A Texture-Based Appearance Preserving Level of Detail Algorithm for Real-time Rendering of High Quality Images

Karsten Hilbert and Guido Brunnett

Computer Graphics and Visualization, Chemnitz University of Technology

Abstract

In this paper we present an appearance preserving Level of Detail (LOD) algorithm that treats attributes as maps on surfaces. Using this approach it is possible to render high quality images of extremely complex models at high frame rates. It is based on the Hierarchical Dynamic Simplification (HDS) framework for LOD proposed by Luebke and Errikson[LE97]. Our algorithm consists of a pre-process and a runtime phase. In the pre-process each object of a scene is segmented into so-called components by normal clustering. A component represents a set of nearly coplanar and partially non-adjacent triangles that do not occlude each other for a proper view. For each component textures containing color and normal information are generated hardware accelerated. Further on a mapping between object space and texture space is determined for each component. This allows a dynamic hardware accelerated computation of texture coordinates at runtime. Finally a dynamic multiresolution representation of each object's geometry is generated in the pre-process. Instead of attributes only texture identifiers are saved for each triangle in each object's multiresolution representation. Creation of textures and multiresolution representations is fast. During runtime for each object an adaptive approximation with an extremely low complexity is extracted out of the object's multiresolution model. Component textures containing color and normal information are applied to these approximations using dynamically generated texture coordinates. Although the complexity of the used approximations is very low high quality images can still be rendered very fast using our texture-based approach. Further on the method described in this paper adapts very well to distributed applications and other real-time rendering methods like back face culling.

Categories and Subject Descriptors (ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modelling - surfaces and object representations.

1. Introduction

1.1 Focus of the Paper

Level-of-Detail (LOD) based rendering approaches process a multiresolution model (MRM) either for the whole scene or for each object in the scene. In a pre-process an MRM (that is a data structure out of that approximations can be extracted) is produced for each object or the whole scene. At runtime approximations are extracted out of a MRM for rendering. While static LOD algorithms select a discrete approximation out of a pre-created set of approximations dynamic LOD algorithms extract approximations out of an abstract data structure at runtime. MRMs can be processed on a single machine or in a distributed manner.

If high quality images should be rendered using an LOD based rendering approach then high visual similarity between the original object and an approximation extracted out of the MRM has to be given at least from the current point of view. High visual fidelity is reached by using an LOD based rendering approach that uses appearance preserving criteria when generating an MRM. In this case fidelity is expressed in terms of maximum screen space deviation, meaning that the approximation's appearance when rendered should deviate from the original's appearance by no more than a user-specified number of pixels. However, in former years appearance preserving LOD algorithms were not used very often, because computation of an appearance preserving criteria can be very expensive.

Instead similarity of shape was used as proxy for similarity of appearance, because its computation is less expensive. In this case shape is preserved by minimizing distances between the original object's surface and the approximation's surface. Surface color and normal orientation are not taken into account. The term subset placement is used to characterize an approach where the vertices of any approximation is a subset of the original vertex set. Other approaches are characterized as optimal placement.

Current rendering hardware provides features like color mapping, bump mapping, large texture memory, multi-texturing, texture compression, fast image read back, occlusion queries etc. that make efficient computation of appearance preserving criteria to control the MRM generation process possible.

1.2 Previous Work

In the case of appearance preserving LOD algorithms fidelity is expressed in terms of screen space deviation, meaning that the approximation's appearance when rendered should deviate from the original's appearance by no more than a user-specified number of pixels. The appearance of an approximation extracted out of an MRM is not only affected by surface position but also by surface color and curvature. There exist three major classes of appearance preserving algorithms.

The first class is formed by algorithms that do an image-driven simplification. An example is the method proposed by Lindstrom and Turk [LT00]. During the process of MRM generation the next applied simplification operation is chosen based on a purely image-based metric. Following to each simplification step the resulting approximation and the original object are rendered from different points of view. To compute the approximations error differences in luminance between corresponding images of the original and the approximation are computed and accumulated over all views. This method balances geometric and shading properties. It couples a high fidelity preservation of silhouette regions with drastic simplification of unseen geometry. Furthermore it has a simplification sensitivity to artefacts caused by shading interpolation and to the content of texture maps across the surface. Several optimizations have been proposed. The main disadvantage of this kind of algorithms is the low speed. Reducing an original object consisting of thousands of triangles to an approximation consisting of a few hundred polygons can last hours.

The second class of algorithms use geometric criteria modified to account for color, normal and texture distortion to control the process of simplification when producing an MRM. Certain et al. [CPD*96] have proposed a method for adding surface attributes to a wavelet decomposition. In Hoppe's error metric that has been used for generating progressive meshes [Hop96] per-vertex-attributes and per face attributes are explicitly included. The quadric error metric (QEM) proposed by Garland and Heckbert [GH97] is an efficient way to represent the error introduced by a sequence of vertex merge operations. QEMs provide a fast and simple way to control the MRM generation process with minor storage costs. First Erikson and Manocha [EM99] addressed the lack of support of shading attributes like color, normal and texture coordinates in the original version of Heckbert and Garland. Heckbert and Garland [GH98] themselves published an extension of QEM for color. The best extension of QEM to handle attributes was proposed by Hoppe [Hop99]. Nowadays the quadric error metric algorithms are the best compromise between efficiency, fidelity and generality for generating MRMs. While the former approaches of this class have been time consuming the younger QEM based algorithms are quite fast and visual fidelity is quite high even at drastic levels of simplification. All these solutions have in common that the user can specified a fixed value for the relative importance given to the preservation of shading attributes versus preservation of geometric fidelity. This relative importance is not adapted during simplification process. The main disadvantage of these methods is the direct coupling of resolution of geometry and resolution of attributes. If there is a surface with a complicated color and normal distribution on a small area the possible level of reduction is limited.

The last class of algorithms treats attributes as maps on the surface. Attributes of the original object are sampled in textures that can be mapped on approximations. One subgroup of this class reparametrizes normal and color maps rather than preserves shading attribute values (e.g. Cohen et al. [COM98]). The other subgroup decouples geometry and attributes more drastically (e.g. Maruya[Mar95], Soucy et al.[SGR96], Cigioni et al.[CMR*98]). When creating a simplified version of an object they first simplify the geometry of the object regardless of attributes, sample the attributes of the original object into textures and apply this texture to the approximation. As mentioned above the main advantage of this class of algorithms is the fact that resolution of geometry and resolution of attributes is decoupled. That means that the geometry of an approximation can be reduced drastically and visual features removed by this reduction can be restored by applying the attribute textures. Approximations generated by these methods can efficiently be used by LOD based rendering systems if the recommended texturing methods are supported by the hardware. It is disadvantageous that these methods generate textures that are highly fragmented. The attributes of neighbouring triangles are not always mapped to neighbouring texels. In consequence, it is difficult to create mip maps for these textures.

1.3 Our Approach

In [HB04] we have described our distributed hybrid LOD based rendering approach. We have integrated this algorithm in our rendering system that we are using for VR visualizations.

In the context of VR it is necessary to render high quality images at constantly high frame rates to provide an appropriate level of immersion. Because our distributed LOD application runs on the nodes of a homogenous cluster connected by Gigabit Ethernet the amount of data that can be transmitted in a fixed time period is limited by network bandwidth. This restricts the speed of our application. There are two possibilities to reach an acceleration. The first possibility is the integration of a caching mechanism. A more complicated, but also more effective way is reducing the amount of data that have to be transmitted. The complexity of the models can be reduced to a low level, but must be compensated. Further on transmission of attribute data can be optimized.

To solve the described problem we extended Luebke's and Erikson's dynamic LOD algorithm HDS[LE97] that uses geometric criteria into a dynamic appearance preserving LOD algorithm that treats attributes as textures on the surfaces. Separation between geometry and attributes of an object allows separated processing. The geometry can be reduced to a minimum and loss of details can be compensated by using texturing approaches like color mapping and dot3 bump mapping.

Just like every LOD algorithm the resulting algorithm consists of two stages- a pre-process and a runtime phase. In contrast to [LE97] it processes an MRM and textures for each object of the scene.

In the pre-process stage the original triangle set is divided into components that contain not always adjacent triangles with a similar orientation that do not occlude each other from a given point of view. For each component textures containing color and normal information are generated hardware accelerated by projecting each component's triangles on its associated plane and rasterizing this projection with an appropriate resolution. These textures are combined to larger atlas maps using a packing strategy. Further on, a piecewise continuous mapping between object space and texture space has to be determined such that for each position on an approximation's surface the corresponding color and normal information can be indexed. Finally, a vertex tree is generated by using a vertex clustering technique. Additional information about mapping between object and texture space are stored in the MRM.

In the runtime phase a cut through the vertex tree is incrementally updated controlled by a view-dependent criterion. The geometry of the approximation that corresponds to this cut is generated.

When the approximation should be rendered the piecewise continuous mapping is used to generate texture coordinates for the approximation's vertices that index the corresponding color and normal information. Finally the approximation is rendered using multi-texturing with a color and a normal map applied to itself.

1.4 Structure of the Paper

In chapter two we will describe the pre-process stage of our approach in detail. The runtime phase will be described in chapter three. After that, we will provide the results reached with an OpenGL implementation of our approach. Finally we will give a short conclusion in the last chapter.

2. Pre-process Phase

In our method we create 2D textures by projecting segments of the object onto appropriate planes. These textures contain color and normal information of the original object. We apply them to approximations extracted out of a hierarchical MRM to render high quality images. In the following section we describe how projection planes are chosen, how the textures for color and normal information are obtained and how the MRM is generated.

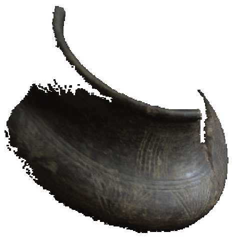


Figure 1. *Original Surface*

2.1 Segments and Components

For the segmentation we want to keep the number of segments small but also to restrict the effect of projection distortion. This is achieved based on a clustering procedure for the mesh normals. The resulting "principal normals" are used to determine the orientation of the projection planes. For each of these planes the associated set of triangles (segment) is further subdivided into components that project onto the planes without occlusion.

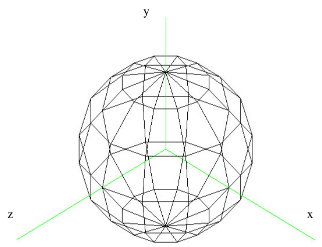


Figure 2. *Segmentation of Unit Sphere*

Initially we compute the Gauß map of the mesh by representing normals of triangles as points on the unit sphere. The sphere is subdivided into quadrilateral spherical sections. The size of sphere sections is defined by a user-specified angular step that defines the extension of a sphere section in longitudinal and latitudinal direction. Decreasing this step results in a segmentation that consist of a larger number of smaller segments. From the tessellation of the sphere the principal normals are extracted by an iterative procedure that consists of the following steps. First we choose that spherical section that contains the largest number of points. The average of the corresponding normals is used as a reference normal. Now, we determine those normals from all sections that differ from the reference normal by an angular value that is smaller than an user-defined threshold. The triangles associated with these normals are combined into a segment of the object's surface. Any normal that belongs to a triangle in the determined segment is removed from the Gauß map. This procedure is performed until each triangle belongs to a segment of the object. The average of the normals of a segment determines a principal normal. Each principal normal is used to define the orientation of a projection plane. Each of these planes is placed such that it contains the point given by the position vector λn where n is its normal and λ is the largest diameter of the original object.

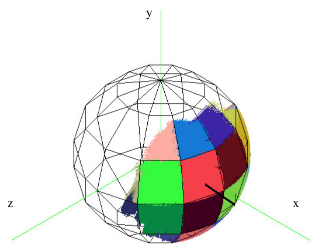


Figure 3. *Estimation of a Reference Normal*

Since we deal with real data deliverd by a 3D scanner we have to cope with noise. For noisy data the segmentation can be very fragmented if the method is directly based on normals of triangles. In this case smoothed normals should be used.

After the projection planes are determined the segments are further subdivided into components. A component of a segment is a subset of its triangles that do not overlap after orthogonal projection onto the segment's projection plane. Components are determined using an iterative hardware accelerated process.

Initially all triangles contained in a segment form the set of candidate triangles that have to be considered. The depth buffer is initialized. The set of candidates is sorted once in ascending order depending on each candidate triangle's distance to the segment's plane. To determine all members of a component the procedure processes all candidate triangles in this order. Therefore each candidate triangle is rendered twice from the point of view of the plane associated with the segment. Both times lighting and writing to color buffer is disabled to enfasten rendering. While the triangle is rendered without depth buffer test in the first pass, it is rendered with depth buffer test enabled in the second pass. In both passes an occlusion query is done. A candidate triangle where both occlusion queries deliver the same number of fragments is added to the current component and removed from the set of candidate triangles. If an candidate triangle is added to a component the content of the depth buffer is updated. After one pass through all candidate triangles depth buffer is cleared and the next component of the segment is determined by examining candidate triangles in the given order. So components are added until no candidate triangle is left. For each component an average color and an average normal is computed. This is done for all segments.

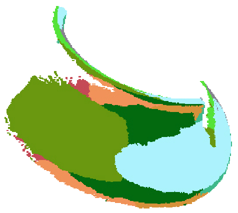


Figure 4. *Segmentation of Original Surface into Subgroups*

In the following step for each component of a segment the surrounding neighborhood is determined. Triangles are adjacent if they share an edge. The degree of relationship of two triangles is denoted as the minimal number of adjacent triangles that connect both triangles. The neighbourhood of a component consists of all triangles that do not belong to the component and whose degree of relationship does not exceed a user-specified threshold.

2.2 Rendering and Packing Color and Normal Textures

If color and normal information should be reconstructable in a proper way choosing the right resolution for sampling these information into texture is important. If the chosen resolution is too small color and normal information can get lost. If the chosen resolution is too large memory space is wasted. Extent of textures into that color and normal information of triangles forming a component is sampled is determined as described in the following paragraph. First the component's axis aligned bounding box is computed. After that the bounding box is projected onto the component's projection plane. The height and width of this projection's bounding rectangle are estimated and rounded up to powers of two. These rounded values define the extent of the textures where color and normal information of this component are sampled into.

The largest extent d_{max} of a component is also an important value. Therefore the length of the largest diagonal of the component's bounding box is used.

The midpoint M_{sg} of the component has to be computed. To align the component properly for rendering a transformation matrix $R = R_{-z} \cdot R_y$ has to be determined. This matrix R can be derived from its projection plane's orientation. A detailed description of this matrix' computation follows in the next section.

These parameters can be used to define a rendering setup. An orthogonal projection is used to avoid distortions of textures. Extension of corresponding viewing volume can be derived from maximum extent of the component's bounding box. The viewport's width and height is defined by the resolution of the textures that should be rendered. Camera is positioned at a position on x-axis where x coordinate is equal to maximum extent of the component's bounding volume. The camera looks along the negative x-axis. Camera's up vector is defined by positive y-axis. Based on the information where the component's center point is the component can be centered at origin. The triangles forming a component are aligned using the computed transformation matrix so that they are nearly coplanar to zy-plane.

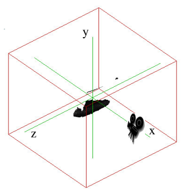


Figure 5. *Rendering Setup. Viewing Volume,* ***Camera looking along x-axis****, Component centred at Origin*

Now for each component textures containing color and normal information can be rendered using nearly the same procedure. In both cases the determined rendering setup is used. Regardless of what map is rendered a special background color value is set. First all neighboring triangles of the component are rendered with color. After that, the depth buffer is cleared and finally all triangles belonging to the component are rendered with color. In both rendering passes lighting is disabled and depth buffer test is enabled. Not considering the kind of map that is actually rendered always background's alpha value is set to zero and alpha value of neighboring and member triangles' vertices is set to one. This creates an alpha mask that can be used during rendering of approximations to reconstruct holes even if the simplification process has closed these holes in geometry.

While the average color of all triangles contained in the component is used as background color in the case of rendering the color map, a color value representing the coded average normal of a component's vertices is used as background color. Further on the original color values of a component's vertices are used during rendering of color map. During rendering of normal map color values are used that represent range compressed normals of a component's vertices. A range compression maps coordinate values from interval [-1,1] to interval [0,1] according to expression (1).

$$\vec{n}_{comp} = 0.5 \cdot \vec{n} + \begin{pmatrix} 0.5 & 0.5 & 0.5 \end{pmatrix}^T \tag{1}$$

Finally in both cases texture names are created, maps are read back from frame buffer and transferred to texture memory. The projection and the modelview matrix used for rendering the component's color and normal texture have to be saved because they are necessary for computing texture coordinates of the component's vertices.

If a component's texture would only contain the component's triangles discontinuities in the rendered image of approximations that were textured with these maps. That is the reason why we set a special background color and additionally sample neighbouring triangles in the textures. These discontinuities are caused by texture filtering and by limited precision of texture coordinates.

These sets of normal and color maps could directly be used to render approximations, but depending on the original object's geometry a lot of small textures may be created in this step. The created sets of textures can be combined to larger atlas map. Using atlas maps minimizes the number of texture object switches that have to be done, but texture coordinates of vertices have to be biased and scaled.

Color and normal maps of components are combined to separate atlas maps using the same method. Maximum size of the atlas maps is limited by the maximum size of textures the used rendering hardware supports. First all color textures are sorted according to their area in decreasing order. Then a two stage algorithm consisting of an insertion and a filling step iterates until all textures are combined to atlas maps.

In the insertion step the algorithm selects the largest texture that has not been inserted into the atlas map yet. If the difference between atlas map's and texture's width is smaller than the difference between atlas map's and texture's height the texture is appended at bottom left or right top position. If the insertion of a texture would create an atlas map whose width or height exceeds the user-defined maximum extent of an atlas map a new atlas map is created and the texture is inserted in the top left corner.

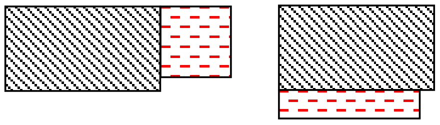


Figure 6. *Possibilities of texture insertion in atlas map depending on best matching edges*

If width or height of the atlas map and the inserted texture do not match perfectly a free area in the bottom right corner is created by the insertion step.

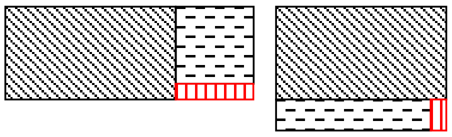


Figure 7. *Possible free areas that can be created by the insertion step.*

This free area is filled in the filling step. The filling step is a recursive process. First the largest texture that has not been inserted until now and that fits best in this free area is inserted in the free area. Caused by this insertion up to two new free rectangular areas can be created. For these free areas filling proceeds recursively until there is no texture left that fits in the remaining free area.

Now a new insertion step is done. This process continues until no texture is left that can be inserted in an atlas map. Normal textures of components are combined in the same way.

Combining the color and normal textures of components to larger atlas maps affects the mapping used to transform a vertex' object coordinates into texture coordinates that index the corresponding color value and normal vector saved in the corresponding color and normal map. Depending on texture's position in the atlas map and ratio between texture's dimensions and atlas map's dimensions the corresponding translation and scaling matrix can be determined.

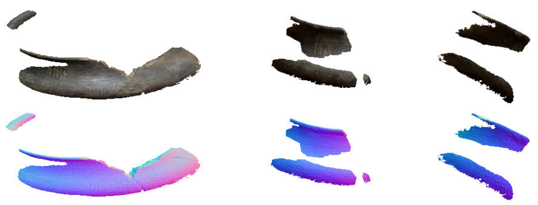


Figure 8. *Example of Color and Normal Atlas Map*

Further on, it should be noted that mip maps of these atlas maps should not be created because this leads to mixing of color values or coded normal vectors that do not belong to neighboring surface areas. We advise using bilinear filtering when rendering approximations using these color and normal atlas maps.

Finally it should be noted that color atlas maps are compressed on the fly if the used rendering hardware supports texture compression. Compression of normal atlas maps is not done because this leads to poor results of dot3 bump mapping.

2.3 Finding a Mapping between Object Space and Texture Space

Now, we have to make sure that these textures can efficiently be applied to geometry during runtime. We do not assign fixed texture coordinates to the vertices of each component's triangle. This can introduce distortion in the rendered image because LOD algorithms create approximations with vertices whose position is optimized to minimize geometric error of approximation. Instead we compute a transformation matrix for each component. As described in (2) the matrix called *TCM'* can be used to transform object coordinates of a component's vertices into texture coordinates that index the corresponding color value and normal in the component's color and normal map.

$$\begin{bmatrix} s \\ t \\ 0 \\ 1 \end{bmatrix} = TCM' \cdot \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix} \qquad (2)$$

A component's color or normal texture is created by rendering the component's triangles transformed according to the orientation of the component's projection plane, using proper camera settings and an orthogonal projection and rendering in a proper viewport. To compute the texture coordinates that index the corresponding color value and normal vector of a component's vertex the transformations mentioned above have to be applied to the vertices' object coordinates in the same order. The matrix *TCM* that realizes this transformation is computed as described in equation (3).

$$TCM = A \cdot PM \cdot MV \tag{3}$$

The matrix *A* realizes a mapping of coordinate values from range [-1,1] to range [0,1] and sets the z coordinate to zero.

$$A = \begin{bmatrix} 0,5 & 0 & 0 & 0,5 \\ 0 & 0,5 & 0 & 0,5 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

PM is a matrix that realizes an orthogonal projection. Described more in detail it realizes a mapping from the cuboid used as viewing volume during rendering of component's textures to a cube defined by $-1 \leq x,y,z \leq 1$. The viewing volume is defined by $-d_{max} \leq x,y \leq d_{max}$ and $0 \leq z \leq 2d_{max}$ where d_{max} denotes the maximum dimension of the component's bounding box multiplied by $\sqrt{3}$.

$$PM = \begin{bmatrix} \frac{1}{d_{\max}} & 0 & 0 & 0 \\ 0 & \frac{1}{d_{\max}} & 0 & 0 \\ 0 & 0 & \frac{1}{d_{\max}} & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

MV is a combination of two translation and two rotation matrices.

$$MV = T_z \cdot R_{-z} \cdot R_y \cdot T_{M_{sg}} \tag{6}$$

While T_z represents the viewing transformation used during rendering of component's textures, the combination of R_{-z}, R_y and T_{Msg} represents the model transformation used during this rendering process. T_z is just a translation along z-axis by $-d_{max}$. While R_{-z} is a clockwise rotation of *wz* degrees around the z-axis, R_y is a counter clockwise rotation of *wy* degrees around the y-axis. The angles *wz* and *wy* define the orientation of the projection plane the component was assigned to. These angles can be computed by determining the angle between projection plane's normal and its projection onto the xz-plane and the angle between this normal's projection onto the xz-plane and $\boldsymbol{e}_x$. T_{Msg} is a translation applied to center the component at the origin.

Finally color and normal textures are combined to separate atlas maps. Because of this we have to apply a translation and a scaling additionally to the texture coordinates of a component's vertex.

$$TCM' = T_O \cdot S \cdot TCM \tag{7}$$

T_O describes a translation along the vector $\vec{t}$ that points from (0,0) to a position where the left upper corner of the component's color or normal texture *n* was inserted in a color or normal atlas map.

$$\vec{t} = \begin{bmatrix} \frac{O_{n,s}}{w_a} & \frac{O_{n,t}}{h_a} & 0 & 1 \end{bmatrix}^T \tag{8}$$

Finally *S* describes the scaling of texture coordinate *s* of a component's vertex by the ratio of texture's width w_n to atlas map width w_a and the scaling of texture coordinate *t* by the ratio of texture's height h_n to atlas map's height h_a.

$$S = \begin{bmatrix} \frac{w_n}{w_a} & 0 & 0 & 0 \\ 0 & \frac{h_n}{h_a} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9}$$

Finally an array is generated that contains one entry for each component. An entry of this array contains the corresponding transformation matrix and identifiers of the color and normal map where the color and normal information of the component's vertices are saved.

2.4 Generating a Vertex Tree

Our MRM of an object consists of a vertex tree, a list of active triangles, textures that contain color and normal information and an array that contains information that are used to realize a mapping between color and normal information and the vertices of components.

A detailed description of our method for generating a vertex tree can be found in [HB04].

A vertex tree is a continuous hierarchical multiresolution representation of a scene's geometry [LE97]. In contrast to [LE97] we create a vertex tree for each object in the scene. In fact a vertex hierarchy can be produced by using several simplification operations. We use an octree-clustering-scheme suggested in [LE97]. This scheme does not require any knowledge about the polygonal mesh. Manifold topology is neither assumed nor preserved. It is very fast. A disadvantage of this vertex tree generation strategy is that meshes with degeneracies (as cracks, T-junctions, and missing polygons) may appear. Although the similarity of shape of approximations extracted out of a vertex hierarchy produced by vertex clustering is not the best we used this method for generating a vertex hierarchy to show how quality of approximations can be approved by using our appearance preserving method.

This clustering scheme does a recursive subdivision of the object's bounding box. For each octree cell in this hierarchy a representative vertex is chosen using a local criterion that is computed based on local curvature and length of adjacent edges. So each node of a vertex tree corresponds to an octree cell. For each node of the vertex hierarchy the position of the corresponding octree cell in the octree structure, its status, the representative vertex of the corresponding octree cell, the bounding sphere of the vertices contained in the corresponding octree cell, pre-computed parameters for fast evaluation of a view-dependent criterion, information about fully and partially included triangles and some additional information are saved. In contrast to other approaches we do not save color and normal information for the representative vertex of an octree cell. Instead identifiers that index the corresponding color and the corresponding normal map are assigned to each triangle. Coordinates of vertices are saved as coordinates of object coordinate system. All computations are done in object coordinate system. This enables us to support dynamic scenes and instantiation of objects.

The list of active triangles contains the active triangles that form the current adaptive geometry-based approximation of the LOD object. During runtime its content is updated by moving a cut through the vertex hierarchy depending on a view-dependent criterion. Moving the cut through the vertex tree means collapsing and expansion of nodes. When a node is expanded a set of triangles is added to the list of active triangles. In the other case some triangles are removed from this list.

3. Runtime Phase

At runtime for each frame and each object a suitable approximation of the object's geometry has to be extracted out of the object's MRM and the corresponding color and normal atlas maps have to be applied to the approximating geometry to render an image. The realization of these two tasks is described in the following sections.

3.1 Extracting Approximation's Geometry out of the Vertex Tree

The extraction of a suitable adaptive approximation of an object's geometry requires the following steps:

Because we perform all computations in the object coordinate system we need the current model transformation of the object to be able to transform all necessary parameters in this coordinate system. That causes additional computational effort, but it enables us to process dynamic scenes and to support instantiation of objects. Knowledge about the current viewing configuration is necessary to be able to extract the correct adaptive geometry-based approximation of an object out of its vertex tree.

The list of active triangles corresponds to a cut that is moved through the vertex tree depending on a view-dependent criterion at runtime. This cut movement is performed incrementally because this is less expensive than determining the current cut by traversing the vertex tree starting at the root node. The cut movement through the vertex tree is controlled by a view-dependent criterion [LE97] that is based on screen-space error.

3.2 Applying Attribute Maps to the Geometry

To render an image of an object the approximation's set of triangles contained in the object's list of active triangles has to be rendered using the color and normal atlas maps. OpenGL is used for rendering.

Before a triangle can be rendered the index that was assigned to the triangle and that identifies the corresponding color and normal map has to be determined. Using this index the matrix that transforms the object coordinates of a triangle's vertices into texture coordinates according to expression (2) and the identifiers of the corresponding color and normal texture can be determined.

A triangle is rendered using multitexturing capabilities of rendering hardware. Texturing Unit 0 is configured to do Dot3-Bump Mapping. Depending on the fact which component the triangle was assigned to the proper texture that contains the normal information of the component the triangle was assigned to is bound to this first texturing unit. To be able to do bump mapping the light direction vector pointing from the vertex to the light source has to be coded as the color value of each vertex of the triangle according to expression (1). The color value that this texturing unit produces for a fragment replaces the old fragment color value.

Texturing Unit 1 is configured to do color and transparency mapping. Depending on the fact which component the triangle was assigned to the proper texture that contains the color and transparency information of the component the triangle was assigned to is bound to this second texturing unit. The color value of a fragment produced by this texturing unit is modulated with the color value produced by the first texturing unit. The result of this modulation is the resulting fragment color.

Texture coordinates have to be computed for each vertex of a triangle that index into the corresponding maps that contain color and normal information of the component the triangle belongs to. The transformation matrix *TCM'* computed for each component is used to transform the object coordinates of a component's vertex into texture coordinates that index the corresponding color and normal in the proper textures. Expression (2) describes this transformation. Texturing hardware has to be configured properly so that the object coordinates of a triangle's vertex are transformed using the matrix of the component the triangle was assigned to automatically. In the context of OpenGL this means setting the GL_TEXTURE_GEN_MODE to GL_OBJECT_LINEAR for *s* and *t* texture coordinate. Always the first two rows of the proper matrix *TCM'* are used as parameters for texture coordinate generation of texture coordinate *s* and *t*.

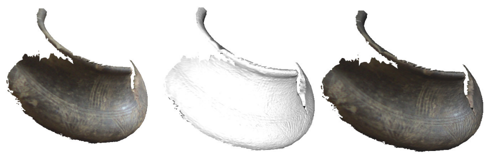


Figure 9. *Approximation textured with Color Maps (left), Normal Maps (middle) and Combination of both (right)*

Furthermore alpha testing has to be enabled. It is used to reconstruct holes in the surface. Because we are using a vertex clustering scheme to construct a vertex hierarchy for each object an approximation can be extracted out of this vertex hierarchy where holes are closed. The correct shape of holes is coded in the alpha channel of the color atlas maps. While an alpha value 1 is assigned to regions that represent the object surface, an alpha value 0 is assigned to regions that do not belong to the object's surface. Fragments that represent regions of approximation's surface where originally a hole was situated are rejected from writing into the color buffer. So holes of surface that were closed during approximation process were cut out using alpha testing.

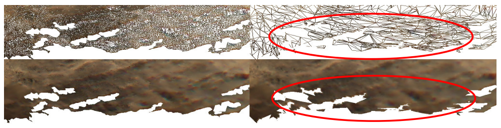


Figure 10. *Reconstruction of holes using alpha testing*

4. Results

We have implemented our approach on a PC with an Intel Pentium4 2,8 GHz, 1 GB RAM, a Nvidia GeForce FX 5900 Ultra and a Windows XP system installed. Functionality of the library VDSlib [Lue02] implemented by David Luebke is used by our implementation for generating hierarchical multiresolution models of objects and extracting adaptive approximations out of them. OpenGL is used for rendering. We tested it with several objects. We used a set of models that were created using a 3D laser scanner. The used multiresolution models were created very fast. Times for creation of MRMs and sizes of MRMs are listed in **table 1**.

Table 2 shows that we are able to render high quality images using approximations with low polygonal complexity at high frame rates. We can speed up rendering using our approach by factor 6 and above. In all cases a fluent interaction with the shown object was possible. Although we reduced the complexity of the objects by factor 100 image quality is still high. As the reader can see in **table 2** in all cases a high visual similarity between the image showing the original object and the image showing an approximation is reached. In some cases we have identified small rendering artefacts represented by some wrong colored and wrong lighted triangles. There are two different kinds of artefacts as can be seen in **figure 11**.

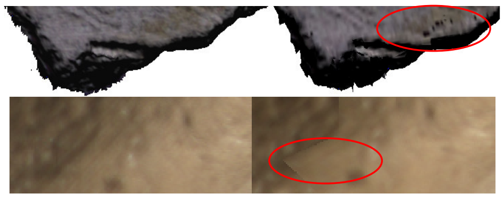


Figure 11. *Comparison of Original Object and Approximation with Depth Buffer Artefacts (top) and with Artefacts caused by Simplification Scheme (bottom)*

The quality of approximations that could be extracted out of an MRM depends on the scheme used for MRM generation. As mentioned before, the vertex clustering scheme produces an MRM out of that approximation can be extracted that include several degeneracies. These degeneracies sometimes cause the first kind of rendering artefacts illustrated in **figure 11** at the bottom. By using another scheme these artefacts can be avoided.

The quality of color and normal textures produced by our approach depends on the accuracy of values saved in the depth buffer of the used rendering hardware. The decision which component a triangle is assigned to is done by doing two occlusion queries- one with depth buffer test disabled, one with depth buffer test enabled. If depth buffer accuracy is low wrong results could be produced if a triangle is processed whose distance to other triangles is very small in comparison two the extent of the object. These wrong decisions cause the second kind of rendering artefacts illustrated in **figure 11** at the top.

Object (# polygons)	**Size of MRM**		**Time for MRM generation**	
	Vertex tree (in MB)	**Textures** (in MB)	**Normal smoothing** (in s)	**Vertex tree and texture generation** (in s)
Ornament (260.767)	36,12	6,55	8,418	11,085
Plate (600.000)	83,02	12,45	26,303	26,182
Pot (2.822.425)	402,181	21,626	121,649	242,273
Tile (3.079.423)	451,172	46,039	128,819	546,095

Table 1. *Sizes of MRMs and Times for MRM Generation*

5. Conclusion

We have presented a method that extents a shape preserving LOD algorithm [LE97] to an appearance preserving LOD algorithm that can be used to render high quality images at high frame rates. To reach this aim we have decoupled the processing of geometry and attributes. Therefore our method samples normal and color information of an object in textures. We have show how several capabilities of rendering hardware can be used to do this efficiently. A vertex tree is generated for each object using a vertex clustering scheme suggested by Luebke and Erikson [LE97]. At runtime view-dependent approximations are extracted out of the object's MRM depending on a view-dependent criterion. These approximations are rendered using the color and normal textures generated in the pre-process. We have show how texture coordinates that index the corresponding color value and normal vector saved in the corresponding texture can be computed efficiently and automatically. We described how this can be done hardware accelerated. It was proven that our approach allows rendering high quality images at high frame rates by using object approximations with very low polygonal complexity. We have located the reasons why artefacts occur sometimes when our approach is used.
In the near future we will integrate this method in our distributed LOD application described in [HB04]. We have located two reasons for artefacts that can be produced by our method. On the one hand artefacts are caused by the simplification scheme we use. Out of an MRM generated by using a vertex clustering scheme high quality approximations can not be extracted in any situation. These artefacts could be avoided by using another kind of MRM, e.g. a progressive mesh. We plan to integrate our attribute processing mechanism in other existing LOD algorithms to test if image quality could be improved. On the other hand artefacts are caused by the fact that the segmentation of

surfaces into components depends on accuracy of depth buffer involved in the decision if a triangle is assigned to a component or not. Solutions for this problem have to be found in the near future.

6. References

[LT00] Lindstrom P., Turk G:. Image-Driven Simplification. *ACM Trans. Graphics. vol. 19,* 3 (2000), pp. 204-241.

[CPD*96] Certain A., Popovic J., DeRose T., Duchamp T., Salesin D. and Stuetzle W.: Interactive multiresolution surface viewing. *SIGGRAPH 96 Proc.* (1996), pp. 91–98.

[Hop96] Hoppe H.: Progressive meshes. *SIGGRAPH '96 Proc.* (1996), pp. 99–108.

[GH97] Garland M. and Heckbert P.: Simplification Using Quadric Error Metrics Computer Graphics. *SIGGRAPH '97 Proc., 31* (1997), pp. 209-216.

[EM99] Erikson C. and Manocha D.: GAPS: General and Automatic Polygonal Simplification. *Proc. 1999 Symp. Interactive 3D Graphics (*1999), pp. 79-88.

[GH98] Garland M. and Heckbert P.: Simplifying Surfaces with Color and Texture using Quadric Error Metrics, *Proc. IEEE Visualization 98* (1998), pp. 263-270.

[Hop99] Hoppe H.: New Quadric Metric for Simplifying Meshes with Appearance Attributes, *Proc. IEEE Visualization 99* (1999), pp. 59-66.

[COM98] Cohen J., Olano M., and Manocha D.: Appearance-preserving simplification. *Proceedings SIGGRAPH 98* (1998), pp. 115–122.

[Mar95] Maruya M.: Generating a texture map from object-surface texture data. *Computer Graphics Forum, 14*, 3 (1995), pp. 397–405,506–507, Proc. EG '95.

[SGR96] Soucy M., Godin G., Rioux M.: A texturemapping approach for the compression of colored 3D triangulations. *The Visual Computer, 12,* 10 (1996), pp. 503–514.

[CMR*98] Cignoni P., Montani C., Rocchini C., and Scopino R.: A general method for preserving attribute values on simplified meshes. *IEEE Visualization 98 Conference Proceedings* (1998), pp. 59–66,518.

[HB04] Hilbert K., Brunnett G.: A Hybrid LOD Based Rendering Approach for Dynamic Scenes, *Proc. CGI 2004* (2004), pp. 274-277, ISBN/ISSN 0-7695-2171-1.

[LE97] Luebke D. and Erikson C.: View-Dependent Simplification of Arbitrary Polygonal Environments, *Computer Graphics (Proc. Siggraph 97), 31* (1997), pp. 199-208.

[Lue02] Luebke D.: VDSlib Homepage, University of Virginia (2002), http://vdslib.virginia.edu

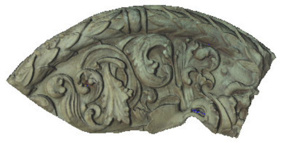
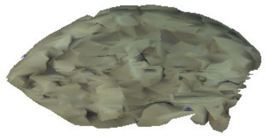
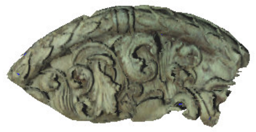
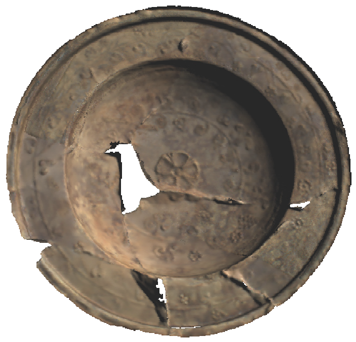
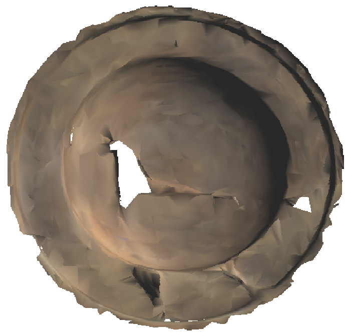
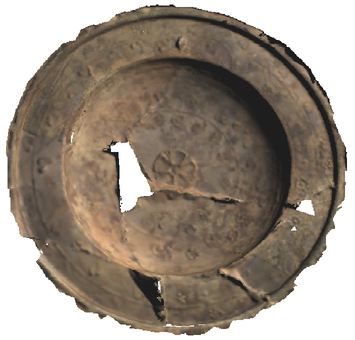
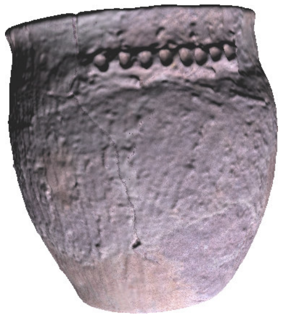
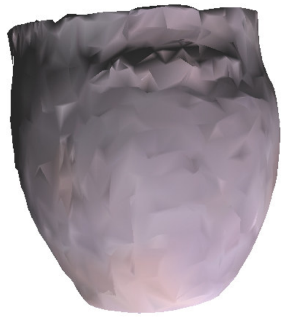
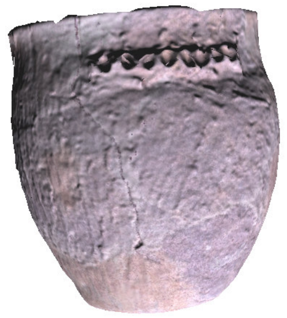
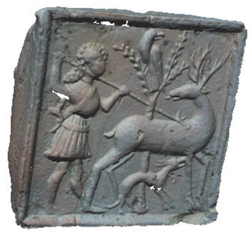
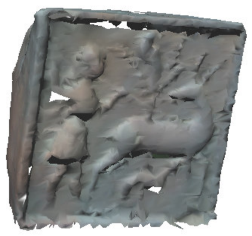
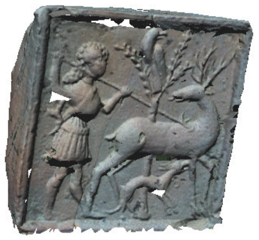
Original Object	**Approximation rendered with native HDS**	**Approximation rendered with Our extended HDS**
Ornament  260.767 polygons / 10,2 fps	 6.132 polygons / 115,5 fps	 6.132 polygons / 95,3 fps
Plate  600.000 polygons / 4,03 fps	 8.183 polygons / 115,4 fps	 8.183 polygons / 84,3 fps
Pot  2.822.425 polygons / 0,95 fps	 5.093 polygons / 121,15 fps	 5.093 polygons / 115,5 fps
Tile  3.079.423 polygons / 0,86 fps	 23.932 polygons / 53,1 fps	 23.932 polygons / 26,1 fps

Table 2. *Comparison of Complexity and Rendering Speed with and without using Our Extension of HDS*