PERGAMON

Computers & Graphics 26 (2002) 491–503

COMPUTERS & GRAPHICS

www.elsevier.com/locate/cag

Technical Section

Visualizer: a mesh visualization system using view-dependent refinement

Uğur Güdükbay[a,*], Okan Arıkan[b], Bülent Özgüç[a]

[a] *Department of Computer Engineering, Bilkent University, 06533 Bilkent, Ankara, Turkey*

[b] *Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, 387 Soda Hall 1776, Berkeley, CA 94720-1776, USA*

Abstract

Arbitrary triangle mesh is a collection of 3D triangles without any shape or boundary restrictions. Progressive mesh (PM) is a multiresolution representation that defines continuous level of detail approximations for arbitrary triangle meshes. PM representation of a mesh can be processed to obtain a mesh approximation between the original and the base (simplified) mesh. Furthermore, PM can be refined in a view-dependent fashion to obtain a simpler mesh within a perceptual image quality. In this paper, we introduce an adaptation and improvements in our implementation for view-dependent refinement of progressive meshes. Essentially, we use a similar approach to Hoppe's framework (ACM Comput. Graphics, Proceedings of SIGGRAPH'97, August 1997, pp. 189–198) for view-dependent refinement with a different algorithm for constructing PM representation. Our method is simple to implement and fast enough to achieve interactive frame rates for moderately complex models (models containing hundreds of thousands of polygons) on a machine with polygon rendering hardware. Moreover, our implementation allows changes to topology and achieves a simpler and sometimes more realistic refinements. © 2002 Elsevier Science Ltd. All rights reserved.

Keywords: View-dependent refinement; Multiresolution modeling; Progressive mesh; Mesh visualization

1. Introduction

Models in computer graphics are usually created using 3D scanners or manually. 3D models are usually converted into a polygon mesh, which is just a collection of planar polygons (generally triangles) that are used to approximate the surface. Linear structure of polygons make them especially suitable for visualization and processing to be done on the geometry. Since polygon rendering is usually implemented at hardware level, it is faster when compared to rendering other kinds of surfaces.

Sometimes, only one mesh representation of a model can bring unnecessary processing burden to the renderer. If the model to render is too far away from the camera and covers only a couple of pixels on the screen, a much coarser representation can be substituted for performance. Using multiple representations of the same model with different detail levels at different contexts is a common practice called *multiresolution modeling* [1]. However, switching between multiresolution models can create visual artifacts on continuous scenes. Moreover, this technique requires a lot of memory for mesh storage.

To obtain multiresolution representations of the models, mesh simplification algorithms could be used. The formal definition of mesh simplification is decreasing the number of vertices and faces without compromising much from the overall geometry. There are many mesh simplification algorithms proposed. The simplification algorithms are based on *vertex decimation* [2,3], *vertex clustering* [4], *iterative edge contraction* [5–7], and *sampling and re-tiling* [8]. A comparative survey of mesh simplification algorithms can be found in [9]. Among these, algorithms based on iterative edge contraction are more popular and produce good quality

*Corresponding author.
E-mail address: gudukbay@cs.bilkent.edu.tr (U. Güdükbay).

approximations. These algorithms produce a simpler mesh by continuously selecting an edge and collapsing it to a vertex. At each step, the faces and vertices adjacent to the selected edge is removed from the mesh, a new vertex is inserted and connectivity information is updated accordingly (Fig. 1). We can undo the process by replacing the newborn vertex with the removed edge by an operation called vertex split. Thus, if the edge to be decimated is an interior edge (has two adjacent faces), at each step, we delete 2 faces and 1 vertex.

A polygon mesh in the rendering pipeline can also suffer from unnecessary processing. Some parts of the mesh can lie outside the viewing frustum or look away from the viewer. Therefore, these parts do not contribute to the final image (Fig. 2). Besides, some parts of the mesh can be closer to the camera than others, needing to be more detailed.

1.1. Previous work

Progressive mesh (PM) representation of a mesh is defined to be a base (simple) mesh and a series of vertex split operations which, when applied in order, lead to the original mesh [10]. These vertex split operations can be performed in a selective manner to obtain the view-dependent refinement of a mesh in which the detail level is dynamically controlled. In this way, a version of the mesh, which is more detailed only on the parts the viewer pays attention, can be obtained. Moreover, this refinement can be done quite fast. In other words, for dynamic view-dependent visualization of complex polygonal environments, selective refinement algorithms are proposed that can simplify certain parts relatively more than the remaining parts to capture a perceptual quality.

Xia and Varshney [11] use edge collapse and vertex split transformations to create a simplification hierarchy allowing real-time selective refinement. They construct a merge tree in a preprocessing step. They constrain the merge tree hierarchy to a set of levels with non-overlapping transformations and store additional dependencies together with the hierarchy to enforce the constraints. Although the main reason for using the dependencies is to prevent folding artifacts, these dependencies also constrain the refinement process,

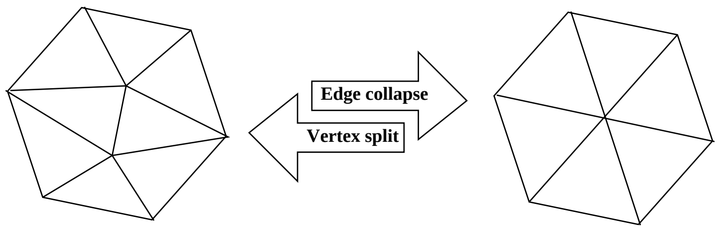

Fig. 1. Edge decimation.

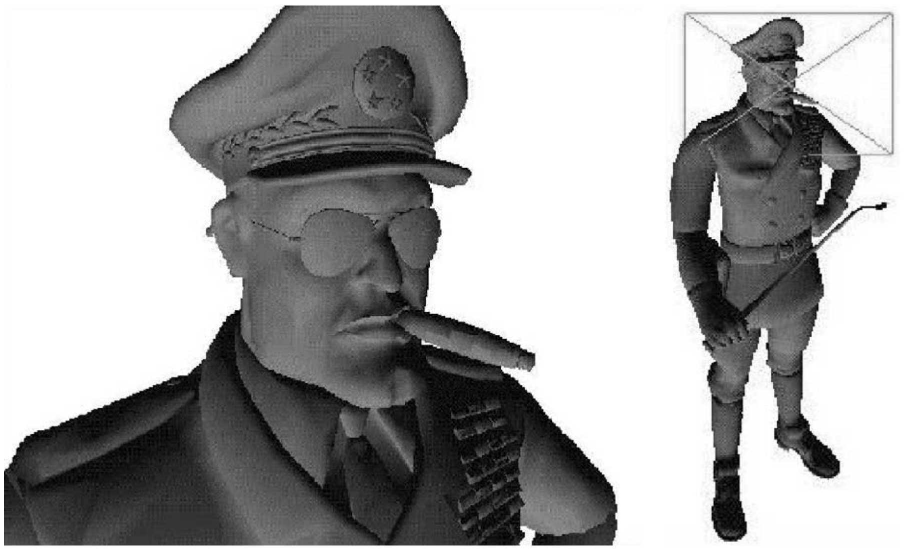

Fig. 2. A general model with some parts outside the view which causes redundant processing.

limiting the degree of drastic simplification. The method only allows gradual changes from regions of high refinement to those of low refinement.

Luebke and Erikson [12] propose a method called hierarchical dynamic simplification (HDS) for dynamic view-dependent simplification. Their method works by clustering vertices in a hierarchical fashion. It uses a vertex tree structure and continuously queries the structure to generate a scene according to the current viewpoint. The vertex tree structure is constructed in the preprocessing step. The method works on non-manifold meshes and may change the topology of the models. An active list of visible polygons are maintained for rendering. When the area occupied by a vertex cluster on the screen is below a user-specified threshold, the vertices in the cluster are collapsed into a single vertex and the triangulation is updated accordingly. The method utilizes frame-to-frame coherence for greater speed. However, they state that their implementation runs with adequate speed on small models, containing no larger than 20,000 triangles. They claim the optimizations they proposed, which are exploiting temporal coherence, using visibility information, and parallelizing the algorithm, increase the speed by almost two orders of magnitude, but it is still too slow to be used for complex models in interactive applications.

Hoppe's framework [13] essentially uses a PM structure and checks three criteria for view-dependent refinement as described in the following sections. Since Hoppe restricts edge collapses to those that preserve the manifold topology of the mesh, the possible amount of simplification is limited. We essentially use Hoppe's view-dependent framework, but construct the PM representation using a simplification method that allows changes to the topology of the mesh. Possible changes to the topology include joining disconnected parts during simplification and eliminating the disconnected parts that are completely outside the viewing frustum.

2. View-dependent refinement framework

In this section we review the techniques that we used in our view-dependent refinement framework, namely the simplification algorithm for constructing PM representation, PM and selective refinement techniques.

2.1. Simplification by quadric error metrics

In our implementation, we use Garland and Heckbert's mesh simplification algorithm [5]. Their method proceeds by successively collapsing pairs of vertices into one by a process, called "vertex pair contraction". Since the vertex pair in question is not required to have an edge in between, their algorithm is not confined to edges. Therefore, this method may change the topology of the mesh. As a result, parts of the mesh can be totally decimated or two unconnected parts can merge together on meshes composed of unconnected sub-meshes (see Fig. 3).

2.2. Progressive meshes

One way to obtain a simpler mesh is to apply successive vertex pair collapse operations. Each vertex pair collapse operation is characterized by removal of a vertex pair (note that it is not necessary to have an edge between these vertices) and associated faces and introducing a vertex instead (Fig. 1). At each vertex pair collapse operation, mesh is simplified by 2 faces (triangles) and 1 vertex in general (in case of a border edge, only 1 face is deleted). Successive application of vertex pair collapses yields a simpler mesh

$$M^n \overset{pcol_1}{\rightarrow} M^{n-1} \overset{pcol_2}{\rightarrow} M^{n-2} \overset{pcol_3}{\rightarrow} \cdots \overset{pcol_m}{\rightarrow} M^0.$$

Each vertex pair collapse operation has a so-called inverse vertex split. Just as applying vertex pair collapses on original mesh leads to a simpler mesh, applying vertex split operations on the simple (base) mesh in the inverse order of their respective vertex pair collapses, leads to

$$M^0 \overset{vsplit_1}{\rightarrow} M^1 \overset{vsplit_2}{\rightarrow} M^2 \overset{vsplit_3}{\rightarrow} \cdots \overset{vsplit_m}{\rightarrow} M^n.$$

Thus, the base mesh with the vertex split records form PM representation [10]. PM representation has some advantages over the conventional meshes. First, continuous LOD approximations can be built by traversing and applying the vertex split records in order. The

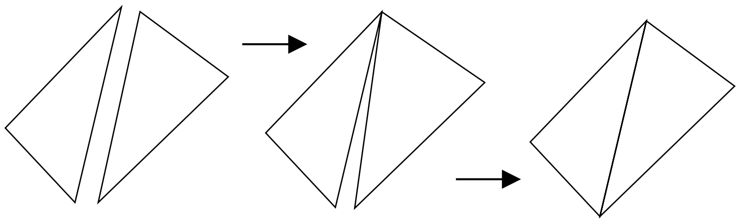
Fig. 3. Vertex pair contractions leading to polygon merge.

uniform nature of edge collapses applied during the simplification provides gradual global increase of detail in the progress. The number of vertex split records applied determines how detailed or how close to the original mesh that the resulting LOD representation will be. In addition, vertex split records do not necessarily be traversed consecutively. The level of detail can be concentrated at some parts of the mesh by selecting the vertex split operations to be performed. By careful manipulation of data structures, progressive meshes also provide an efficient compression scheme and progressive transmission by definition.

2.3. Selective refinement

The structure of PM is essentially represented as a base mesh and a series of vertex split operations. Some of these vertex split operations are applied to obtain a mesh representation with desired detail level. For example, an intermediate representation between the original and simplest mesh can be obtained by applying only first half of the vertex split records in order.

While rendering a mesh, polygons lying outside the viewing frustum and those that look away from us do not contribute to the final image. Thus, they can be sorted out to increase performance. Moreover, some parts of a mesh may be much closer to the screen than others (especially in a terrain flythrough). These close parts should be represented in more detail.

By selectively applying these vertex split records, we can increase the level of detail on parts that we want to see more detailed without visual discontinuities. For example, dividing the mesh into different parts and visualizing each part in a different level of detail would result in a similar view-dependent refinement for terrain models. However, such methods may create gaps at the boundaries and lead to unsatisfactory results [14].

The vertex split operations to be performed are selected by a refinement function and must satisfy certain preconditions to become legal. As illustrated by Hoppe's terms [13], to use a vertex in a vertex split operation the vertex must be an active vertex and all the neighboring faces of two yet to be introduced faces must be active before the vertex split operation. Active vertices are the vertices that are used in the current refinement of the mesh. In Fig. 4, the configuration for a valid vertex split operation is shown. Here, V_s is the vertex to be split, and $(f_{n0}, \ldots, f_{n3})$ are the neighboring faces of two yet to be introduced faces after the vertex split operation.

The refinement function determines whether a vertex needs to be split or not by comparing the split record against three criteria. These can be summarized as follows [13]:

Viewing frustum: If a vertex with all its descendants in our vertex hierarchy lie outside the viewing frustum, then splitting that vertex does not contribute to the final image. This can easily be done by comparing the sphere whose center is the vertex and contains all the descendants in it against six frustum planes. If the sphere is outside the frustum pyramid, we should not split.

Surface orientation: If a vertex with all its descendants in our vertex hierarchy looks away from the eye, then splitting that vertex does not contribute to the final image either. This can be computed by constructing a cone of normals whose central axis is the normal of the vertex being considered and the cone angle capturing the deviation in normals of all the descendants.

Screen space error: If a vertex split operation causes more than a certain deviation when projected on the screen, then applying the operation causes unexpected geometry changes on the model [14]. Thus, by dynamically changing the deviation tolerance we can control the detail level or the maximum error tolerable on the screen. This criterion also takes the distance to the eye into account.

Here, one could argue that the view-frustum culling process eliminates the faces outside the view-frustum and the back-face culling process eliminates the faces that look away from the viewer at the earlier stages of the rendering pipeline. However, these culling processes will be done much more efficiently, thereby reducing the graphics load, if a model is represented using lower levels-of-detail at these parts.

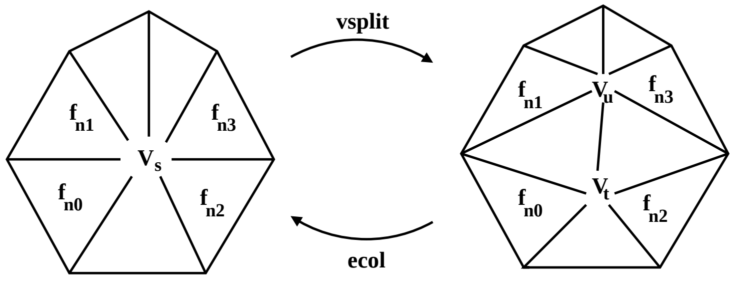

Fig. 4. Preconditions for a vertex split to be selected for refinement.

The refine function could be modified to satisfy different needs. For example, a collision detection system might want to decrease the number of polygons to check, increasing the detail level on possibly colliding parts while keeping the mesh quite coarse on other parts.

3. Our improvements

3.1. Topology compromise

To create the necessary PM representation, we use simplification by quadric error metrics, which is essentially a vertex pair collapse method. Since the method does not necessarily need to have an edge between the vertex pairs to collapse them, it can merge or totally decimate unconnected parts of the original mesh and compromise the topology leading to a more compact representation. These features become quite handy for the view-dependent refinement of meshes. For example, if the mesh is composed of several unconnected sub-meshes, then those lying outside the viewing frustum can be totally omitted.[1] Since the changes to topology are allowed in our view-dependent framework, the models that can be visualized are not restricted to 2-manifolds. The framework transforms general manifolds to general manifolds. That's why it can work on any arbitrary triangle mesh without going through complex geometry checks.

Throughout the paper, we will refer to pair collapses (`pcol`) instead of edge collapses that are inverse of vertex splits to avoid ambiguity. In order to accommodate these pair collapse operations into our PM representation, we start with the original mesh and traverse our `pcol` records in order.

3.2. Inverse traversal

In our implementation, we have taken our PM representation as the original mesh and a series of pair collapse operations. As a logical consequence, our view-dependent refinement framework takes the original mesh and decreases the detail level where appropriate instead of taking the simple mesh and increasing the detail level. Thus, our resulting PM representation takes more space to store but using it for view-dependent refinement provides some advantages. First, our traversal method is much simpler to implement as discussed in the sequel and fast enough to achieve interactive frame rates, mainly because of the simplicity of the implementation.

In Hoppe's view-dependent refinement method [10], two new vertices are introduced and one is deleted at each vertex split operation. Thus, the refine function must iterate over the active vertex list, which contains the vertices used in the current refinement, and possibly consider some vertices more than once executing unnecessary refine functions [13]. In addition, the traversal function tends to be more complex because it requires to check the preconditions stated in Section 2.3. Furthermore, the vertex split and face records need to hold some additional information for these preconditions. Specifically, Hoppe's implementation of the vsplit and ecol transformations requires adjacencies between elements of the mesh. Thus, for each face, pointers are stored to its neighboring faces [16]. Our traversal algorithm can infer the connectivity information from the configuration before applying an edge (vertex pair) collapse. That is why it can work on arbitrary triangle meshes without going through complex checks to understand the neighborhood formation. For example, the mesh in Fig. 5 can be handled without any auxiliary data structures or processing as opposed to the previous methods. We should mention that for cases where there is a large reduction of faces due to limited screen space resolution, coarse-to-fine refinement might be preferable since adding detail to the coarse mesh is easier. For cases where the reduction of faces is not so drastic, fine-to-coarse refinement (inverse traversal) would be preferable because of the reasons explained.

We sequentially iterate over the vertex pair collapse records, which can be taught as inverse of vertex splits. At each step, if the vertices involved in the collapse operation are active then we execute the refine function to determine whether the collapse should be done or not. We delete two vertices and introduce a new one at each performed `pcol` operation. Since we start with original mesh and decrease the detail instead of starting with simple mesh and adding detail, our refine function is in fact just the opposite of that mentioned in Section 2.3. It considers the vertex pairs used in each `pcol` operation and returns `True` for the pairs that should be collapsed.

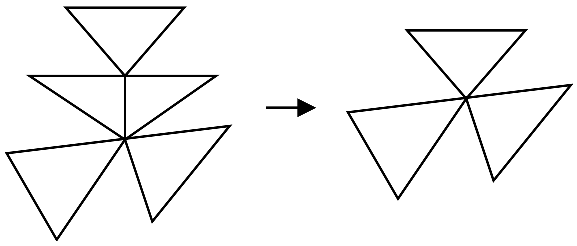
Fig. 5. A collapse that lacks the necessary neighborhood formation.

[1] Popović and Hoppe described "progressive simplicial complex" representation that achieves better fidelity approximations by allowing changes to the topology of the models [15]. However, this representation is not used in a view-dependent framework.

```
procedure view_dependent_refine()
    for each i in  {1 .. number_of_ecols}
        if  (pcol[i].v1.active) and (pcol[i].v2.active)
            if refine(pcol[i])
               Execute(pcol[i])
```

Fig. 6. View-dependent refinement algorithm.

Since we check pcol operations for refinement and the number of original vertices is too high, we store the precalculated data for refinement in pcol records. The pseudo-code for our view-dependent refinement algorithm is given in Fig. 6.

Notice that we do all the edge (vertex pair) collapse operations in the worst case. Also, we do not check the neighboring face data. Moreover, since we do not need to determine which faces the newborn vertices goes into as in the case of refinement with vertex split operations, the construction of the final mesh can be done quite fast.

4. Implementation details

In this section, we present the data structures and some implementation details to perform view-dependent refinement of PM efficiently. The overall structure of the system, called *Visualizer*, is given in Fig. 7.

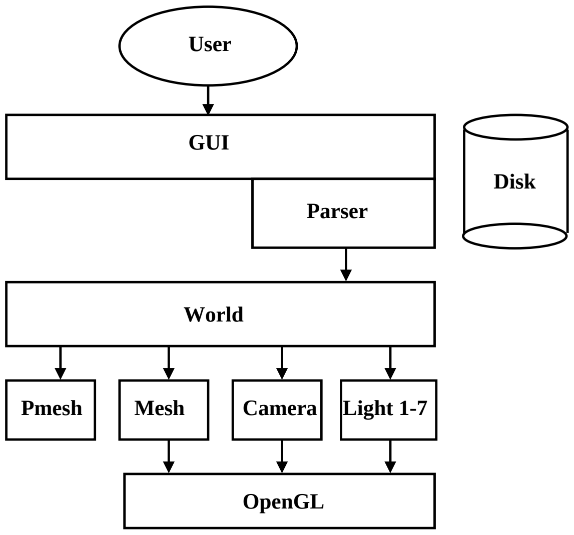

Fig. 7. Overall structure of *Visualizer*.

4.1. Data structures

The data structures are given in Fig. 8. The data structures are in a C++-like language format and some details have been omitted for simplicity and clarity.

PM representation of a model is produced as a preprocessing step and written into a file. When the *Visualizer* runs, it reads this file and produces a forest of binary vertex trees to store the PM representation in a hierarchic structure. The vertices at the roots of the trees correspond to the vertices in the simple mesh. Each vertex split operation denotes a branch in this tree such that if v_l and v_k are the children of v_n, then the removal of edge between them leads to v_n. Note that with this terminology, vertices that are at the leaves of the trees represent the vertices in the original mesh.

Our PM representation consists of the original mesh and a series of pair collapse operations. In order to decrease run-time memory movement and complexity, we also store the vertices created by pair collapses in our base mesh and refer to them by indices.

4.2. View-dependent refinement operations

In order to do view-dependent refinement, the following operations are done. First, the active flags of all the vertices and faces in the original mesh are marked (notice that these flags will change during the refinement). Then, the pcol records are sequentially traversed and the collapse operations that need to be performed are determined. For the maximum performance, we do not actually perform the collapse operations on the faces at the time a pcol record is designated to be executed. Instead of this, we simply clear the active flags of decimated vertices and faces and set the active flag of newborn vertices. Once the vertices and faces that will be present in the refined mesh are determined, we start updating the vertices of final faces.

In order to update the face information, a two-phase referencing scheme is used. In the mesh structure, v field of a face is used to reference into a vertex reference (vref) array. Once the final vertices are determined, the references in our vref array are changed using the children field, which keeps the vertex hierarchy. With the vref array, we do not need to do any modification, except changing the active flags of faces. For example, if a pcol collapses vertices 1 and 2 into vertex 3, then the indices at locations 1 and 2 in vref array will be changed by 3. Of course, vref array should be used to reference actual vertices at rendering time. Thus, we save a great

```
class vertex {                  // simple 3d vertex def.
    vector  coordinates;
    vector  normal;
    int     children;           // index to children
                                // left child = children
                                // right child = children+1
    boolean active;             // vertex active or not
}

class face {
    int     v[3];               // indices to the vertices
                                // of the triangle
    boolean active;             // face active or not
};

class mesh {
    vertex  vertices[num_vertices]; // vertices of the mesh
    face    faces[num_faces];       // faces of the mesh
    int     num_vertices;           // number of vertices
    int     num_faces;              // number of faces
};

class pcol {
    int     v1,v2;              // collapsed pair of vertices
    int     vf;                 // newborn vertex
    float   r,a;                // r is the radius of the sphere used
                                // to check against viewing frustum,
                                // a is the sinus of the angle that records
                                // the deviation along the normal axis of vf.
    int     num_decimated;      // number of decimated faces
                                // with this collapse
    int     f[num_decimated]; // array of decimated faces
};

class pmesh {
    mesh    original;           // original mesh
    pcol    pcols[num_pcols]; // array of pcols
    int     num_pcols;          // number of pair collapses
    int     num_bvertices;      // number of vertices in simple mesh
    int     num_bfaces;         // number of faces in simple mesh
};
```

Fig. 8. Primary data structures for the PM representation.

deal of time instead of going to the face record for each vertex and update the faces that it appears.

5. Results

In this section, still images of view-dependent mesh visualizations are presented (a set of animated view-dependent mesh visualizations in different movie formats can be found in [17]).

In these examples, the rectangular areas or pyramids containing only a part of the models describe position of the viewing frustum for the phantom camera spawned by the user. This is done to demonstrate the view-dependent refinement process. If the user does not spawn a phantom camera, the refinement process is

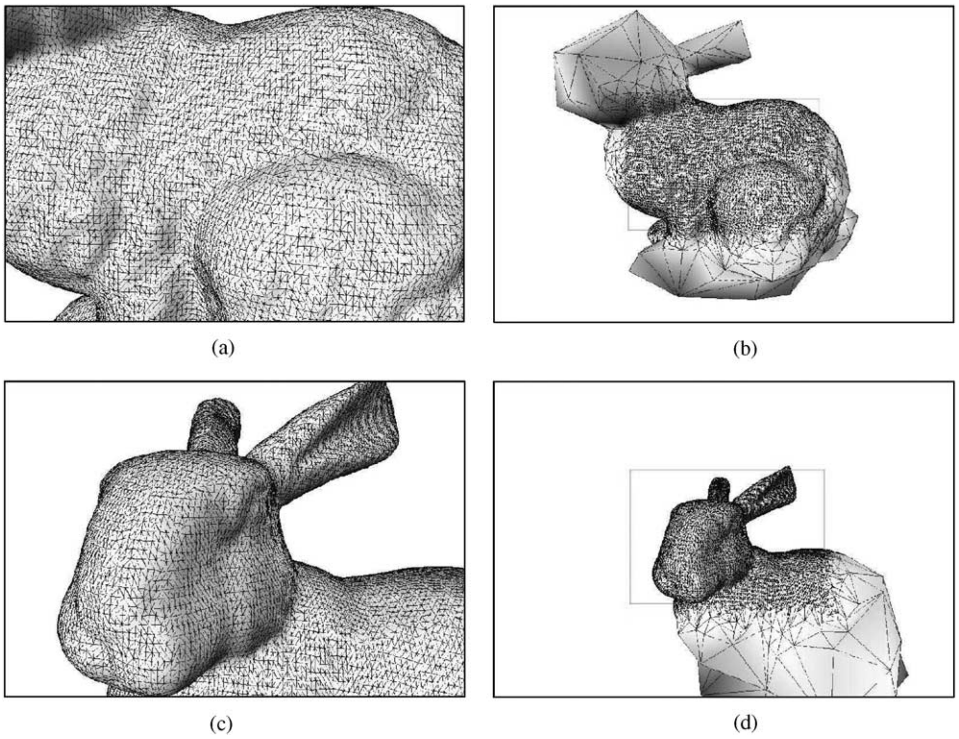

Fig. 9. Effect of viewing frustum in view-dependent refinement.

done with respect to the normal camera whose image plane is the whole window.

Fig. 9 shows the effect of viewing frustum culling in view-dependent refinement. Notice how the parts lying outside the viewing frustum are rendered at a lower detail. The original bunny model has approximately 70,000 triangles. The refinement in (a) and (b) has 32,558 triangles, and the one in (c) and (d) has 32,330 triangles. It should be noted that since only the parts lying outside the viewing frustum are refined, the resulting mesh appears to be the same as the original model when projected onto the screen.

Fig. 10 shows the effect of surface orientation in view-dependent refinement. Notice how the parts looking away from the view is refined to a lower detail. This refinement decreases the number of triangles from 70,000 to 28,949. Notice that the resulting mesh is not different from the original model when projected onto the screen (see Fig. 10(a)).

Fig. 11 shows the effect of screen space error tolerance on the view-dependent refinement. The meshes in (a), (b) and (c) have 1657, 7385 and 13,292 triangles, respectively. Notice that the silhouette edges are highly detailed because the screen space error criterion is more sensitive in these parts.

Fig. 12 demonstrates the operation of *Visualizer* on "Happy Buddha" model. The original model has about 1,088,000 triangles. Although it takes minutes to load the PM representation of such a fairly large model to memory, the user can interactively manipulate the model without trouble after the loading process is completed. When a model does not fit into the main memory, the interaction with the model will be much slower due to the swapping operations between the main memory and the disk.

Fig. 13 demonstrates the effect of topology compromise in simplification. The mesh in (a) leads to one in (b) when simplified by a topology-preserving algorithm because the mesh consists of four unconnected sub-meshes. However, our implementation can handle such non-topology preserving simplifications that lead to the mesh in (c) without trouble and sometimes achieve a better refinement. For example, if the input mesh consists of several unconnected sub-meshes then our implementation can totally decimate those sub-meshes that do not contribute to the final image.

6. Performance

In Table 1 statistics for some data sets and interactive frame rates for the view-dependent refinement of these data sets are given. Interactive frame rates are the averages taken during a 10-s period and the numbers of

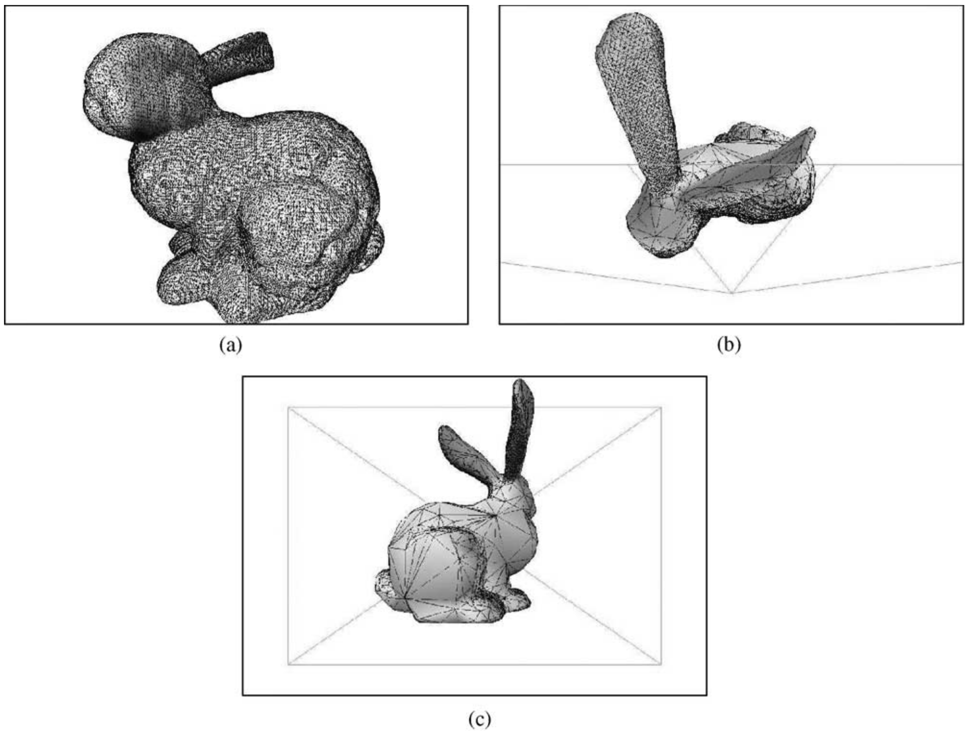

Fig. 10. Effect of surface orientation in view-dependent refinement.

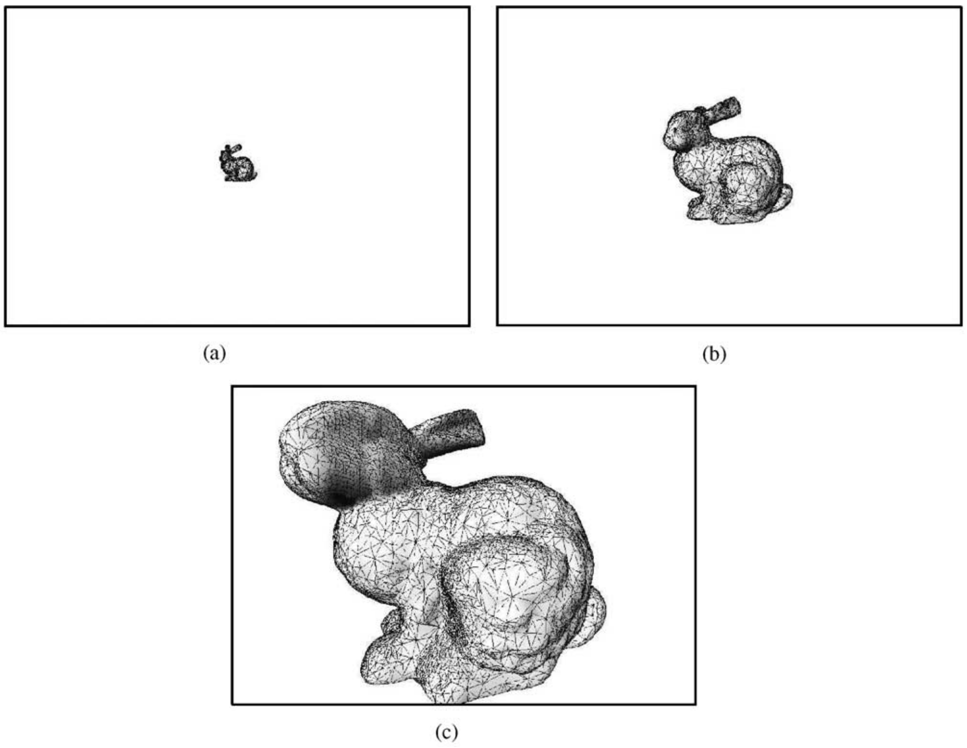

Fig. 11. Effect of screen space error tolerance in view-dependent refinement.

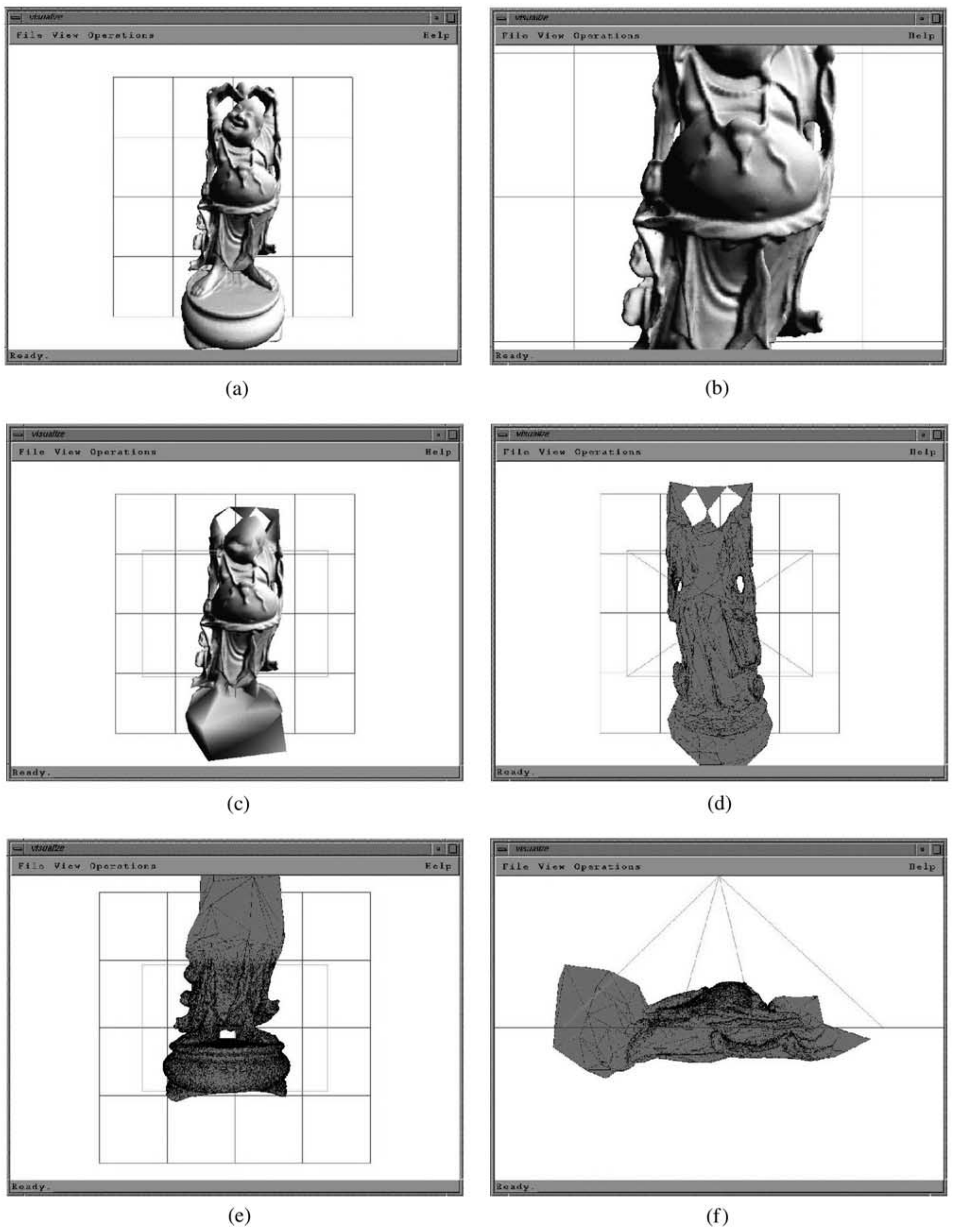

Fig. 12. Operation of *Visualizer* on "Happy Buddha" model.

faces and vertices in the simplified models are taken as typical snapshot values during that period. In the experiments, the view point is selected to see the whole model and the model is displayed as large as possible to fit the display window of size 600×800 pixels. For the view points where the model covers only a small portion of the display window, frame rates increase accordingly. Different view-dependent visualization parameters such as screen-space error threshold may result in different frame rates. The values of these parameters are tuned to achieve a reasonable image quality. The models are rendered using Gouraud shading with a single active light source. The experiments are performed on SGI Octane with 128 MB of memory and 250 MHz MIPS R10000 processor. The PM representation is constructed by an offline process, namely the `qslim` [5] program that records the vertex pair collapse transformations and a converter program that writes these records to a file in

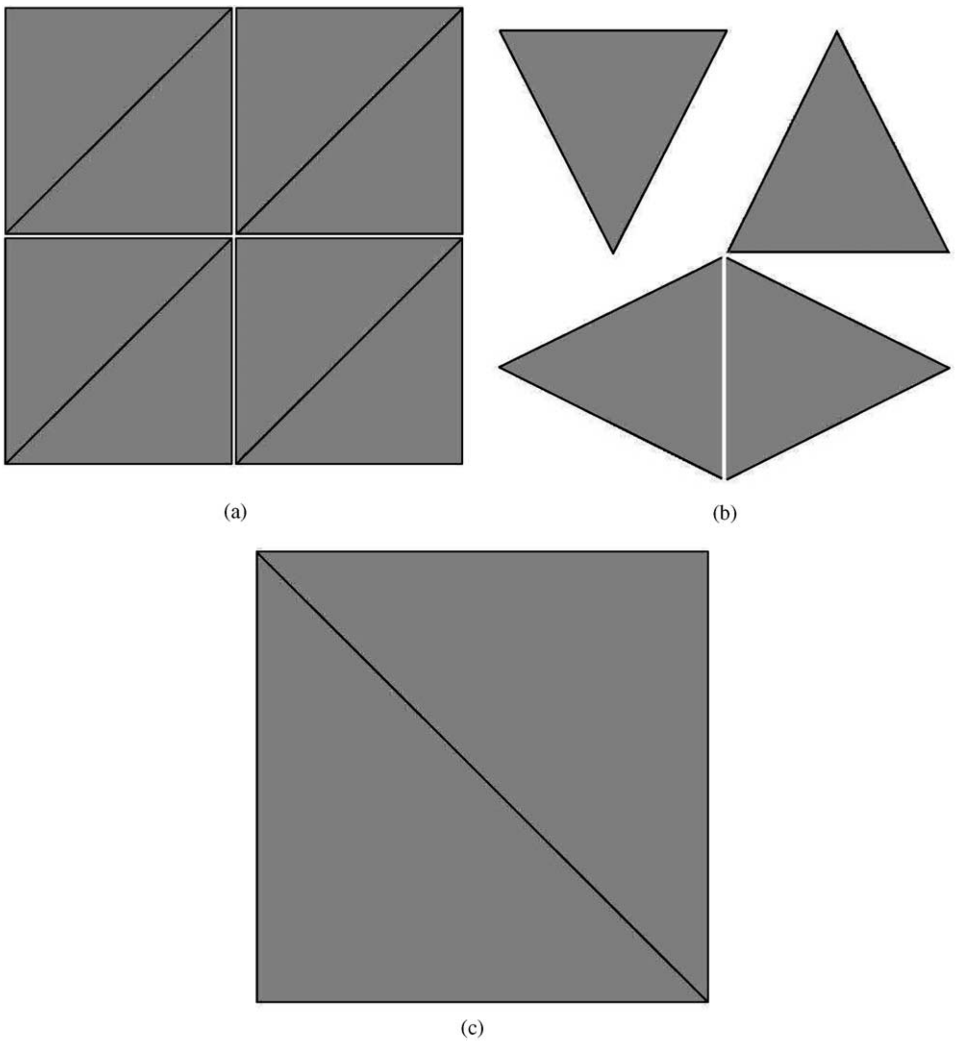

Fig. 13. The effect of topology compromise in simplification.

Table 1
Statistics about some data sets and interactive frame rates for the view-dependent visualizations

Data set	Original model			View-dependent refined model		Reduction ratio (%)	Frames per second
	No. vertices	No. faces	No. pcols	No. vertices	No. faces		
Cow	5773	5804	2869	930	1856	28.5	43.9
Boat	12,198	11,534	6015	1416	2412	20.9	27.8
General	24,385	22,262	11,828	2289	3011	13.5	15.3
Flamingo	25,557	25,080	12,671	2142	3822	15.2	15.0
Bunny	70,728	69,451	34,781	3459	4627	6.7	10.3
H. Buddha I	289,210	293,232	144,563	5297	10,626	3.6	2.5
Dragon	874,615	871,414	436,970	6807	13,759	1.6	0.9
H. Buddha II	1,087,123	1,087,716	543,471	7340	16,494	1.5	0.7

our system's PM format. This process is performed on a machine with larger main memory for very large models. When the program runs, the PM representation is loaded into memory in a few seconds for small models (models containing up to a hundred thousand faces) and a few minutes for very large models (models containing a million faces). PM construction, which is an offline process, takes several hours for Happy Buddha model containing 1,087,716 faces.

The results in Table 1 show that our view-dependent refinement implementation is fast enough to be used with moderately complex models in interactive applications. Since different hardware is used, a direct performance comparison with the results presented in the most notable work of Hoppe [13] would be misleading. Hoppe used an SGI Indigo2 Extreme (150 MHz R4400 with 128 MB of memory) for his experiments. However, the interactive frame rates for the experiments we performed are comparable to those presented in [13]. For example, Hoppe's implementation achieved 14.7 frames per second (fps) for the teapot model (original model has 10,000 faces and simplified version has 1782 faces). Our implementation achieved 27.8 fps for a boat model of similar size (original model has 11,534 faces and simplified model has 2412 faces). For the bunny model, Hoppe's implementation achieved an interactive frame rate of 6.7 fps (original model has 69,473 faces and simplified version has 10,528 faces) whereas our implementation achieved 10.30 fps (original model has 69,451 faces and simplified version has 4627 faces). For a larger terrain model (Grand Canyon with 400,000 faces), his implementation achieves a frame rate of 7.2 fps. Approximately 9000 faces are displayed at each frame. As a comparable model, we experimented with a simplified version of Happy Buddha model (the model has 293,232 faces) and achieved a frame rate of 2.5 fps. It should be noted that the simplified version that is rendered in our experiment has 10,626 faces. Hoppe's implementation achieves better frame rates for larger models. The reason for this performance difference for larger models is that his implementation utilizes frame-to-frame coherence and amortization, and uses generalized triangle strips for rendering. Frame rates of our implementation could be increased by utilizing frame-to-frame coherence and amortization, and by using generalized triangle strips for rendering, as described in [13]. Besides, since we store the original model and pair collapse transformations for the progressive mesh representation, the memory requirement of our implementation is high and disk swapping operations degrades the performance for larger models.

We also would like to emphasize that although we achieved comparable results with Hoppe's implementation for small or moderately complex models, the main difference between his implementation and ours is that our implementation allows topological changes to the models during view-dependent simplification because of the nature of the simplification algorithm that we used to construct the PM representation. This is a desirable property when visualizing complex models composed of a number of unconnected parts. These unconnected parts may be joined during simplification, which produces more compact and realistic simplified models.

If we examine the face reduction ratios for different models in Table 1, the number of faces has been decreased to 28.5% compared to the original model for the smallest model that we tested (cow model). However, this ratio is approximately 1.5% for the largest model we tested (Happy Buddha). The reduction ratio depends on refinement parameters, like screen-space error threshold. For cases where there is a large reduction of faces due to limited screen space resolution, coarse-to-fine refinement might be preferable.

Regarding the space consumption of our view-dependent refinement framework, since the original mesh and the vertex pair collapse records for the PM representation should be loaded to the memory before a view-dependent visualization begins, the memory requirements can be high.

7. Summary and future work

We described and implemented a framework for efficient implementation of view-dependent refinement of PM. The implementation is simple and fast enough to achieve interactive frame rates for moderately complex models. Moreover, its ability to work on non-edge collapses makes it especially suitable for refinement of models composed of unconnected parts. Since the implementation allows changes to the topology of the models and it is not confined into manifold surfaces, some parts of the mesh can disappear or merge to the other parts throughout the refinement process. However, although the `pcol` structure that is used to keep vertex pair collapse records is simple and short, the implementation requires more space since the original mesh is stored along with `pcol` records.

The future work categories may include:

(1) *Utilizing frame-to-frame coherence and amortization*: Frame rates could be increased by utilizing frame-to-frame coherence and amortization (traversing only a fraction of vertices for a frame), and by using generalized triangle strips for rendering.
(2) *Refinement of moving objects*: Since eye is naturally less sensitive to the moving parts of a mesh, refinement function can also take the motion into account and decrease the level of detail in proportion with the speed.
(3) *PM representations of regular meshes*: So far, all the work on PM confined into triangle irregular

networks (TIN). However, a suitable PM representation for regular meshes can also be developed. Such a representation will decrease the memory requirements for landscape visualization applications drastically.

(4) *Mapping on PM*: Efficient implementation of texture or bump mapping on PM can be developed. Because of the continuous structure of PM representation, mapping on PMs become tricky to implement.

(5) *Solution to popping problem*: Real-time view-dependent refinement of a progressive mesh can introduce some visual artifacts when the model is too close. Since we either do or not do a collapse, some parts of the mesh may appear to pop. This problem can be solved by introducing a weight between 0 and 1 with each collapse. A weight 0 means that a collapse should not be done and 1 means that the collapse should be done completely. The intermediate values can be used to interpolate the distance of the collapsing vertices. In order to incorporate this, refine function should be changed so that it returns this real value for each collapse instead of a boolean value.

Acknowledgements

We are grateful to Michael Garland for making his surface simplification program `Qslim` publicly available. Bunny, Dragon and Happy Buddha Models are obtained from Stanford University, Computer Graphics Laboratory. Other models are obtained from Viewpoint Data Laboratories. We thank Türker Yılmaz for his help in modifying the implementation to measure the performance. Special thanks to Varol Akman and Uğur Doğrusöz for valuable comments. We also would like to thank the anonymous referees for their valuable comments to improve the paper. Especially, one of the referees provided some technical insights that were included in the manuscript. The work in this paper is supported by an equipment grant from Turkish Scientific and Technical Research Council (TÜBİTAK) with grant number EEEAG 198E018.

References

[1] Funkhouser T, Sequin C. Adaptive display algorithm for interactive frame rates of complex virtual environments. ACM Computer Graphics, Proceedings of SIGGRAPH'93, August 1993. p. 247–54.

[2] Schroeder WJ, Zarge JA, Lorensen WE. Decimation of triangle meshes. ACM Computer Graphics, Proceedings of SIGGRAPH'92, Vol. 26, No. 2, July 1992. p. 65–70.

[3] Soucy M, Laurendeau D. Multiresolution surface modeling based on hierarchical triangulation. Computer Vision and Image Understanding 1996;63(1):1–14.

[4] Rossignac JR, Borrel P. Multiresolution 3D approximations for rendering complex scenes. In: Falcidieno B, Kunii T, editors. Modeling in computer graphics: methods and applications, 1993. p. 455–65.

[5] Garland M, Heckbert PS. Surface simplification using quadric error metrics. ACM Computer Graphics, Proceedings of SIGGRAPH'96, August 1996. p. 209–16.

[6] Gueziec A. Surface simplification with variable tolerance. In: Second Annual International Symposium on Medical Robotics and Computer Assisted Surgery (MRCAS'95), November 1995. p. 132–9.

[7] Hoppe H, DeRose T, Duchamp T, McDonald J, Stuetzle W. Mesh optimization. ACM Computer Graphics, Proceedings of SIGGRAPH'93, August 1993. p. 19–26.

[8] Turk G. Re-tiling polygonal surfaces. ACM Computer Graphics, Proceedings of SIGGRAPH'92, July 1992. p. 55–64.

[9] Cignoni P, Montani C, Scopigno R. A comparison of mesh simplification algorithms. Computers & Graphics 1998;22(1):37–54.

[10] Hoppe H. Progressive meshes. ACM Computer Graphics, Proceedings of SIGGRAPH'96, August 1996. p. 99–108.

[11] Xia J, Varsney A. Dynamic view-dependent simplification of polygonal models. Proceedings of Visualization'96, 1996. p. 327–34.

[12] Luebke D, Erikson C. View-dependent simplification of arbitrary polygonal environments. ACM Computer Graphics, Proceedings of SIGGRAPH'97, August 1997. p. 199–208.

[13] Hoppe H. View-dependent refinement of progressive meshes. ACM Computer Graphics, Proceedings of SIGGRAPH'97, August 1997. p. 189–98.

[14] Lindstrom P, Koller D, Ribarsky W, Hodges L, Faust N, Turner G. Real-time continuous level of detail rendering of height fields. ACM Computer Graphics, Proceedings of SIGGRAPH'96, August 1996. p. 109–18.

[15] Popović J, Hoppe H. Progressive simplicial complexes. ACM Computer Graphics, Proceedings of SIGGRAPH'97, August 1997. p. 199–208.

[16] Hoppe H. Efficient implementation of progressive meshes. Computers & Graphics 1998;22(1):27–36.

[17] Güdükbay U. View dependent refinement of polygonal meshes. http://www.cs.bilkent.edu.tr/~gudukbay/view_dependent.html