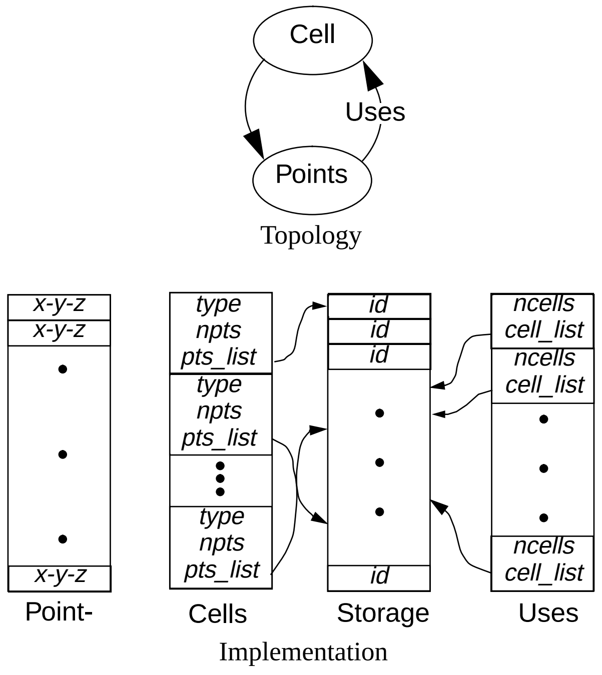

Figure 2. Cell representation.

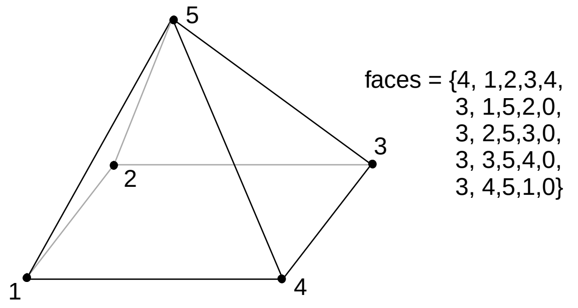

Figure 3. Accessing implicit topology

points $Ci = \{p_1, p_2, \ldots, p_n\}$ with $p_i \in P$ where P is a set of *n*-dimensional points. The particular meaning of the sequence of points, or cell topology, is determined by the *type* of cell. The number of points *n* defining the cell is the *size* of the cell.

Examples of cells (Figure 1) include points (0D topology), lines (1D topology), polygons and triangle strips (2D topology), and unstructured grid elements such as tetrahedron, hexahedron, pyramids, and triangular prisms (3D topology). Higher dimension cells are also possible, such as n-dimensional simplices.

A key concept of the cell structure is the "use" of a point by a cell. A cell C_i "uses" a point p_i when $p_i \in C_i$. Hence the "use set" $U(p_i)$ is the collection of all cells using p_i:

$$U(p_i) = \{C_i : p_i \in C_i\}$$

2.2 Representation

The cell structure expresses the relationship between points *P*, cells *C*, and use sets *U*. The representation of this information may take a variety of forms, but the preference is for a data structure that is simple, compact, and can be accessed in constant time. It is also desirable that the structure can be directly retrieved from or written to storage. That is, the use of pointers, or memory locations, is not directly evident in the data structure.

The implementation of the cell structure consists of four *dynamic arrays* (Figure 2).A dynamic array is simply an array (e.g., a contiguous, addressable memory space) that grows dynamically to accomodate new data. The elements of the array are addressed with a unique, non-negative integer id.

The first array represents a list of points *x-y-z* coordinates whose id is the array access id. The second array represents the cells: the size, type, and a list of point ids that define the cell. In the third dynamic array the use arrays for each point are expressed. Each use array consists of the number of cells using a particular point. That is, at use array position *i* the cells that use point *i* are listed. The final dynamic array provides storage and consists of id types in no particular order. This array is not required, but it provides a contiguous pool of memory from which the cell arrays and use arrays construct their lists. Using the storage array reduces memory fragmentation greatly, and reduces the number of system calls to acquire memory.

A key feature of the cell structure is that it implicitly represents intermediate topology between the cell (*n*-dimensional) and the defining points (0-dimensional). As a result, supplemental information is required to address the $(n-2)$ layers of intermediate cell topology. If a polygon is represented as an ordered sequence of points, then pairs of adjacent points represent polygon edges. A more complex topology is a pyramid. The list shown in Figure 3 allows direct access to the five faces of the pyramid. The first number represents the number of points defining the face, followed by a list of ordered indices into the cells point list (1-offset). This implicit representation of topology is the reason why the cell structure can compactly represent adjacency information.

Other information can represent the relationship between the cell geometry and topology. A typical example is when traversing cells, such as streamline tracking.

Often the traversal is computed using parametric coordinates, and when a boundary is encountered (parametric coordinate value ± 1), it is necessary to obtain the corresponding topology (e.g., face or edge).

Another important capability of the cell structure is the ability to simultaneously represent cells of different type. For example, Figure 5 shows an unstructured grid consisting of hexadra, tetraheda, pyramids, triangular prisms (3D topology), polygons (2D topology), and lines (1D topology).

2.3 Access Methods

There are three categories of methods for manipulating the cell structure: *primitive methods, topological methods,* and *adjacency methods.* A description of these methods follow.

2.3.1 Primitive Methods

Primitive methods are used for creating, destroying, modifying, and traversing the cell structure. Sample methods include:

Cells = initialize ()
Create an empty cell structure and return a pointer *Cells* to the structure. Optional arguments for specifying initial storage size are possible.

create_point (Cells, pt_id, x)
Given a point id and x-y-z coordinate, create a point in the *Cells* structure.

create_cell (Cells, cell_id, type, npts, pts)
Given a cell id, a cell type, the number of points, and the point ids defining the cell, create a cell in the *Cells* structure.

build_uses ()
Using the current cells and points, build the use lists for the *Cells* structure.

npts = get_number_points (Cells)
Return the number of points in the structure.

ncells = get_numer_cells (Cells)
Return the number of cells in the structure.

destroy (Cells)
Release the *Cells* structure back to system memory.

Building the cell structure consists of creating points and cells, followed by the building the use lists. This process is of linear time complexity. For each cell in the structure, a traversal of the cell's point list is made. Then for each point in the cell's point list, the corresponding use list is updated. Once all cells are visited the data structure is complete. In some applications the *build_uses()* method need not be executed. Then the structure is basically a display list.

2.3.2 Topological Methods

Topological methods provide access to the topology of the cell. Some of these methods are as follows.

get_cell_pts (Cells, cell_id, pts, npts)
Return a list of point ids that define the given cell.

get_pt_cells (Cells, pt_id, cells, ncells)
Return a list of cell ids that use the specified point.

These operators return information directly from the cell structure. It is also possible to return other topological information using supplemental information associated with the type of the topology. For example, if the cell type is a pyramid, then accessing the faces of the pyramid can be implemented using the table of Figure 3 and the operator:

get_cell_face_pts (Cells, cell_id, face_id, npts, pts)
Return a list of point ids that define the specified face of the given cell.

2.3.3 Adjacency Methods

Adjacency methods are used to obtain information about the neighbors of a cell. A neighbor of a particular cell $\bar{C}$ is simply a cell that shares one or more points in common with $\bar{C}$. Examples of these methods follow.

get_cell_pt_nbr (Cells, cell_id, pt_id, nbrs, nnbrs)
Given a point and cell, return a list of neighboring cells that use the point.

get_cell_feature_nbr (Cells,cell_id,pts,npts,nbr,nnbrs)
Given a list of points from a cell, return a list of neighboring cells that each use all the specified points.

The *get_cell_feature_nbr()* method is useful for extracting adjacency information across topological features. For example, if the feature is specified by two points defining an edge, this method returns edge neighbors. Or, if the points define a face, this method returns a face neighbor.

The adjacency operators are simple set operations. For a particular cell $\bar{C}$ and point list $\bar{P} = (\bar{p}_1, \bar{p}_2, \ldots, \bar{p}_n)$ with $\bar{P} \subset P$, where $\bar{P}$ typically corrsponds to a topological feature of the cell, the result of the *get_cell_feature_nbrs()* method is the adjacency set $A(\bar{C}, \bar{P})$. The adjacency set is simply the intersection of the use sets for each point, excluding the cell $\bar{C}$

$$A(\bar{C}, \bar{P}) = \left(\bigcap_{i=1}^{n} U(\bar{p}_i)\right) - \bar{C}$$

The adjacency set implicitly represents a variety of useful information. In a manifold object represented by a polyhedra, for example, each polygon must have exactly one edge neighbor for each of its edges. Edges that have no neighbors are boundary edges; edges that have more than one edge neighbor represent non-manifold topology.

Volume data sets that consist of 3D cells (e.g., unstructured grids) are topologically consistent only if for each cell there is exactly one face neighbor for each face. Faces that have no neighbors are on the boundary of the volume. More than one face neighbor implies that the neighbors are self-intersecting.

The construction of an adjacency set is of $O(n)$ time and space complexity provided that a constant bound can be placed on the number of uses of a point. Examining Equation 2 above, if $n \le \mathrm{MAX\_USE} \ll npts$, and $\mathrm{MAX\_USE}$ is independent of *npts*, then the time complexity of the set operations are bounded by a fixed constant. It is possible to design pathological cases where a particular point is used by every cell, but in application such situations do not occur. Typically a point is used by 5-6 triangles in a triangle mesh, or eight hexahedra in a hexahedral mesh.

3.0 Algorithms

A variety of algorithms have been implemented using the cell structure. In the following subsections a few are expressed in pseudo-code using the cell access methods described earlier. The pseudo-code examples are simplified to demonstrate the use of the data structure.

3.1 Streamlne Propagation

A common vector field visualization technique is to generate streamlines. Streamlines are the path that a massless particle takes when moving through the vector field. Typical examples include visualizing fluid flow.

The cell structure provides a convenient structure to propagate the streamline through a computational grid (Figure 6). Computational grids are typically composed of many thousands or perhaps millions of cells in which numerical computation is carried out. The streamline traverses many of these cells, and the propagation algorithm requires tracking the streamline from cell to cell.

```
determine initial cell cell, position x, and velocity v;
while ( inside Cells) {
    $x_{i+1} = x_i + v_i \cdot \Delta t$ ;
    map $x_{i+1}$ to cell coordinates (r,s,t);
    if ( (r,s,t) outside cell ) {
        find cell face f that streamline passed through;
        get_cell_face_nbrs (Cells, cell, f, nbrs, nnbrs);
        if (nnbrs < 1) outside Cells;
        else cell = nbrs[0];
    }
    evaluate velocity $v_i$ in cell at (r,s,t);
    $x_i = x_{i+1}$;
}
```

3.2 Decimation

The goal of the decimation algorithm is to reduce the number of polygons in a polygonal mesh, while maintaining the original topology of the mesh. Schroeder et al[15] have implemented this algorithm using the cell structure and have achieved reductions of greater than 90% on maximum model sizes of 1.7 million triangles. Figure 7 shows a 90% decimated model of a human face.

The algorithm repeatedly visits all non-decimated vertices, gathering the polygons surrounding each vertex into a list. These polygons are than evaluated against a local planarity (or edge) condition. If the condition is satisfied, then the vertex and using polygons are deleted, and the resulting “hole” is triangulated. The process repeats until an appropriate number of vertices are eliminated.

(pseudo-code)

3.3 Feature Normals

Realistic rendering of polygonal representations depends upon using vertex normals to smooth the transition from one polygon to the next. Frequently polygons are generated without normals, and techniques must be used to generate the normals from the polygon connectivity.

One naive approach to normal generation is to compute a vertex normal by averaging the normals of polygons using the vertex. This works well in situations where the dihedral angles between polygons is small, and when the polygons are all ordered consistently. In many cases, e.g., a cube, the angles between polygons are quite large, resulting in images that appear less than realistic.

In the algorithm that follows, normals are generated on a polygonal mesh that may or may not be consistently ordered. In addition, polygons whose dihedral angle is greater than a specified feature angle are separated along their common edges (Figure 8).

```
for each cell in Cells { /*make order consistent*/
    if (not visited cell) order (cell); /*recursively reorder*/
}
for each cell in Cells { /*compute cell normals*/
    get_cell_pts (Cells, cell, pts, npts);
    generate polygon normal;
}
for each point p in Cells { /*split feature edges*/
    if (not visited p) split(p); /*recursively split*/
}
for each point p in Cells{
    get_pt_cells (Cells, p, cells);
    determine average normal based on using cells;
}

order (cell) {/* recursive reorder function */
    mark cell visited;
    get_cell_pts (Cells, cell, pts, npts);
    for each edge (p1,p2) in cell pts {
        get_cell_edge_nbrs (Cells, cell, p1, p2, nbrs, nnbrs);
        if (nnbrs > 0 && nbrs[i] not visited) {
            if (nbrs[i] edge order not (p2,p1) reverse(nbrs[i]);
            order (nbrs[i]); /*recursive call*/
        }}}

split (p) {/* recursive edge splitting function */
    mark p visited;
    get_pt_cells (Cells, pt, cells, ncells);
    for each cell cells[i] {
        get_cell_pts (Cells, cells[i], pts, npts);
        for each cell edge (p,pts[i])
          get_cell_edge_nbrs (Cells,cells[i],p,pts[i],nbrs, nnbrs);{
          if ( nnbrs > 0 ) && (nnbrs > 1 or
              dihedral angle > feature angle ) {
            create new point pnew;
            replace p (in cells[i]) with pnew;
          } }
        if (not visited pts[i]) split (pts[i]);
      }}
```

3.4 Triangle Strip Generation

Triangle strips are compact representations of adjacent triangles. Most rendering hardware supports triangle strips directly as a high-performance graphics primitive. Unfortunately triangle strips are not typically generated in visualization algorithms. Instead, as this simple algorithm illustrates, triangle strips can be easily generated from triangles (or polygons) using the cell structure.

A typical application of this algorithm is shown in Figure 9. Here the original data of x polygons is stripped to produce y strips. The modest length of the strips is misleading: the result is a six-fold increase in rendering speed.

```
initialize list of triangle strips strips;
for each cell in cell list Cells {
  if cell not visited {
    mark cell visited;
    start new triangle strip strip;
    get_cell_pts (Cells, cell, pts, npts);
    for each cell edge p1,p2 { /* assumed triangles */
      get_cell_edge_nbrs (Cells, cell, p1, p2, nbrs, nnbrs);
      if (nnbrs>0 && (nbr=nbrs[0]) not visited) break;{
    }/* start growing strip */
    while ( nbr != NULL ) {
      add nbr to strip;
      mark nbr visited;
      get_cell_edge_nbrs (Cells, nbr, p1, p2, nbrs, nnbrs);
      if (nnbrs < 1) nbr = NULL else nbr=nbrs[0];
    }
    add strip to strips;
  }
}
```

3.5 Extrusion

A simple geometric construct is an extrusion or sweep. Here it is assumed that the starting point for this operation is a collection of points, lines, and polygons, and that the surface is swept along some path to create a "volume".

The use of the cell structure in this algorithm is to identify boundary edges, i.e., polygon edges that are used by only a single polygon. These edges when swept create the sides of the resulting polyhedron (Figure 10).

```
for each cell in cell list Cells {
  if cell type is POINT {
    extrude point to create line;
  } else if cell type is LINE {
    extrude line to create triangle strip;
  } else {
    for each edge (p1,p2) in cell {
      get_cell_edge_nbrs(Cells, cell, p1, p2, nbrs,nnbrs);
```

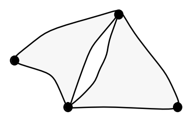

Figure 4. Curvilinear triangles

```
        if (nnbrs == 0) {
          extrude edge to create triangle strip;
        }
      }
    }
}
```

4.0 Conclusion

The cell structure is a compact, general data structure that has been used to implement a variety of visualization algorithms. The cell structure is particular useful when cell adjacency information is required. Such operations require *O(1)* time and space complexity. Hence the cell structure can be used to implement algorithms of *O(n)* time complexity.

As compared to more complex data structures, the cell structure is limited in two important ways. First, the cell structure does not represent "ordering" information. That is, given a topological feature such as an edge, the particular order of using faces around the edge. Second, because intermediate topology is implicit, certain types of topology requiring explicit information cannot be properly represented. Figure 4 is one such example. Here two curvilinear triangles share common vertices, put not edges. In such situations more explicit hierarchical structures, such as the winged edge or radial edge structures, are appropriate. It is possible to address these limitations by performing local geometric processing to order usage, or to add conditional hierarchial information to the cell structure.

References

[1] B. H. McCormick, T. A. DeFanti, and M. D. Brown. Visualization in Scientific Computing. *Computer Graphics,* 21(6), Nov. 1987.

[2] N. M. Patrikalakis, editor. *Scientific Visualization of Physical Phenomena*. Springer-Verlag. Tokyo 1991.

[3] G. M. Nielson and B. Shriver, editors. *Visualization in Scientific Computing*. IEEE Computer Society Press. Los Alamitos, CA. 1990.

[4] A. V. Aho and J. E. Hopcroft and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company. Reading, MA. 1983.

[5] W. E. Lorensen and H. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, 21(4):163-169, July, 1987.

[6] G. Wyvill and C. McPheeters and B. Wyville. Data Structures for Soft Objects. *Visual Computer.* 2(4):227-234

[7] A. Paoluzzi and F. Bernardini and C. Cattani and V. Ferrucci. Dimension-Independent Modeling with Simplicial Complexes. *ACM Transactions on Graphics.* 12(1), January, 1993.

[8] E. Brisson. Representing geometric structures in d-dimensions: Topology and order. *ACM Symposium on Computational Geometry.* ACM Press, New York,1989.

[9] R. Haimes and M. Giles. VISUAL3: Interactive Unsteady Unstructured 3D Visualization. AIAA Report No. AIAA-91-0794. January, 1991.

[10] L. Gelberg, D. Kamins, D. Parker, and J. Stacks. Visualization Techniques for Structured and Unstructured Scientific Data. *SIGGRAPH `90 Course Notes for State of the Art Data Visualization*. August, 1990.

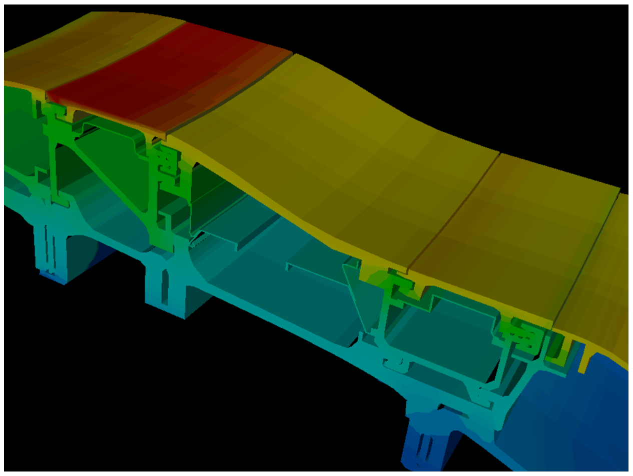

Figure 5. Unstructured grid representation.

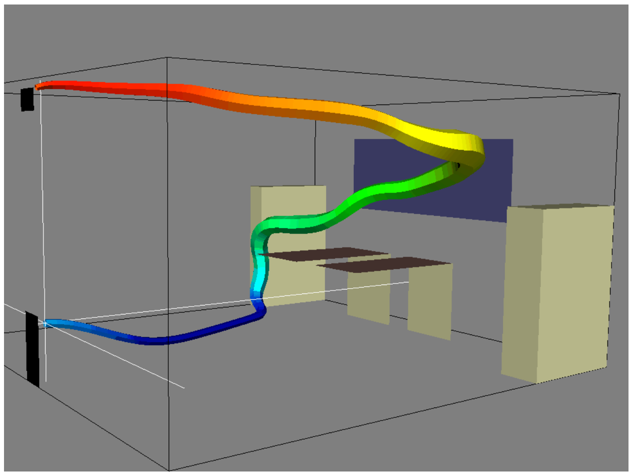

Figure 6. Stream tube propagation.

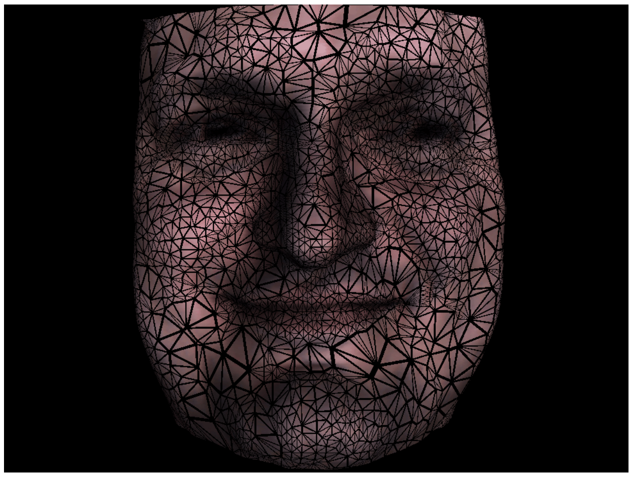

Figure 7. 90% decimated polygonal mesh, triangles shrunk to show shape.

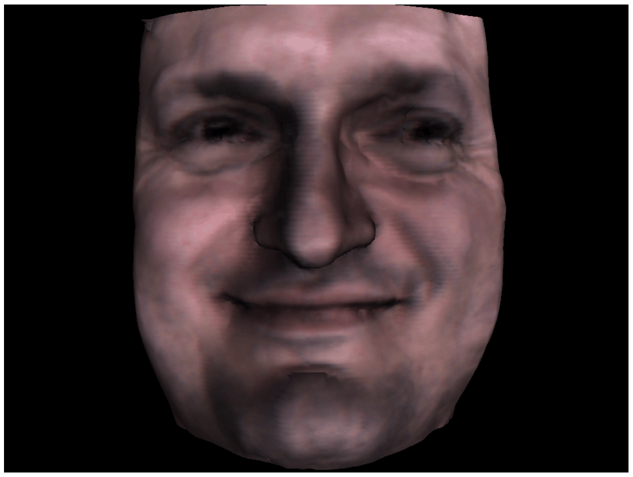

Figure 8. Feature normal generation from decimated geometry of Figure 7.

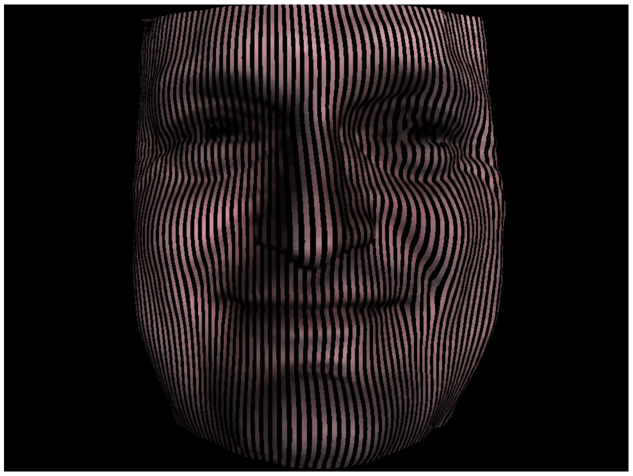

Figure 9. Triangle strip generation from original range data. Every other triangle strip is turned off.

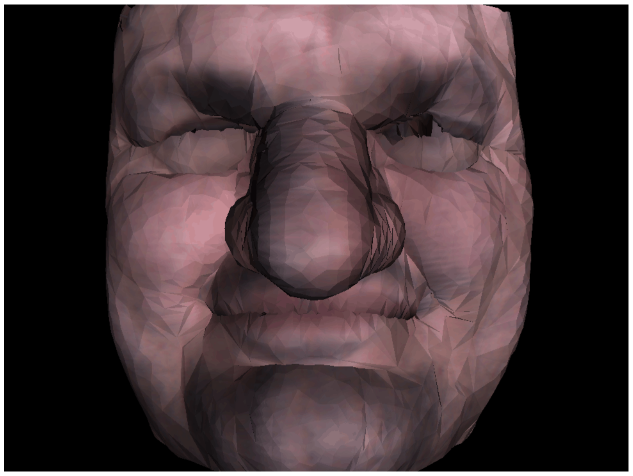

Figure 10. Extrusion (along surface normals) of surface from Figure 8 to create closed geometry.