Appearance Preserving Octree-Textures

Julien Lacoste[*] LIUPPA University of Pau
Tamy Boubekeur[†] TU Berlin
Bruno Jobard[‡] LIUPPA University of Pau
Christophe Schlick[§] INRIA University of Bordeaux

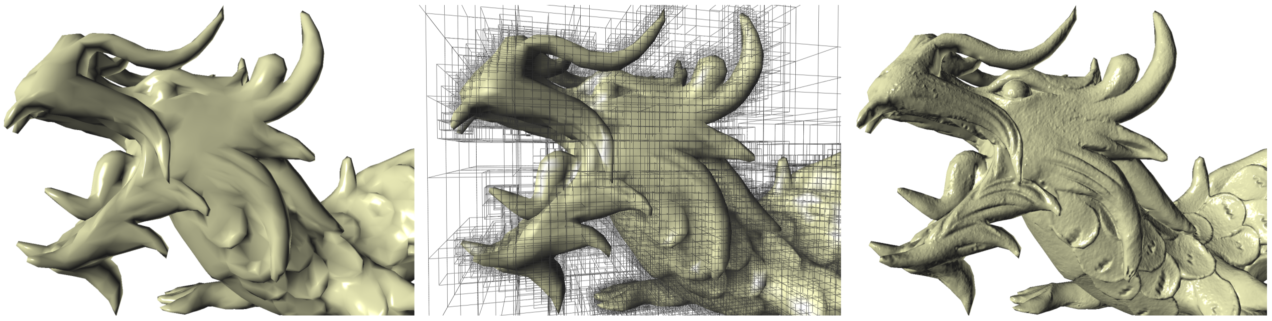


Figure 1: *Left: simplified mesh of 15K triangles. Middle: an octree is built around this mesh to adaptively sample and store the normal field of the high resolution version of the mesh. Right: the simplified mesh is normal mapped via a special GPU traversal of the octree cells encoded in a 2D texture. The operation does not requires 2D parametrization of the mesh.*

Abstract

Because of their geometric complexity, high resolution 3D models, either designed in high-end modeling packages or acquired with range scanning devices, cannot be directly used in applications that require rendering at interactive framerates. One clever method to overcome this limitation is to perform an *appearance preserving geometry simplification*, by replacing the original model with a low resolution mesh equipped with high resolution normal maps. This process visually preserves small scale features from the initial geometry, while only requiring a reduced set of polygons. However, this conversion usually relies on some kind of global or piecewise parameterization, combined with the generation of a texture atlas, a process that is computationally expensive and requires precise user supervision. In this paper, we propose an alternative method in which the normal field of a high resolution model is adaptively sampled and encoded in an octree-based data structure, that we call *appearance preserving octree-texture* (**APO**). Our main contributions are: a **normal-driven octree generation**, a **compact encoding** and an **efficient look-up algorithm**. Our method is efficient, totally automatic, and avoids the expensive creation of a parameterization with its corresponding texture atlas.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—;

Keywords: Appearance Preserving simplification, Octree-Textures, GPU, Normal mapping

1 Introduction

Among the various lessons that high quality off-line rendering has taught us, the most important one involves the fundamental role played by normal vectors: the final appearance of a surface is more influenced when locally editing the surface gradient rather than the surface geometry. This explains the growing interest in *normal mapping* techniques with recent real-time 3D engines. The normal mapping techniques permits to strongly reduce the complexity of a mesh, by converting the local geometry variation into a map of normal vectors, that will be used at the rendering stage by adapted fragment shaders. However, with existing techniques, the use of normal maps suffers from two strong drawbacks:

- As normal maps are stored as 2D textures, they require either a global or a piecewise parameterization of the 3D model, which may involve complex computation and generally can't be performed automatically for non trivial topologies.
- To compensate for the geometric distortion involved by the projection operator, over-sampling has to be used in order to avoid loss of details.

Both drawbacks are directly linked to the 2D nature of the normal map, so seeking for an alternative representation may be worth trying. While raw 3D textures would clearly be too expensive even for moderately complex models, *octree-textures* recently introduced for painting-on-surface applications, offer many interesting properties: they do not require any surface parameterization, and provide easy efficient adaptive sampling, as well as compact storage, as no values are stored in the empty space surrounding the object. In this paper, we propose a general framework to extract and encode the normal map of an arbitrarily complex object using octree-textures. Our algorithm takes two versions of the same object as an input: a full resolution version M providing all the features that we would like to reproduce, and a simplified version m, onto which we apply the generated normal map for real-time rendering. Four main steps are involved: first an octree data structure is adaptively built around m, driving its refinement with the normal field of M. Second, a ray casting procedure is employed to accurately sample the normal field at each cell of the octree. The resulting normal octree-texture is then encoded as a regular 2D texture for optimized storage and manipulation by the GPU. Finally, at rendering time, an efficient adaptative octree traversal algorithm is performed on the fragment shader stage to achieve interactive framerates.

2 Related Work

Appearance-Preserving Simplification: Replacing a detailed mesh with a coarse one together with several textures storing the

[*]e-mail: julien.lacoste@univ-pau.fr
[†]e-mail: boubek@cs.tu-berlin.de
[‡]e-mail: bruno.jobard@univ-pau.fr
[§]e-mail: schlick@labri.fr

normal field has been independently introduced by Cignoni et al. [Cignoni et al. 1998; Cignoni et al. 1999] and Cohen et al.[Cohen et al. 1998].

In the work of Cignoni et al. [Cignoni et al. 1998; Cignoni et al. 1999], textures are built for a coarse mesh m, by sampling details on the full resolution mesh M. The user defines a fixed sampling rate, used for all triangles of m, then for each sample, the closest point on M is found. All the normal maps of each triangle are packed in a texture atlas, eventually sheared to match the best fitting shape in the texture. As the sampling rate is the same for each triangle, the texture quickly becomes large when this rate is increased, while low detailed areas are over-sampled. To avoid the storage of high resolution textures, an over-sampling technique is used. It increases the quality of low resolution textures, by computing an average value for each texel.

In the work by Cohen et al. [Cohen et al. 1998], a simplification algorithm is applied on M to produce m as well as a set of textures encoding either normals or surface colors. To generate the texture map, each polygon of the initial mesh is projected on a 2D plane. Then an edge collapse simplification is performed to create m, and the texture coordinates of m are computed during this simplification. The authors introduce a texture deviation metric which minimizes texture distortion during the simplification. Compared to the previous one, the main drawback of this technique is that it requires a parameterization of the initial mesh M which is not adapted to highly detailed objects.

Normal mapping has been employed to obtain smooth transition when streaming progressive meshes [Sander et al. 2001]. The use of ray-casting to sample a normal field has also been studied in [Rogers 2003] and [Sander et al. 2000]. A classification and a comparison of these methods is proposed in [Tarini et al. 2003].

Another application of appearance preserving techniques is the visualization of very large point clouds, acquired by laser range scanners, for instance. Boubekeur et al. [Boubekeur et al. 2005] proposed a direct point cloud to normal map conversion which works in streaming and generates a coarse mesh combined with a set of normal maps fitting a given memory budget, by using out-of-core simplification, local triangulation and hierarchical diffusion.

Octree-Textures: The idea of octree-textures has been simultaneously introduced by DeBry et al. [(grue) DeBry et al. 2002] and Benson and Davis [Benson and Davis 2002]. Both papers proposed to encode color map in an octree [Samet 1989], which avoids the complex construction of 2D parameterization on the input mesh. Such methods are particularly well suited for interactive painting on 3D objects, where the intrinsic adaptive sampling of the octree structure reduces the waste of memory exhibited by fixed-resolution 2D maps. More recently, Lefebvre et al. [Lefebvre et al. 2005] and Lefohn et al. [Lefohn et al. 2006] have proposed GPU implementation of octree-textures, encoding them in simple 2D or 3D textures, adapted to efficient access by the fragment shader.

In the next sections we describe an original approach that achieves precise appearance-preserving simplifications with adaptative normal field sampling, compact storage and fast GPU rendering by taking advantage of the hierarchical nature of the octree-texture data structure.

3 Appearance Preserving Octree-Texture

3.1 Octree Construction

In the framework we propose, the user first provides two versions of a given object: M, the original full resolution mesh, and m, the simplified version which may be obtained by any existing mesh simplification technique. Our goal is then to build an *appearance preserving octree-texture* (**APO**) for m, by sampling the normal field defined by M. In order to capture all the fine scale variation of this normal field, we apply the following adaptive process to generate the APO.

In a first step, an initial coarse octree with a restricted depth is built around m. At each leaf cell C of this coarse octree, a list called t^C (resp. T^C) is created to store the triangles of m (resp. M) that belong to cell C. An error criterion is then used to check if further subdivision is required for this leaf cell. The use of an initial octree with a restricted depth avoids subdivision tests on coarser levels, since the details will be in leaves at finer levels. Our error criterion is based on the $L^{2,1}$ metric introduced in [Cohen-Steiner et al. 2004]:

$$L^{2,1}(C) = \sum_{T_i \in T^C} ||N_{T_i} - N_{T^C}||^2$$

where N_{T^C} is the average normal vector of all the triangles that belong to the list T^C. The $L^{2,1}$ metric measures the normal field variation in the cell. Using this metric allows to catch all small-scale high frequency features that must be reproduced in the APO. If the error criterion is above a user-provided threshold, the leaf cell is subdivided into 8 sub-cells, but only the children which actually contain some triangles of m are kept. Indeed, as the octree-texture will be used on the mesh m, all cells that do not intersect m are useless for the rendering step. After the subdivision, both triangle lists T^C and t^C must be updated, simply by moving each triangle in any sub-cell it belongs to. As we do not necessarily keep all the sub-cells, some triangles from T^C may not be entirely enclosed in the sub-cells; in that case, we simply keep the corresponding triangles in T^C, as they will be needed during the field sampling step. This subdivision scheme is recursively performed and is stopped either when the error criterion is fulfilled, or when a user-provided maximum depth is reached (see Figure 10).

3.2 Normal Field Sampling

Once the octree has been built, each cell must be filled with a representative sample of the normal field of M. To get an accurate sampling, we use a ray casting procedure. More precisely, for a given leaf cell C, a ray R_i is cast for each triangle $t_i \in t^C$. The origin of R_i is obtained by projecting the center of the cell C on t_i, and as in [Sander et al. 2000], the direction of R_i is the interpolated normal on t_i obtained at the ray origin, to avoid sampling discontinuities that may appear between neighboring triangles (see Figure 2). To speedup the ray casting procedure, the octree is used

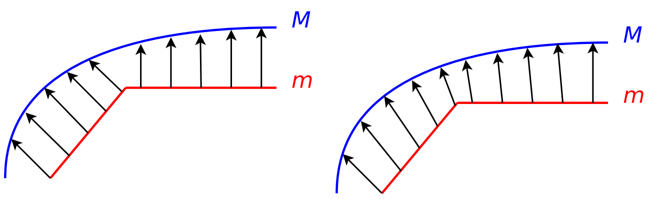


Figure 2: *Throwing rays with interpolated normal (right) rather than the triangle normal as directions avoids sampling discontinuities.*

as a space partitioning structure. For that, we first find all the cells of the octree intersected by R_i within a chosen maximum distance. The intersected cells can be either leaf cells or internal cells whose T list is non-empty. These cells are then sorted according to distance from the ray origin to the intersection of the bounding sphere surrounding the cell. The algorithm loops over these cells to find the intersection. When a cell can't own a closer intersection than the one previously found, the algorithm stops. If the ray fails at intersecting any triangle of M, we use a similar alternative as proposed in [Sander et al. 2000] by searching the closest point of the ray origin on M. The normal found at the intersection on M is accumulated in the corresponding leaf cell C. At the end, the accumulated vector is normalized to obtain the representative normal N_c of C. Once a representative normal has been computed for all leaf cells, the normals at the internal nodes are computed in a bottom-up process, by averaging the normals of their children. The octree levels equiped with average normals will provide at render time a built-in mipmapping mechanism (see section 5.2).

4 2D Encoding of APO

Once the APO has been constructed, as detailed above, it is first converted into a 1D array, by enumerating nodes in a breadth-first order. As all siblings of a node are thus contiguous in this 1D array, forming a brood, we only need to store a pointer to the first child of each node [Hunter and Willis 1991] (see Figure 3). Moreover,

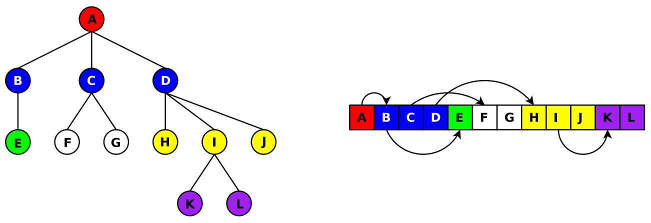
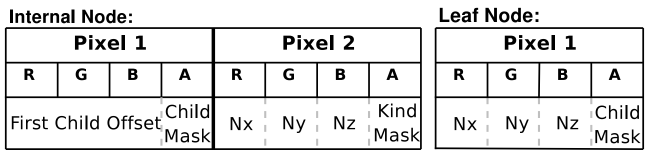


Figure 3: *Encoding an octree in a 1D array by breadth-first ordering. Arrows show the pointer between parent nodes and their first child.*

to avoid waste of space in the texture, we go back to the original idea of Benson and Davis [Benson and Davis 2002] by only storing non-empty nodes. At each node, a mask of eight flags, one for each child, defines whether a child is present (flag set to 1) or not (flag set to 0). A leaf node can thus be easily identified, as all its flags will be set to 0. For internal nodes, we add another mask of eight flags, to tell whether each child is a leaf (0) or a node (1). This second mask will be necessary during the octree traversal, in order to know how many texel jumps are required to access a particular child in the children list. For internal nodes, we also need to store the pointer to the first child. This pointer could become large when octrees have a maximum depth greater than 10. We decide to store not a pointer, but a local offset to the brood, which can be safely quantized to three bytes. So, by encoding each normal vector on three bytes, we actually need four bytes for a leaf node, and eight bytes for an internal node. By using an RGBA texture, a leaf node is thus represented by one texel, while an internal node needs two texels, as shown on Figure 4. As some GPU architectures have a relatively low bound on the size of 1D textures, the 1D array is finally converted into a 2D texture, by cutting the array into slices of 2^K texels, where K is chosen according to the size of the 1D array, to get 2D textures that are close to squares. Figure 5 presents an example of the final 2D encoding of an APO. As we only keep non-empty nodes in the broods, our encoding scheme is more compact than the one presented in [Lefebvre et al. 2005]. Moreover, our data structure directly encodes intermediate levels, while [Lefebvre et al. 2005] would require a second texture for that. Globally, our encoding needs about 40% less storage size to store equivalent data.



Internal Node:

Pixel 1				Pixel 2			
R	G	B	A	R	G	B	A
First Child Offset			Child Mask	Nx	Ny	Nz	Kind Mask

Leaf Node:

Pixel 1			
R	G	B	A
Nx	Ny	Nz	Child Mask

Figure 4: *Encoding of the APO elements. A leaf node requires 3 bytes for the normal and 1 for the child mask (which equals 0). A node requires 3 bytes for the normal, 3 for the first child offset, 1 for the child mask (telling which children exist) and 1 for the kind mask (telling whether the children are leaves or nodes).*

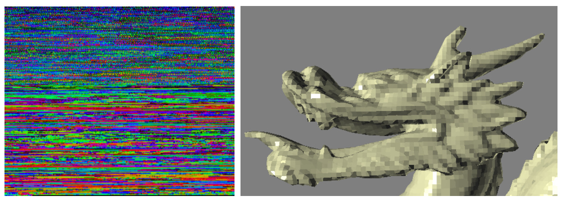


Figure 5: ***Left:*** *A low resolution 256x221 2D encoded APO.* ***Right:*** *Same APO mapped over a simplified mesh.*

5 GPU Rendering

The last step of our framework involves the rendering step, performed on the GPU by a single-pass *fragment shader*. After having transmitted the texture width, the fragment shader loops over the provided position P of each fragment and computes its shading with the associated normal defined in the APO. This section focuses on this last rendering step. Note that all listings provided in this section are written in GLSL.

5.1 Octree Traversal

Starting at the root, the APO is traversed top-down, until encountering the leaf node containing P. At each internal node, we have to determine which of its children contains P as well as the offset used to access this child from the current position. The first node to be read is the root, which is located as index 0 in the array. As we work in the unit cube, the root is centered around $(0.5, 0.5, 0.5)$ and its width is 1.0. These two parameters will be useful for the traversal, and are updated at each iteration, as well as the current node $index$. We now describe more precisely all steps involved in the octree traversal, performed after the following initialization code:

```
int index = 0;
vec3 octEltCenter(0.5,0.5,0.5);
float cellWidth = 1.0;
```

(a) Finding the next node: At each level, a texture fetch is performed to obtain the first texel corresponding to the current node (see Figure 4). If the child mask of this texel is null, the node is a leaf and the traversal stops. Otherwise, the number of the child to process is obtained by expressing P in the frame of *octEltCenter*:

```
int getChildIndex(vec3 octEltCenter, vec3 P) {
  vec3 dep = octEltCenter - P;
  dep = sign(dep);
  dep = step(0.0, dep);
  return int(dep.x*4.0 + dep.y*2.0 + dep.z);
}
```

The *sign* function replaces negative values by -1 and positive ones by 1, while the *step* function replaces negative values by 0. Once processed, the *dep* vector has either 1 or 0 at each component, which can be used as a binary code. The last line thus directly returns the number of the child to be accessed. This process avoids browsing and intersecting children bounding boxes.

Once this child index has been obtained, we have to efficiently update *octEltCenter*. We start by applying just the *sign* function on the vector *dep* which gives the direction toward the next child. Then, we adjust the length of this vector, in order to obtain a quarter of the current node width for each component. Adding this vector to the current node center gives the next node center. The procedure is illustrated in 2D in figure 6.

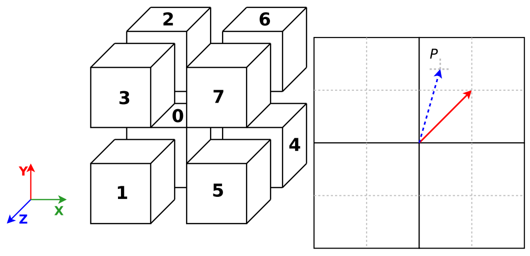


Figure 6: *According to the children's ordering (left), the relative coordinates of P (blue dashed arrow) can be converted into the corresponding child index. The full red arrow has been obtained from the blue arrow by using the sign function and adjusted to reach the center of cell.*

(b) Computing the node offset: Finally, we must compute the offset to the desired child. If *index* is the current node index, the index *indexC* of its child c is obtained with following formula:

$$\text{indexC} = \text{index} + \text{offset(index)} + \text{offsetToCthChild(index,c)}$$

where *offset(index)* is the offset to the first child, and *offsetToCthChild(index,c)* is the extra offset to add in order to access the c^{th} child. The first child offset is retrieved by converting the RGB channels of the first texel node to an integer (see Figure 4). Then to adjust the offset, we access the second node texel (which is at $index + 1$ in the octree-texture) and get the *kind mask*. The function *adjustOffset* returns a boolean, telling whether the desired child exists or not, and stores the extra offset to the c^{th} child in the *extraOffset* parameter.

```
bool adjustOffset(float childrenMask, float kindMask,
                  int cth, out int extraOffset) {
 int it, count = 0;
 float modc, modk;
 extraOffset = 0;
 for(it = 0 ; it<8 ; ++it) {
   // count only children before the one we want
   if (it < cth)  {
     modc = mod(cmask, 2.0); modk = mod(kmask, 2.0);
     extraOffset += int((modk + 1.0) * modc);
     kmask -= modk;kmask /= 2;cmask -= modc;cmask /= 2;
   }
 }
 // test if the wanted child exists
 return (mod(cmask, 2.0) == 1.0);
}
```

In this function, the sum $modk + 1$ gives for each child the number of texels to jump (1 for leaf nodes and 2 for internal nodes). Multiplying this sum by $modc$ allows us to only count existing children, as $modc$ will be zero for an empty child. Successive division by 2 generates a bit shift process, while the modulo 2 operation returns the values of individual bit flags. Note that, while this method works on all devices, direct bitwise operations can be used when supported by the graphics hardware.

5.2 Volumetric Mip-Mapping

Until now, the only stopping condition during the octree traversal occurs when a leaf is encountered. We can further improve rendering performances and quality considering that:

- for far or dense objects, a fragment may correspond to a whole subtree of our octree-texture, so stopping the recursion at this internal node would make the rendering faster without quality degradation, as it discards the (possibly large) part of subpixel traversal operations.
- full traversal may cause temporal and spatial aliasing, which can be partially prevented by using the average normals stored on internal nodes.

We propose to use the hierarchical nature of the APO to avoid this sub-pixel precision. A new condition is added in the loop, to stop the traversal before a leaf is reached when the object-space size corresponding to the current fragment is larger than the diagonal of the current cell. This can easily be computed by estimating the number of pixels crossed by the image space projection of the cell diagonal:

$$NbPixels = \frac{\sqrt{3 \times 2^{-d}}}{cameraDist} \times \frac{screenSize}{2 * tan(\frac{FOV}{2})}$$

where d is the depth of the current node and FOV is the angle defining the field of view. When $NbPixels$ is less than one, the traversal stops and the average normal stored in the last traversed internal node is used to compute the shading for the current fragment. This volumetric mip-mapping improves both quality (by filtering the texture under minification) and efficiency (by reducing the average traversal cost).

5.3 Texture Filtering

While the mip-mapping performs efficient filtering under minification, additional filtering has to be used to smooth the APO under magnification. Just as in [Lefebvre et al. 2005] a bilinear interpolation can be performed at each cell, between the eight values stored at the cell vertices. However, our process is more efficient, as we do not need to ensure that all eight neighbors exist as in [Lefebvre et al. 2005]. Indeed, when we try to access a neighboring cell that does not exist, the traversal loop is simply stopped and the average normal stored at the last traversal node is used for the interpolation.

The use of texture filtering greatly improves the visual quality, as shown on Figure 7, but at the cost of slower performance, as the octree has to be fully traversed eight times, to get the value of the eight neighboring cells. Several solutions may be envisaged to improve the performance. For instance, one may store at each node, the gradient of the normal vector in addition to the normal itself, to get a first order approximation (instead of a zeroth order) at the center of each cell, which may be sufficient for filtering without access to neighboring cells. Another solution would be to preprocess the octree, in order to add an additional offset for each internal node, that links to the closest parent node which contains all eight neighbors involved in the bilinear filtering, so that the eight traversals do not need to restart from the root. But all these techniques clearly require additional storage in the data structure, and thus involve a traditional computation vs storage trade-off. In Section 6, we describe an alternative optimization providing significant speed-up.

6 Results and Performances

The system has been implemented under Linux 2.6 with OpenGL and GLSL. Performances have been measured on an AMD Athlon 3500, with 2GB of memory and an nVidia GeForce 8800 GTX.

Stanford Dragon	Original Simplified	874 414 triangles 3 000 triangles		**XYZRGB Dragon**	Original Simplified	7 218 906 triangles 15 000 triangles	
Octree	**Coarse depth:** 5 **Max depth:** 13			**Octree**	**Coarse depth:** 5 **Max depth:** 13		
Error Bound	0.3	0.1	0.05	**Error Bound**	0.3	0.1	0.05
Creation Time	36 s.	54 s.	1'19 min	**Creation Time**	5'04 min	7'57 min	10'48 min
Texture res.	1024×389	1024×770	2048×611	**Texture res.**	2048×1475	4096×1650	4096×2631
Texture size	1.1 MB	2.3 MB	3.6 MB	**Texture size**	9.2 MB	20 MB	30.6 MB
FPS - 512×512	210	190	170	**FPS -** 512×512	150	130	120
FPS - 1024×1024	75	60	58	**FPS -** 1024×1024	56	47	46

Table 1: *Preprocess timing, memory consumption and (optimized) rendering frame rates for various screen resolution.*

We have performed a full evaluation of our approach on various models, ranging from 500k to 10M polygons, and exhibiting different topological genus and different feature distributions. We use an algorithm based on the Quadric Error Metric [Garland and Heckbert 1997] for simplifying the original meshes to a prescribed resolution. This error metric is perfectly adapted to our application: it smoothes out all the small-scale high frequency features, that will later be visually reproduced by the normal vectors stored in the APO, while preserving the global shape even under strong reduction rates. Table 1 presents a complete performance analysis, when using different settings on the Stanford Dragon and the nine time bigger XYZRGB Dragon. Table 2 compares the performance obtained on various models, by using standard parameter settings (error bound at 0.05 and screen resolution at 1024×1024). Note that the framerate is measured in the worst case for the APO shading, i.e. when the object nearly covers all the pixels of the screen. Regarding the rendering performance reached, the larger the original mesh, the more interesting our method. For instance, we failed at obtaining interactive framerates for the larger meshes when rendered directly (indexed on-GPU vertex buffers), even at low screen resolution, while our appearance preserving rendering is real-time with the same setting. Figure 8 compares four versions of the same object: the original full resolution mesh (1.7M triangles), the simplified mesh (5K triangles), the simplified mesh with a 1024×544 APO, and the same with a 4096×2467 APO. Basically the error introduced by the APO has two main sources: the quality of the initial mesh simplification, and the error criterion used during construction of the APO. The first error is mostly visible on object silhouette, while the second one impacts the visible details inside the object. Finally, Figures 9 and 11 present additional examples of visual enrichment performed by the APO data structure. For all examples that we tested, we always achieved a good visual quality compared to original objects. However, we plan to study the error in a more formal way, by using some of the techniques presented in [Tarini et al. 2003].

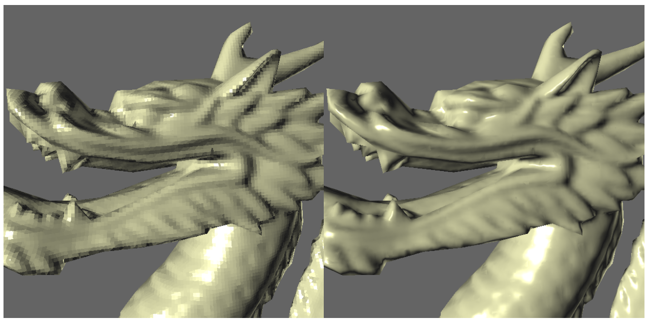


Figure 7: ***Left:*** *Octree normal mapping without filtering.* ***Right:*** *With filtering.*

Discussion: As already mentioned above, the main benefit of using a volumetric texture, is to avoid the expensive computation of surface parameterization, but also to avoid the storage of the (u, v) texture coordinates. In order to control the construction of the APO, we have chosen to expose only a reduced set of intuitive parameters to the user: the initial and final maximum depths of the octree, and the error threshold used by the subdivision process. By construction, an APO automatically provides an adaptative sampling of the normal field: areas with fewer details will be covered with bigger octree cells, while regions exhibiting high frequency features will automatically generate an higher number of smaller cells, thanks to the $L^{2,1}$ error metric. Compared to a classical 2D normal map, this adaptive sampling avoids a large number of useless texels. The technique allows interactive rendering of highly detailed meshes that may not fit in-core memory without normal mapping.

The main drawback of the technique is the cost of texture access. If a fragment is rasterized for a leaf at depth d, there will be $(2d - 1)$ accesses to reach the leaf. Even if the leaves addressed by adjacent fragments may be neighbors in the texture, our process requires each fragment to restart from the root, which induces accessing the texture at very different locations for a single traversal. Thus, when the object is viewed in a large close-up, the framerate decreases as there are more traversals to perform by the fragment shader. Likewise, using texture filtering and its eight octree traversals does not allow a realtime framerate, as it falls around 20 FPS (resp. 7 FPS) when the object is relatively close at 512x512 (resp. 1024x1024) screen resolution.

There are other limits to the construction of the APO. One is that the geometry of m must not introduce a too large error compared to M. Actually, as we only keep leaf nodes around m, these nodes do not always intersect M, and then they contain no data to compute the $L^{2,1}$ metric. If m is not a "good" simplification of M, this case can even happen at internal nodes and thus local features of M might be lost in the octree-texture. By using the Quadratic Error Metric, we have not explicitly encountered that case, but one preventive solution would be to twist the QEM, to account for that potential loss of data.

Another limitation is that thin two-sided surface could not be rendered. Actually, if two sides of a surface end up after maximal

Model	**Size**	**Preproc.**	**Texture**	**FPS**
Stan. Dragon	$874k$ / $3k$ tri.	$1m19s$	$3.6MB$	58
Grog	$1.7M$ / $5k$ tri.	$3m42s$	$12.3MB$	48
Neptune	$4M$ / $15k$ tri.	$6m3s$	$10.3MB$	68
XYZ Dragon	$7.2M$ / $15k$ tri.	$10m48s$	$30.6MB$	46
Statue	$10M$ / $20k$ tri.	$18m18s$	$46MB$	58

Table 2: *Comparison with different models, with original and simplified size. The error bound is set to 0.05 and the screen resolution is* 1024×1024.

subdivision in a same octree leaf, normals will be averaged and the averaged one could be near zero. In the actual structure, there is no way to avoid these problems, as we store only one normal per octree leaf. One solution is to store a second texel for the leaves representing two sides, and choose the correct one at rendering step, using a dot product operation. A third mask has to be added in each node to tell which children are represented by two texels and thus knowing if one or two texels must be jumped for a particular leaf during the octree traversal. So a third texel is necessary for each node, the remaining three bytes could be used as the second side averaged normal. But this solution involves more texture accesses, as well as more computation to find the correct offset or normal to apply.

Optimization: The algorithm described in Section 5 is general enough for handling octrees of arbitrary topologies, but remains bottlenecked by the number of texture accesses. To reduce that number, the only solution is to be able to find the leaf node more efficiently than traversing the whole octree from the root. One solution that we have implemented, is to force the first levels of the octree to be *complete*, by explicitly storing all children of a given node, even if some of them contain no geometry. By limiting this octree completion to a small depth (5 in our implementation), only a small amount of additional memory is required, compared to the full octree size. Starting from the root, the location of each child can then be computed without the recursive process, avoiding several texture accesses and thus, improving performances. For instance, the models of Table 1 have been generated with a depth ranging from 5 to 13, which means that the number of texture access per-fragment may reach 13 in the worst case. With our optimization by octree completion, we avoid the 5 first texture accesses, and improve the framerate by 30% to 40% for a negligible memory overhead. In a way, this optimization leads to the generation of a forest of small octrees organized in a 3D grid instead of a global octree. The depth choosen for octree completion is therefore an intuitive *memory-vs-performance* tuning parameter. Note that choosing a deeper completion level quickly increases the memory cost of this optimization. In practice, we observed that maintaining a completion less than half the maximum depth was a good trade-off.

7 Conclusion and future work

We have proposed an efficient appearance-preserving method based on octree-textures for normal mapping. While previous work, such as [Cignoni et al. 1998; Cignoni et al. 1999], had a fixed sampling rate, our algorithm adaptively captures the normal field and avoids the over-sampling in low detailed areas. Our volumetric texture also avoids the creation of a 2D parameterization necessary in other methods [Cohen et al. 1998]. We have also introduced a compact GPU storage scheme for octree textures and an efficient traversal method.

Our main direction for future work is to explore usage of other structure to encode normal maps, may be using lower-dimensional approach as proposed in [Boubekeur et al. 2006] and [Lefebvre and Dachsbacher 2007]. By using the scaled offsets presented in [Lefebvre and Hoppe 2007] we could obtain a more compact encoding. Even though the proposed construction needs a complete tree, which is impractical for high detailed octrees, the use of autumnal trees or data quantization could improve the compactness and the cost of rendering.

We also plan to develop a better filtering method between leaves located at different depths of the octree, possibly using a frequency analysis [Han et al. 2007] or ideas presented in [Ljung et al. 2006], as well as a hierarchical normal quantification inspired by point-based rendering methods.

References

Benson, D., and Davis, J. 2002. Octree textures. *ACM Siggraph*.

Boubekeur, T., Duguet, F., and Schlick, C. 2005. Rapid visualization of large point-based surfaces. *Eurographics VAST*.

Boubekeur, T., Heidrich, W., Granier, X., and Schlick, C. 2006. Volume-surface trees. *Eurographics*.

Cignoni, P., Montani, C., Rocchini, C., and Scopigno, R. 1998. A general method for recovering attribute values on simplifed meshes. *IEEE Visualization*.

Cignoni, P., Montani, C., Rocchini, C., Scopigno, R., and Tarini, M. 1999. Preserving attribute values on simplified meshes by resampling detail textures. *The Visual Computer 15*.

Cohen, J., Olano, M., and Manocha, D. 1998. Appearance-preserving simplification. *ACM SIGGRAPH*.

Cohen-Steiner, D., Alliez, P., and Desbrun, M. 2004. Variational shape approximation. *ACM SIGGRAPH*.

Garland, M., and Heckbert, P. 1997. Surface simplification using quadric error metrics. *ACM SIGGRAPH*.

(grue) DeBry, D., Gibbs, J., Petty, D. D., and Robins, N. 2002. Painting and rendering textures on unparameterized models. *ACM Siggraph*.

Han, C., Ramamoorthi, B. S. R., and Grinspun, E. 2007. Frequency domain normal map filtering.

Hunter, A., and Willis, P. 1991. Classification of quad-encoding techniques. *Computer Graphics Forum 10*, 2 (June).

Lefebvre, S., and Dachsbacher, C. 2007. Tiletrees. *I3D*.

Lefebvre, S., and Hoppe, H. 2007. Compressed random-access trees for spatially coherent data. In *EGSR*, Eurographics.

Lefebvre, S., Hornus, S., and Neyret, F. 2005. *GPU Gem's 2: Octree Textures on the GPU*.

Lefohn, A. E., Kniss, J., Strzodka, R., Sengupta, S., and Owens, J. D. 2006. Glift: Generic, efficient, random-access gpu data structure. *ACM Transaction on Graphics*.

Ljung, P., Lundström, C., and Ynnerman, A. 2006. Multiresolution interblock interpolation in direct volume rendering. *EUROVIS*.

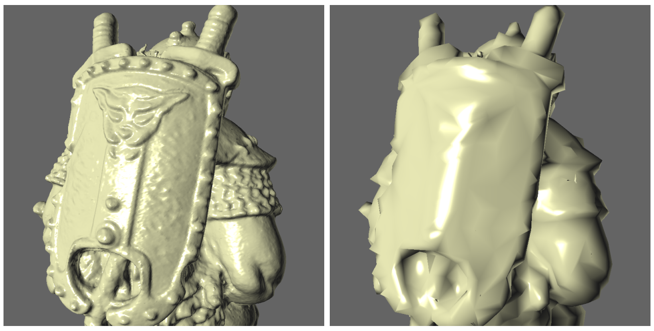
Rogers, D. 2003. All the polygons you can eat. *GDC*.

Samet, H. 1989. *Quadtree, Octrees, and Other Hierarchical Methods*. Addison Wesley.

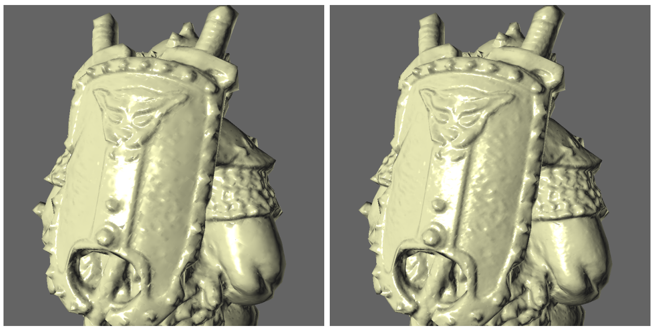
Sander, P. V., Gu, X., Gortler, S. J., Hoppe, H., and Snyder, J. 2000. Silhouette clipping. *ACM SIGGRAPH*.

Sander, P., Snyder, J., Gortler, S., and Hoppe, H. 2001. Texture mapping progressive meshes. *ACM SIGGRAPH*.

Tarini, M., Cignoni, P., and Scopigno, R. 2003. Visibility based methods and assessment for detail-recovery. *IEEE Visualization*.

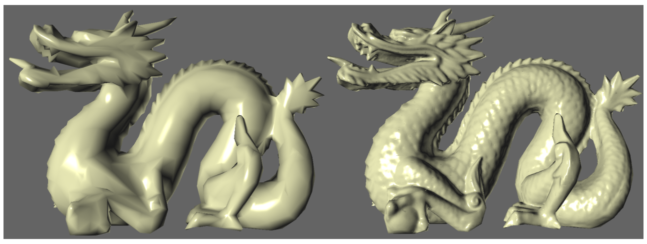


(a) Original (1.7M tri.) and simplified (5k tri.) meshes

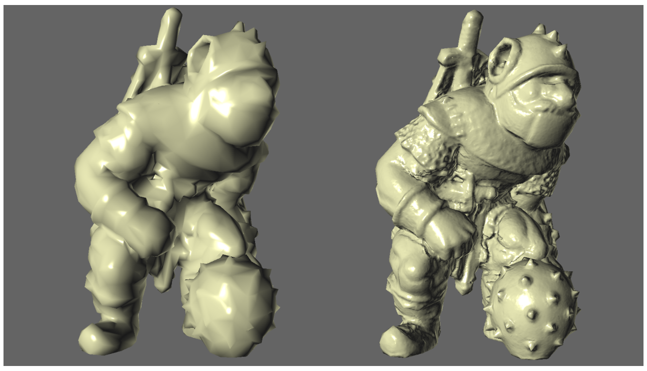


(b) Simplified mesh with APO normal mapping

Figure 8: ***Top:*** *Geometry simplification.* ***Bottom:*** *Simplified mesh with APO normal mapping, with two different resolution:* 1024×544 *(epsilon 0.8) and* 4096×2467 *(epsilon 0.01).*



(a) Stanford Dragon, original mesh has 874K triangles, the simplified one 3K. Texture dimension: 2048x611



(b) Grog, original mesh has 1.5M triangles, the simplified one 5K. Texture dimension: 4096x1046

Figure 9: *Examples of APO normal mapping.*

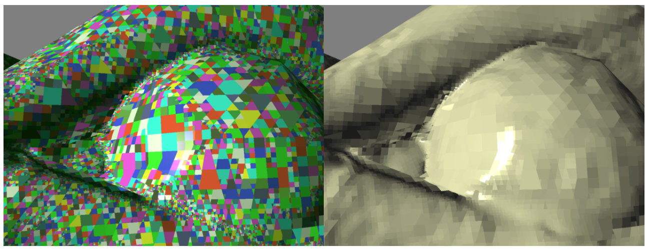
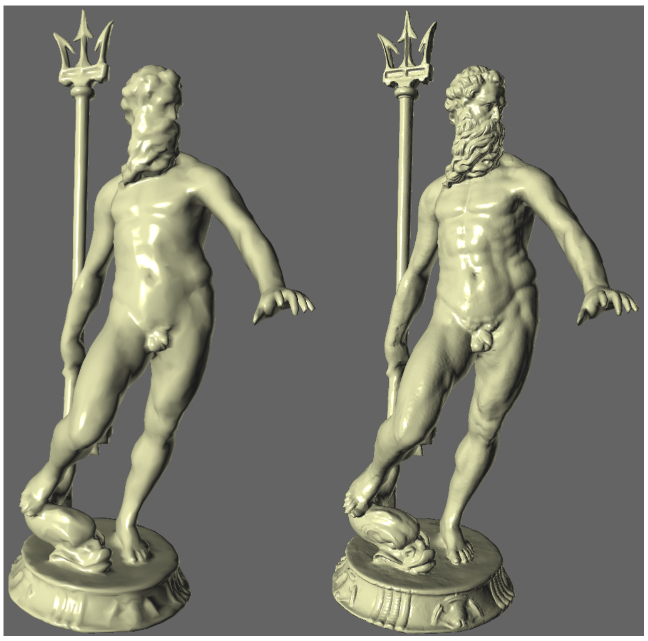
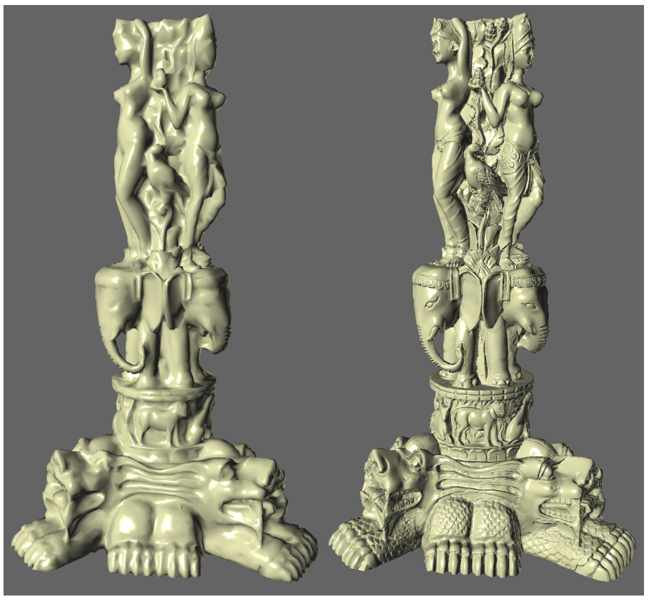


Figure 10: ***Left:*** *close-up on an octree-textured mesh ; the fragments have been colored depending on the cell they belong to in order to reveal the adaptive sampling rate.* ***Right:*** *the associated normal mapping.*



(a) Neptune, original mesh has 4M triangles, the simplified one 15K. Texture dimension: 2048x1766



(b) XYZRGB Statue, original mesh has 10M triangles, the simplified one 20K. Texture dimension: 4096x3705

Figure 11: *Additional examples of APO normal mapping.*