- Draft -

Efficient Handling of Shading Discontinuities for Progressive Meshes

Henrik Buchholz
Hasso Plattner Institute
University of Potsdam
buchholz@hpi.uni-potsdam.de

Jürgen Döllner
Hasso Plattner Institute
University of Potsdam
doellner@hpi.uni-potsdam.de

Abstract

For visual perception of 3D models, shading plays one of the major roles. The shading quality of level-of-detail models is limited generally because existing LOD algorithms assume a conceptually smooth surface. A complex mesh, however, is likely to have conceptually smooth and angular parts. We introduce an extension to the progressive mesh LOD approach that efficiently handles triangle meshes with large numbers of shading discontinuities. For this the algorithm distinguishes three principal shading situations for mesh vertices: completely continuous shading, completely discontinuous shading, and mixed continuous/discontinuous shading. Remarkably, the algorithm does not introduce any overhead for completely smooth surfaces. As one field of application, we briefly outline its application for LOD representations of 3D city models.

1. Introduction

Multiresolution modeling aims at reducing the complexity of 3D models to optimize rendering performance, storage, or data transmission. Various algorithms and data structures have been developed in the past to cope with this task. A prominent example represents the progressive mesh representation [5] used to generate LOD approximations of triangle meshes.

Most multiresolution modeling techniques are designed for and operate on implicitly smooth surfaces. In the case of complex polygon meshes, however, we cannot always assume implicit surface smoothness, for example in the case of most man-made or machine-made objects. In these cases, the visual characteristic of the polygon meshes is determined to a considerable degree by angular parts, represented by hard edges. Figure 1 compares results from LOD representations that do and do not distinguish hard and soft edges. From a rendering perspective, hard edges represent *shading discontinuities*, common vertices of adjacent polygons do not share vertex normals (Figure 2).

For any kind of surface shading, vertex and face normals are essential, because they represent the main parameters for illumination and shading calculations.

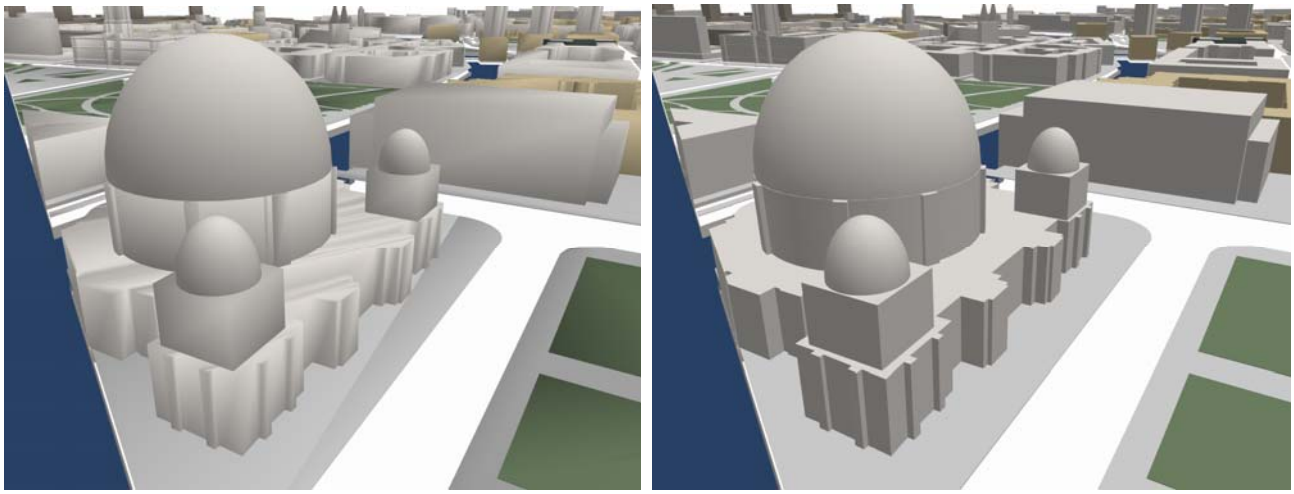


Figure 1. LOD representation of complex 3D buildings without (left) and with (right) distinguished smooth and angular parts (left).

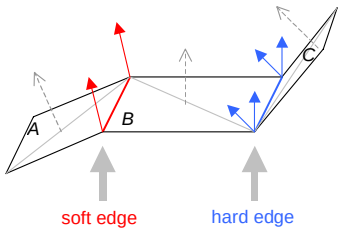


Figure 2. Polygon mesh containing both smooth and angular parts (A-B, B-C). Even edges of a single polygon (B) can exhibit both soft and hard edges.

We distinguish between three principal shading situations at mesh vertices: completely continuous shading, completely discontinuous shading, and mixed continuous/discontinuous shading. The partitioning enables us also to handle the shading discontinuities in a memory-efficient manner. Since the data transfer between application memory and graphics hardware represents the core bottleneck in real-time rendering (as well as in the case of progressive transmission of data), the efficient handling of vertex normals in general, and the efficient handling of different types of normals are essential for an implementation of the progressive mesh technique.

2. Related Work

Simplification algorithms for polygon meshes mostly use one of the following four local operations: vertex clustering (e.g., [8][10]), vertex removal (e.g., [12][13][15]), face collapsing (e.g., [3][4]), and edge collapsing (e.g., [5][7][17]) or a generalization of the latter (e.g., [2][9]). Our approach can be applied to extend any simplification algorithm that is exclusively based on edge collapsing (Figure 3) or the more general pair contraction [2]; we therefore discuss related work in this context.

Hoppe introduced the progressive mesh data-structure [5], which has been used and extended by several authors (e.g. [1][2][11][16]). It efficiently stores a sequence of simplified meshes, beginning with the simplest mesh (*base-mesh)* and ending with the original one. The central idea is to take advantage of the reverse collapse operation (*split* operation). A progressive mesh consists of the explicitly stored base-mesh and a sequence of split operations.

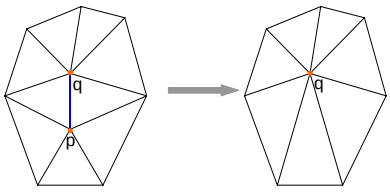


Figure 3. Result of an edge-collapse operation.

For creation, Hoppe [5] uses an energy function considering the surface geometry, scalar attributes, and attribute discontinuities to select edges to be collapsed and to determine positions of new vertices.

One known way to manage shading discontinuities is to replicate the vertices at the corresponding sharp edges. This is applicable for meshes for which the number of shading discontinuities is small in comparison to the mesh complexity. But if these discontinuities are predominating in the mesh, the replication of all these vertices would involve considerable overhead in memory and performance. In addition, simple replication of vertices introduces problems during the simplification addressed in [6], in which Hoppe suggests a flexible concept to implement the progressive mesh data-structure. This implementation is also able to manage meshes with attribute discontinuities by associating scalar attribute values not with vertices, but with *wedges* [6]. A wedge is a set of vertex-adjacent corners whose attributes are the same. This concept is well suited for many types of meshes, but not for those with a large number of shading discontinuities, which we are especially interested in: For example, in a 3D city model, it is not improbable that the model contains approximately exclusively face normals. Since each wedge belongs to a singular vertex, the face normal of most triangles had to be stored in three wedges.

Garland presented in [2] the quadric error metrics, which achieve a connection of high quality approximations and short pre-calculation time. During the pre-calculation for each vertex a *quadric* is stored consisting of a symmetric 4×4 matrix, a 3-vector and a scalar. The quadrics are used to quickly calculate the sum of squared distances of a given point from a set of planes being represented by the quadric. Hoppe [7] extends this concept by support for scalar attributes at vertices, e.g., ver-

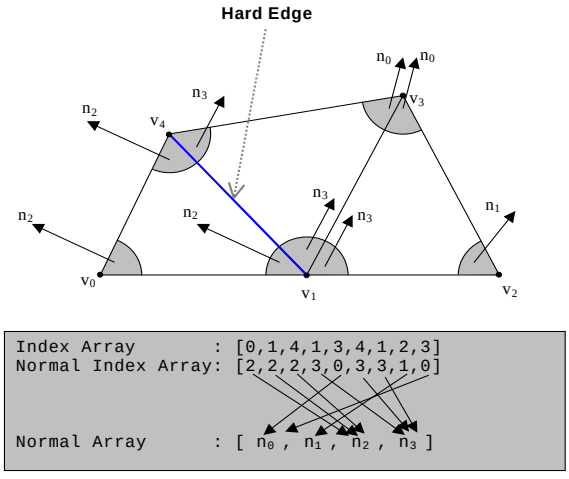
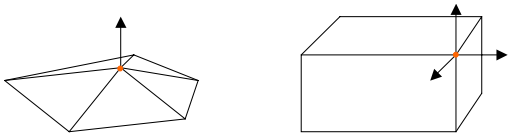


Figure 4. An input mesh with one hard edge.



a) Example of an SN-vertex. b) Example of an NSN-vertex.

Figure 5. Different types of vertices.

tex normals or texture coordinates. These extensions are orthogonal to our extensions and could be used to improve the quality of the simplification on smooth regions. Since meshes with large numbers of shading discontinuities consist mainly of non-smooth regions, we used the simple variant of Garland's algorithm [2], which is fast and easy to implement.

3. Algorithm

Input. The input of the algorithm consists of four arrays. The mesh geometry is specified by a *vertex array* and a related *index array*. The normal of each mesh corner is specified by a *normal array* and a related *normal-index array*. Each triangle is represented by a sequence of three indices. The term *corner* denotes a (*vertex*,*triangle*) tuple according to [6]. The normal at the corner (*v*, *T*) specifies the normal for *T* at vertex *v*. We assume that two normals n_1 and n_2 are equal if and only if their indices i_1 and i_2 are identical. Two or more corners share a *common normal* if and only if their corresponding normal indices are equal (Figure 4).

Vertex Types. We distinguish different types of vertices and triangles, which form central elements of our algorithm. They are defined in the following:

We call a vertex *v shared-normal vertex* (SN-vertex), if all corners that are adjacent to *v* have a common normal at *v* (Figure 5a). In the neighborhood of an SN-vertex, the surface appears smooth. If *v* does not fulfill the SN-condition (Figure 5b), we call it a *non-shared normal vertex* (NSN-vertex).

Triangle Types. Based on this classification of vertices, we distinguish three different types of triangles. We call a triangle *T* an *SN-triangle* if all vertices of *T* are SN-vertices. If *T* is no SN-triangle, but has a face normal, i.e., all its vertices have a common normal, we call it an *FN-triangle*, otherwise *T* is referred to as *NSN-triangle*. The algorithm brings most profit for meshes with a large number of shading discontinuities, i.e., a large number of FN-triangles.

The algorithm is subdivided into three phases (Figure 6): Restructuring of the mesh data, processing the simplification steps based on an adapted simplification algorithm, and finally constructing the resulting progressive mesh.

8.1. Restructuring of Mesh Data (Phase I)

This phase restructures the given data into a form that allows us to handle vertices and triangles of different types separately.

In the first step, we partition the vertex array into two sections, the first containing SN-vertices, the second containing NSN-vertices. For this, we have to implement two methods first:

- Inquiring vertex classifications. The method `isSNvertex(int v)` returns true, if all adjacent triangles of the vertex *v* have the same normal index at *v*.
- Permuting vertex indices. The method `swapVertices(int a, int b)` swaps two vertices and updates the index array, so that this swap does not change the mesh geometry.

In the next step, the index array is partitioned into SN-triangles, FN-triangles, and NSN-triangles. Here, swapping two triangles means to swap their indices and the

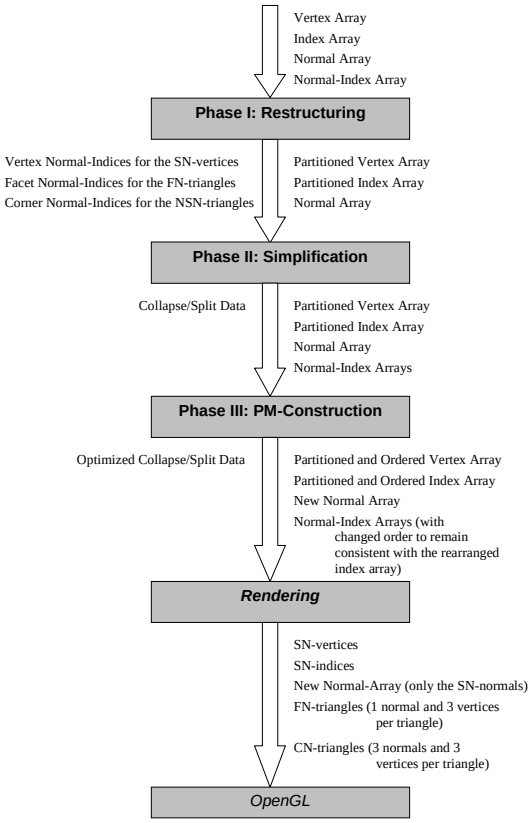


Figure 6. Process overview: The arrays on the left of each arrow has been newly created in the previous stage, the ones on the right has been sustained or changed.

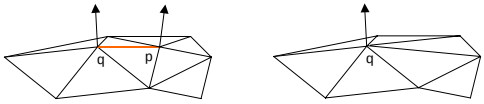


Figure 7. To avoid visual artifacts, smooth surfaces must remain smooth.

corresponding normal indices to keep both index arrays consistent. We start partitioning with swapping the SN-triangles to the front of the index array. 2) Having divided the index array into two sections – the first one containing the SN-triangles, the second one the remaining ones – we continue with the second section of the index array and divide it further into FN-triangles and NSN-triangles.

Now, based on the partitioned index array, we can replace the normal-index array by three new ones:

1) *SN-normal indices*. It contains exactly one index per SN-vertex.
2) *FN-normal indices*. It contains exactly one index per FN-triangle.
3) *NSN-normal indices*. It contains exactly one index per NSN-triangle corner.

We obtain these arrays as follows:

1) The normal index of an SN-vertex v is the corner normal-index of (v,T) for an arbitrary triangle T being adjacent to v.
2) The normal index of an FN-triangle T is the normal index of an arbitrary corner of T.
3) The normal indices for the NSN-triangles can simply be taken from the original normal-index array.

8.2. Processing Simplification Steps (Phase II)

In our algorithm, we use *pair contraction* [2] for mesh simplification. This represents a generalized version of the edge-collapsing method: Edge-collapsing corresponds to pair contraction restricted vertex pairs that represent edges. Contractions can be represented by a

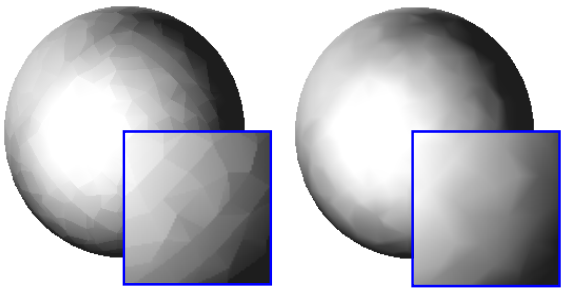


Figure 8. The sphere on the left is simplified without regarding rule a). The picture on the right shows the sphere in the same level of detail with preserved SN-vertices.

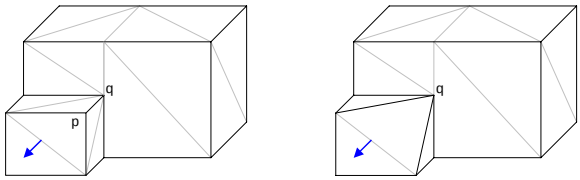


Figure 9. Face normals should not be changed due to a contraction.

data structure that consists conceptually of the following elements:

```
struct Contraction {
  Vertex p;
  Vertex q;
  List<Vertex> lv;
  List<Triangle> lt;
  List<Corner> ct;
};
```

For a given contraction c, vertex p is replaced by a vertex q, i.e., p gets lost. In *lv*, we store additional vertices that get lost, namely those vertices that exclusively belong to triangles containing both p and q. In *lt*, we store *lost triangles*, i.e., those triangles that contain both p and q. In *ct* we record *changed triangles*, i.e., triangles T containing p but not q. Thus, *ct* contains all corners (p,T) for which q is not a corner of T.

If we use optimal placement, i.e., moving q to a new position (which might not correspond to an existing vertex), we extend the contraction data-structure by a vector `delta` encoding the translation vector of q.

To avoid obvious visual artifacts in the resulting simplified mesh, two conditions must be met when replacing a vertex p by another vertex q:

a) If p and q are part of a smooth surface, this surface should remain smooth in the neighborhood of q, i.e., q should keep a shared vertex-normal. Otherwise, the resulting shading discontinuities would be more striking than the simplification itself (Figure 7, Figure 8).
b) The face normal of an FN-triangle should not be changed due to a contraction (Figure 9), as this would strengthen the visibility of the geometrical modification.

Since the handling of different normal types should not restrict the simplification process in the choice of a vertex pair to contract, we need to take into account the following configurations:

– Each of the vertices p and q can be of arbitrary type (SN or NSN).
– Each changed triangle can be of arbitrary type (SN, FN, or NSN). More exactly, there is one exception: If p is an NSN-vertex, a changed triangle cannot be of type SN.

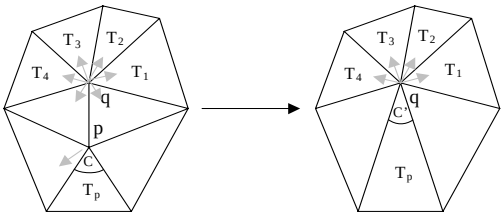


Figure 10. The problem: *q* is NSN-vertex and defines no own normal. All adjacent triangles define their own normals at *q*. The normal at corner *C* is implicitly defined via *p*. Which normal gets *C'*?

Particularly, we must ensure that we always have a well-defined normal for every corner of the mesh. Fortunately, most cases need not be handled explicitly:

(i) The normals of FN-triangles and NSN-triangles will not be affected because their normals are defined explicitly in the corresponding normal-index arrays.

(ii) If *p* is an NSN-vertex, there cannot be any SN-triangle adjacent to *p*.

(iii) If *p* and *q* are both SN-vertices, any changed SN-triangle gets the new normal at the changed corner implicitly via the index *q*.

Note that the conditions a) and b) are fulfilled because of properties (iii) and (i). The only problem is the case in which *p* is an SN-vertex and *q* an NSN-vertex. If there is any SN-triangle *T* that is changed but does not vanish during the contraction, the normal of the corner (*q, T*) will be undefined after the contraction (Figure 10, Figure 11). In this case we need to modify the contraction such that after its application the following properties hold:

I. The normal of (*q,T*) is well-defined for all changed triangles *T*.

II. The resulting mesh geometry is the same as it would be after the application of the unmodified contraction.

To meet condition I. we replace *q* by *p* instead of replacing *p* by *q*. If we use optimal placement, condition II. is already met by this modification, because it does not matter whether we replace *p* with *q* and move *q* to the new position or vice versa. If we use subset placement, we can also meet both conditions by swapping in addition the positions of *p* and *q* in the vertex array. In the case of subset placement, therefore, we need to extend our contraction data-structure by an additional Boolean value `swapVectors` to store whether the position vectors have been swapped due to the contraction.

8.3. Creating the PM Structure (Phase III)

After completing Phase 2 the mesh is now simplified to the *base mesh* [5]. We can reconstruct the original mesh by applying the inverse operations to the contractions, the *splits*. Those vertices and triangles that are not relevant for the current progressive mesh are called *inactive*.

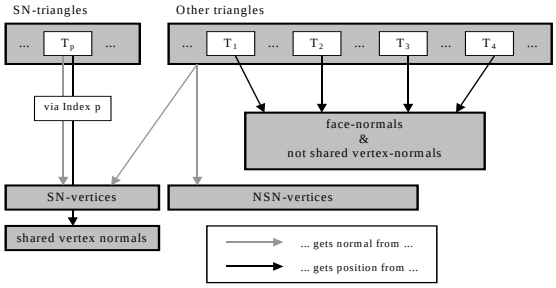


Figure 11. The situation of Figure 10. T_p gets its normal implicitly from the array of shared vertex-normals via index *p*. T_1-T_4 get their normals explicitly.

For a given contraction, we would waste memory if we would store lost triangles, lost vertices, and *p* explicitly. A better method is to arrange vertices and indices according to the contraction sequence, so that the inactive vertices (resp. triangles) can be found at the end of the vertex array (resp. index array) in every state of the progressive mesh [14]. Our algorithm adapts this idea: We subdivide each section into subsections for active and inactive elements (Figure 12).

To create the subsections, we traverse the pre-calculated contraction sequence replacing each contraction *c* by a new, more compact contraction *c´* and swapping vertices and triangles. The rewritten *c´* does not need to store the lists of lost vertices and lost triangles. Instead, it just has to store the numbers of lost vertices and lost triangles for each section. Note that vertex *p* is defined implicitly: prior to applying the contraction it is defined as the last active vertex; the section is defined by a Boolean value.

```
struct CompactContraction {
  Vertex q;
  bool pIsSNS;
  int NumberLostSNVertices;
  int NumberLostNSNVertices;
  int NumberLostSNTriangles;
  int NumberLostFNTriangles;
  int NumberLostSNSTriangles;
};
```

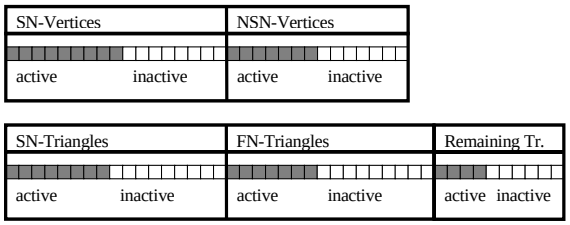


Figure 12. Arrangement of vertex array and index array.

In analogy to the contraction data-structure, this data structure contains an additional Boolean `swapVectors` (if we use subset placement) or an additional Vector `delta` (if we use optimal placement), respectively.

Maintaining Subsections. Throughout this traversal we maintain an integer value $N_{ActiveSNvertices}$ with the following property: At the beginning of each contraction rewrite the section of SN-vertices is divided into two subsections; all SN-vertices that have been lost during the previous contractions are stored in the second subsection, which begins at the position $N_{ActiveSNvertices}$ and ends at $N_{SNvertices}-1$. For the first contraction rewrite this condition can be obtained simply by initializing $N_{ActiveSNvertices}$ with $N_{SNvertices}$. In analogy, we maintain appropriate subsections of the NSN-vertices, SN-triangles, FN-triangles, and NSN-triangles.

Rewriting Contractions. We rewrite each contraction c by new contraction c' as follows: Swap vertex p with the last active one of the section containing p and decrease the size of the corresponding active subsection by one.

1. Store q explicitly.
2. Copy the `swapVectors` flag or the `delta` vector, respectively.
3. Store in which section of the vertex array p can be found (SN or NSN).
4. For any additional lost vertex proceed as in the first step.
5. Store the numbers of lost SN-vertices and lost NSN-vertices.
6. The lost triangles are handled in the same way as the lost vertices.

Vertex Swap and Triangle Swap. Note that we must keep the other data valid when we swap vertices or triangles. If we swap two vertices v_1 and v_2, the following steps take place:

Update the index array: Any element of the index array that has previously pointed to v_1 has to be set to v_2 and vice versa.

a) Update the contraction sequence: Any contraction for which p or q equals v_1 or v_2 has to be updated.
b) Update the lost vertices: If v_1 or v_2 vanishes during a certain contraction of the sequence, the corresponding contractions have to be updated.
c) If v_1 and v_2 are SN-vertices, swap the corresponding SN-normal indices

Similarly, when we swap two triangles T_1 and T_2, we have to do the following:

d) Update the index changes: Any contraction that replaces any vertex of T_1 or T_2 has to be updated.
e) Update the lost triangles: If T_1 or T_2 vanishes during a certain contraction of the sequence, the corresponding contractions have to be updated.
f) If T_1 or T_2 are FN-triangles or NSN-triangles, swap the corresponding normal indices.

Rebuilding the Normal Array. To render SN-triangles efficiently based on OpenGL's vertex array, the normal array must meet the condition that the normal of each SN-vertex v is located at position v in the normal array. So we have to rebuild the normal array. Since the normal-index arrays must contain the positions of the corresponding normals in the new normal array, we need to map an arbitrary normal index (i.e., a position of a normal in the old normal array) to the position of the corresponding normal in the new normal array. For this, we allocate an array of integers, called `newPosition`. Its size equals that of the old normal array.

The size of the new normal array is still unknown because we possibly have to duplicate normals that are shared by several SN-vertices. The size is given by the size of the old one plus the number of replicated normals. For this, we must count the number of replicated normals.

To rebuild the normal array, we allocate the new normal array and copy the normals for the SN-vertices to the first positions of the new normal array, i.e., we copy the normal of the i-th SN-vertex to the i-th position of the new normal array. Next, we assign the remaining normals to the remaining positions of the new normal array. Then, we are able to update the normal indices for FN-triangles and NSN-triangles by replacing each index `n` with `newPosition[n]`.

4. Case Study: 3D City Models

In 3D city models, most complex buildings are represented by polygonal meshes having both smooth and angular parts. LOD representations are essential for any interactive visualization due to the large number of buildings.

In a conventional implementation, progressive meshes would force us to base shading exclusively on vertex normals, which results in obvious shading artifacts. Smooth and angular parts within a single complex mesh need to be distinguished to achieve visually convincing results while having full LOD functionality (Figure 13). The only previously known way to achieve this would be to use wedges. However, in this case, we would be forced to store nearly the complete normal

information twice since the vast majority of triangles in our model are FN-triangles.

5. Conclusions

Our approach of an integrated treatment of smooth and angular parts of complex polygon LOD meshes extends the widely adopted progressive-mesh technique, offering the correct surface normals without storing redundant mesh information. As a possible extension, the algorithm can be extended to handle any information that occurs per-vertex and per-face in a similar way to vertex normals (e.g., per-vertex colors and per-face colors).

References

[1] P. Borodin, R. Klein, "Progressive Meshes with Controlled Topology Modifications", *Proceedings OpenSG Symposium*, 2002.

[2] M. Garland, *Quadric-Based Polygonal Surface Simplification*, Ph. D. Thesis, Carnegie Mellon University, 1999.

[3] T. S. Gieng, B. Hamann, K. I. Joy, G. L. Schussman, I. J. Trotts, "Smooth Hierarchical Surface Triangulations",.*IEEE Visualization Proceedings ´97*, 379-386, 1997.

[4] B. Hamann, "A Data Reduction Scheme for Triangulated Surfaces", *Computer Aided Geometric Design*, 197-214, 1994.

[5] H. Hoppe, "Progressive Meshes", *Computer Graphics (Proceedings SIGGRAPH ´96)*, 99-108, 1996.

[6] H. Hoppe, "Efficient Implementation of Progressive Meshes", *Computers & Graphics Vol. 22 No. 1*, 27-36, 1998.

[7] H. Hoppe, "New Quadric Metric for Simplifying Meshes with Appearance Attributes", *IEEE Visualization ´99 Proceedings*, 59-66, 1999.

[8] D. Luebke, "View-Dependent Simplification of Arbitrary Polygonal Environments", *Computer Graphics (Proceedings SIGGRPAH ´97)*, 199-208, 1997.

[9] J. Popovi'c, H. Hoppe, "Progressive Simplicial Complexes", *Computer Graphics (Proceedings SIGGRPAH ´97)*, 217-224, 1997.

[10] J. Rossignac, P. Borrel, "Multi-Resolution 3D Approximations for Rendering Complex Scenes", Technical. Report RC 17687-77951. IBM Research Division, T. J. Watson Research Center. Yorktown Heights, NY 10958, 1992.

[11] C. Prince, *Progressive Meshes for Large Models of Arbitrary Topology*, M.S. thesis, University of Washington, 2000.

[12] W. J. Schroeder, J. A. Zarge, W. E. Lorensen, "Decimation of Triangles Meshes", *Computer Graphics (Proceedings SIGGRAPH ´92)*, 65-70, 1992.

[13] W. J. Schroeder, "A Topology Modifying Progressive Decimation Algorithm", *IEEE Visualization Proceedings´ 97*, 205-212, 1997.

[14] J. Svarovsky, "View-Independent Progressive Meshing", *Game Programming Gems*, Charles River Media, 454-464, 2000.

[15] G. Turk, "Re-Tiling Polygon Surfaces", *Computer Graphics (Proceedings SIGGRPAH '92)*, 55-64, 1992.

[16] P. Sander, J. Snyder, S. Gortler, H. Hoppe, "Texture Mapping Progressive Meshes", *Computer Graphics (Proceedings SIGGRAPH 2001)*, 409-416, 2001.

[17] J. Wu, L. Kobbelt, „Fast Mesh Decimation by Multiple-Choice Techniques", *Vision, Modeling and Visualization*, Erlangen, 2002.

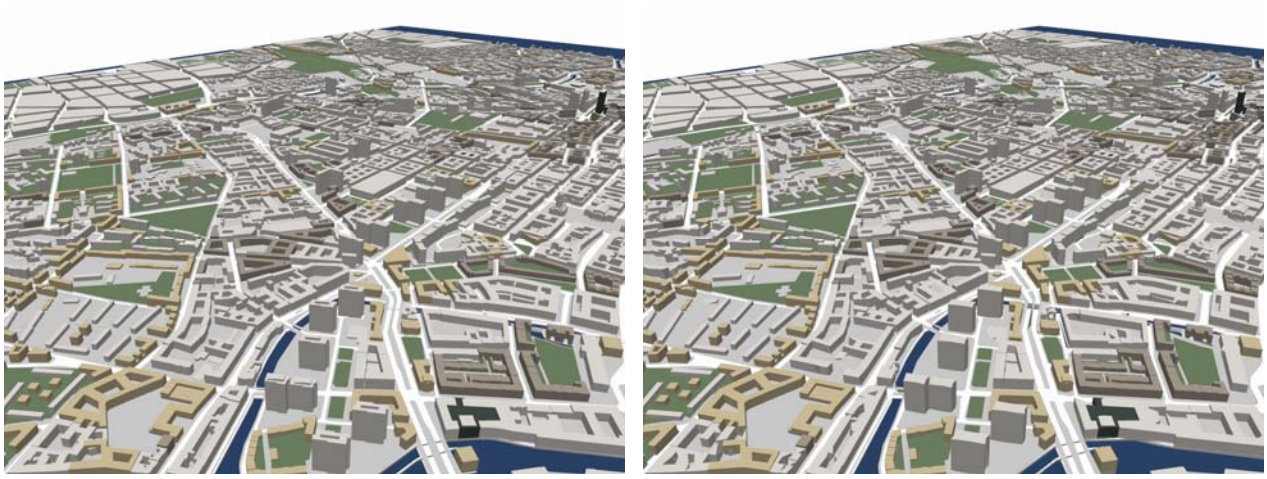


Figure 13. Part of the 3D city model of Berlin. LOD representation with ≈960.000 triangles (left) and ≈200.000 triangles (right).