Real-Time High-Quality View-Dependent Texture Mapping using Per-Pixel Visibility

Damien Porquet*
MSI - University of Limoges

Jean-Michel Dischler†
LSIIT - University of Strasbourg

Djamchid Ghazanfarpour‡
MSI - University of Limoges

(a) Original mesh; 1.1 M triangles; 50 FPS

(b) Simplified mesh; 6 K triangles; 2000 FPS

(c) Our method applied to the simplified mesh; 606 FPS.

Figure 1: Our method applied to the Buddha mesh from Stanford Graphics Laboratory.

Abstract

We present an extension of View-Dependent Texture Mapping (VDTM) allowing rendering of complex geometric meshes at high frame rates without usual blurring or skinning artifacts. We combine a hybrid geometric and image-based representation of a given 3D object to speed-up rendering at the cost of a little loss of visual accuracy.
During a precomputation step, we store an image-based version of the original mesh by simply and quickly computing textures from viewpoints positionned around it by the user. During the rendering step, we use these textures in order to map on the fly colors and geometric details onto the surface of a low-polygon-count version of the mesh.
Real-time rendering is achieved while combining up to three viewpoints at a time, using pixel shaders. No parameterization of the mesh is needed and occlusion effects are taken into account while computing on the fly the best viewpoints for a given pixel. Moreover, the integration of this method in common real-time rendering systems is straightforward and allows applying self-shadowing as well as other z-buffer effects.

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

*e-mail: porquet@msi.unilim.fr
†e-mail: dischler@dpt-info.u-strasbg.fr
‡e-mail: ghazanfarpour@unilim.fr

Keywords: Real-Time Rendering, Image-Based Rendering, View-Dependent Texture Mapping, Graphics Hardware

1 Introduction

Rendering of scenes composed of a huge set of polygons at real-time rates is an important research topic in computer graphics. In this context, the realism of 3D objects has been greatly improved during the last decade, and particularly in the domain of real-time rendering. This is mainly due to the fact that graphics hardware is more and more powerful, beeing able to render millions of triangles per second, allowing the user to display very complex meshes. Nevertheless, needs for realism are constantly above hardware capacity. This requires to find alternative approaches in order to reduce the rendering cost of such meshes. For instance, it is now possible to render a mesh composed of one million of triangles at 50 FPS. However, a real scene composed of several complex objects quickly decrease frame rate below an unacceptable threshold.

In this paper, we propose to extent View-Dependent Texture Mapping (VDTM) [Debevec et al. 1998] to provide a solution to this problem. Figure 1 shows the result of our method applied to the Buddha mesh from the Stanford Graphics Laboratory which is composed of 1.1 million triangles. Conventional rendering gives 50 FPS, using NVIDIA GeForce 6800GT graphics card, and approximately 600 FPS using our method, with very little discernible difference.

The aim of our method is to cumulate the advantages of mesh simplification with those of an image-based representation of the mesh, to obtain the quality of the real mesh surface, while only sending the simplified mesh and some textures to the graphics card. The simplified mesh provides a rough approximation of the scene surface location. The medium scale details, lost during simplification, are recovered using textures computed from some adequate viewpoints positionned around the real mesh. The main differences

with other methods, and in particular VDTM, are:

1. We do not apply any interpolation among pixel colors, thus avoiding blurring effects.

2. We do not replace simplified parts of the geometry by mapping precomputed texture maps representing underlying complex surface. Instead, we use the simplified mesh as basis to retrieve the original surface points stored in the pre-computed viewpoints. This avoids to maintain a parameterization of the mesh.

3. We do not need to subdivide the simplified mesh polygons to reduce blurring. Instead, we exclude bad viewpoints from rendering on the fly.

4. Since our technique reconstructs for each pixel a good approximation of the real mesh surface, we can further relight the scene on the fly with standard hardware based shading models. This includes accurate shadowing effects.

The remaining parts of the paper are organized as follows. We describe the previous work in section 2. In section 3, we present an overview of the method. Then we describe in section 4 the preprocessing step which mainly consists in gathering the reference viewpoints. The core of our surface reconstruction algorithm is described in section 5, followed in section 6 by a description of our real-time implementation using shaders. Some results of our work are presented in section 7, finally followed by conclusions and future work.

2 Previous Work

This paper is aimed towards real-time rendering of complex 3D objects, which are composed of a huge number of triangles, using hybrid Image-Based Rendering (IBR) and geometrical approach. A lot of work has been done in this context.

Geometrical mesh simplification methods [Cohen et al. 1998; Sander et al. 2001; Hoppe 1996; Heckbert and Garland 1997; Luebke and Erikson 1997] can be used to decrease the rendering cost of a given mesh when seen from far distances, but this equally reduces its realism.

Displacement mapping methods like [Cook 1984; Moule and McCool 2002] add surface details on top of the simplified mesh while only sending two dimensional maps containing height data to the graphics card. But the reconstructed triangles need to be conventionally projected on the screen in order to be displayed, which does not reduce substantially the graphics card task.

Hybrid geometric and image-based methods like [Wang et al. 2003; Hirche et al. 2004] can reproduce the surface displacement while computing it per-pixel, with graphics hardware, but either the method cannot be applied to more than one mesh because of rendering time [Hirche et al. 2004], or it is restricted to small displacement patterns, because of graphics memory cost [Wang et al. 2003].

Commonly, texture and bump maps can be applied to a simplified object surface in order to improve its appearance [Cohen et al. 1998], but the resulting surface is always flat: mapped images are unable to take into account parallax effets due to underlying relief. Moreover, this mapping approach requires to maintain a parameterization.

More generally, pure image-based methods [Levoy and Hanrahan 1996; Gortler et al. 1996; Buehler et al. 2001; Chen et al. 2002] can be used as alternative rendering techniques to avoid dealing with a scene of a high geometrical complexity. Unfortunately, these methods require a lot of memory, have difficulties to handle illumination variations and remain complex to manipulate. For instance, data decompression is a CPU intensive task, not well suited for a real-time rendering system. Furthermore, relighting or use of multiple instance of objects rendered with theses methods is hard and sometimes impossible [Chen et al. 2002]. This makes their integration in a classical real-time rendering system a very difficult problem.

Hybrid methods combining image-based and geometrical approaches [Debevec et al. 1996; Pulli et al. 1997; Debevec et al. 1998; Cohen et al. 1998; Pighin et al. 1998; Oliveira et al. 2000; Yamakasi et al. 2002; Décoret et al. 2003] have also known some noticeable success, but all of these methods still raise several problems like blurring, popping and skinning or they have some restrictions concerning the nature of the mesh complexity.

Among these last works, the two main approaches related to our method are Appearance Preserving Simplification [Cohen et al. 1998] and View-Dependent Texture Mapping [Debevec et al. 1996; Debevec et al. 1998]. We therefore describe these two approaches with more details in the next two paragraphs.

Appearance Preserving Simplification In [Cohen et al. 1998], a complex mesh is first simplified with purely geometrical method, then, in order to compensate lost details, a bump texture representing the original surface is applied to the simplified mesh. The main drawback in using such a method in real-time applications is that the texture needs to be reconstructed at each level of details (LOD). The texture can be precomputed for each LOD, but this will exclude for using dynamic simplification, such as in [Hoppe 1996]. Furthermore, the well known artifact of texture mapping is that visible relief is only appearance: mesh surface remains flat, which can be seen at grazing angles. Also, visible simplified mesh edges lead to unrealistic appearance.

In comparison, our method does not require to generate bump maps, thus avoiding to maintain a parameterization of meshes. It only needs a raw approximation of the original surface to recover relief stored in reference views. Moreover, this approximate mesh is able to change during rendering because it is only used as a starting point for our visibility computations.

View-Dependent Texture Mapping VDTM was first introduced by Debevec *et al.* [Debevec et al. 1996] in off-line rendering context, and extended to real-time in [Debevec et al. 1998]. The main goal of VDTM is to obtain novel views of a scene while only using a set of reference photographs (reference viewpoints) and a simplified mesh of the scene geometry. This mesh is constructed by-hand from photographs. During rendering, these photographs are mapped to it. Displaying this mesh from a virtual viewpoint[1] allows to get new views of the scene. Since more than one reference viewpoint can see a face, photographs are blended using weighted average of colors. Weighting function is based on the proximity of reference viewpoints from virtual viewpoint. In their real-time implementation, Debevec *et al.* use a view-map of closest viewing angle to precompute a set of reference viewpoints used to map a given polygon. When a polygon is partially visible from a reference viewpoint, it is subdivided and missing textures are filled with color interpolation. These subdivisions increase the number of polygons in the scene and are prone to numerical imprecision, which can be problematic in some cases (highly complex models and/or meshes composed of small polygons). Moreover the view-map structure does not incorporate well in a real-time rendering application because view-map queries are performed by the CPU. In spite of the fact that queries are simple and fast, they must be evaluated one time per polygon, which can potentially be heavy. Moreover, storing each faces view-map requires a significant amount of system memory in the case of a complex model. From a general point of view, methods based on VDTM [Debevec et al. 1998; Pulli

[1] That is a camera position different from reference viewpoints.

et al. 1997; Pighin et al. 1998; Yamakasi et al. 2002] are fast but deal at best partially with occlusion effects, thus generating blurring, ghosting or popping. Moreover relighting is difficult, even sometimes impossible.

In our case, among a sparse set of viewpoints, we do not use any kind of view-map. We only select three of them to be used for all rendered triangles. Moreover, we do not blend images mapped to the mesh: for a given pixel to be drawn, we simply determine the best viewpoint to use. This approach leads to two main advantages compared to VDTM: firstly, it avoids to subdivide faces, thus simplifying preprocessing stage. Secondly, we do not need to blend multiple reference images to get the pixel color, thus avoiding blurring and skinning artifacts. Furthermore we are able to change on the fly the simplified mesh with another one: this can be useful for LOD-based real-time rendering applications.

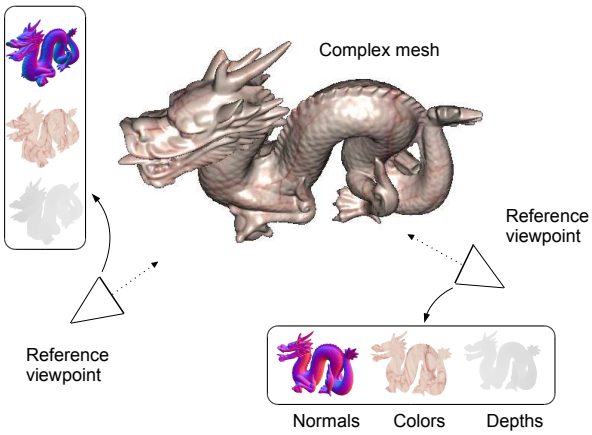


Figure 2: Viewpoint textures

3 Overview of the method

Our method consists in replacing complex 3D objects laid out in a scene with geometrically simplified versions of them over which we apply our rendering algorithm. The aim is to obtain the nearest visual quality possible between the original object and the simplified one.

To obtain simplified meshes, we can use common mesh simplification algorithms such as, for instance [Hoppe 1996; Garland and Heckbert 1997].

The rendering process consists in displaying simplified meshes using our algorithm (see section 5) implemented with recent per-pixel computing capabilities of graphics hardware (see section 6).

During a preprocessing pass, the user must grab some viewpoints for each complex object of the scene. This task is carried out while navigating around the complex model and storing some special kind of snapshot as textures.

4 Reference viewpoints acquisition

Considering an isolated complex mesh, firstly we have to grab some reference viewpoints of it. This task is quickly accomplished by the user, while navigating around the object. The number of reference images depends on the mesh complexity but is not large: for instance we used nine viewpoints for the Stanford dragon mesh and seven for the buddha. The capture process and the nature of the information stored in the reference viewpoints are detailed in the next section. When this operation is finished we obtain a set of viewpoints associated with the object.

A subset composed of three of them is used during rendering stage in order to apply our reconstruction algorithm. The aim of the method, occuring during rendering process, is to compute the final color and depth of each rendered pixel, thus by extracting adequate points stored in the reference viewpoints. This task is completely done by the GPU, using pixel shaders.

A viewpoint is a snapshot of the real mesh for a given camera position. It is a structure composed of the camera transformation matrix, used to transform points from camera space to scene space (see section 5.1), and three kinds of textures: colors texture, depths texture and normals texture [2]. Thus, we have to render three times the complex object for a given camera position. Figure 2 shows two reference viewpoints and obtained images.

Colors texture of the mesh is obtained while drawing it without lighting. Textured models (see middle images on figure 2) can be used since we just need raw surface colors, used as a base for the lighting computations (Phong model).

Depths texture is retreived while reading the depth buffer content (see bottom and right images of figure 2). We used a precision of 16 bits.

Normals texture is obtained while rendering the mesh and substituting vertices color component with normal component (top and left images on figure 2). To store signed vector orientation in unsigned RGB components texture, we simply apply a linear transformation [3]. Later, we recover the signed components with the symetrical operation. We finally obtain a RGB texture representing for each pixel the rendered mesh surface orientation in object local space.

5 Rendering process

Given a simplified mesh and its set of reference viewpoints, we are able to apply directly our method. For each rendered pixel of the simplified mesh, our pixel shader computes its depth and color using a subset of three reference viewpoints. Among the viewpoint set, the three selected viewpoints are the three closest to the actual camera center of projection. These three viewpoints covers the mesh surface from three differents angles[4]. Reconstruction consists in selecting the best point stemming from these reference viewpoints in order to compute lighting and depth of each rasterized pixel.

To find the point stemming from a given viewpoint at 2D screen position (u_p, v_p), we use projective texture mapping as explained below.

5.1 Multiple projective textures mapping

Projective texture mapping is based on *reprojection* which consists in transforming a given pixel lying on the projection plane of a camera into scene space, using a depth map.
Considering an isolated pixel on a camera at coordinates (u_p, v_p), we create the 3D point $P_c = (u_p, v_p, z)$, in camera space coordinates. The z component is the depth of the pixel, directly taken from a depth texture (or z-buffer). To finally get the point in scene space, we simply have to transform P_c using the inverted camera transformation matrix, that is $P_s = M^{-1}.P_c$.

[2] It is similar in nature to the G-Buffer structure used in Deffered Shading [Deering et al. 1988; Harris and Hargeaves 2004].

[3] let $\vec{n}(x,y,z)^\top \; x,y,z \in [0..1]$ be a normal, we store in a given pixel $c(r,g,b)$ the value $((x+1)/2, (y+1)/2, (z+1)/2)$.

[4] As detailed in [Buehler et al. 2001], is not an ideal solution, but this method is fast to compute and gives good results with our algorithm.

In our algorithm, we have to carry out this operation three times, one time per reference viewpoint. As previously said, reprojecting a pixel means one matrix multiplication. To implement efficiently this operation, we used projective texture mapping in order to precompute pixels projected positions onto each reference viewpoint plane. This allows to delocalize this reprojection into vertex shader stage of the graphics pipeline instead of computing this for each drawn pixel. This will be detailed in section 6.

Considering a triangle ABC to be rendered (see figure 3), each of its vertices are projected into each reference viewpoints space.

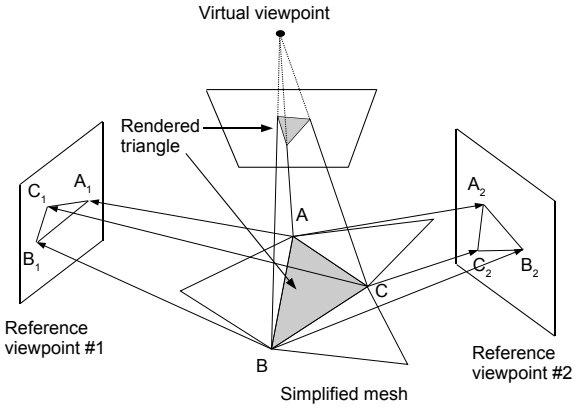


Figure 3: Projective texture coordinate generation. A_i,B_i,C_i are 2D projected texture coordinates of A,B,C triangle vertices, for each viewpoint i.

In the figure we have only considered two reference viewpoints for clarity of explanations: given the point A, we then obtain the 2D points A_1 and A_2. These points are 2D texture coordinates of the vertex, while considering reference viewpoints planes of projections as texture spaces.

5.2 Per-Pixel Visibility

Let us consider P, a rendered point of the simplified mesh (see figure 4).

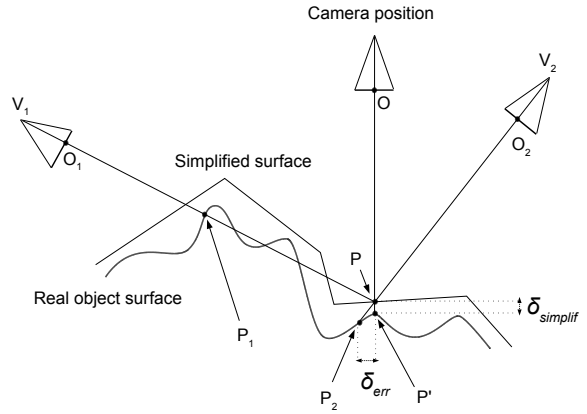


Figure 4: Determination of pixel visibility. For clarity of explanation, we use only two reference viewpoints in this figure.

P is projected onto the observer camera plane, in position O. P gives a rough approximation of the real scene surface point P'. To find the most suitable point stored in the reference viewpoints (V_1 and V_2 in the figure), we get O_1 and O_2, the projective texture coordinates of P for each viewpoint.

Each of these points correspond by retroprojection to a point belonging to the real scene surface (P_1 and P_2). To compute this reprojection, we extract the z component of O_1 and O_2 while simply fetching the depth stored in the viewpoints depth texture.

Among P_1 and P_2, the one which best approximate P' is the one for which its associated scene space point is nearest to P. In this example, viewpoint V_2 is the best: distance between P_2 and P is smaller than distance between P_1 and P.

Finally, we apply this method using three viewpoints instead of two. Our viewpoint selection criterion guarantees that we use the best possible viewpoint among the selected ones. In other words, occlusion effects are taken into account.

Figure 5 shows our visibility algorithm in action. The three selected reference viewpoints are shown in left part of the figure. Right part shows close-ups of our method for four different camera positions. In these close-ups, red pixels means that best reference viewpoint for these pixels is viewpoint #1, and so on for green and blue pixels.

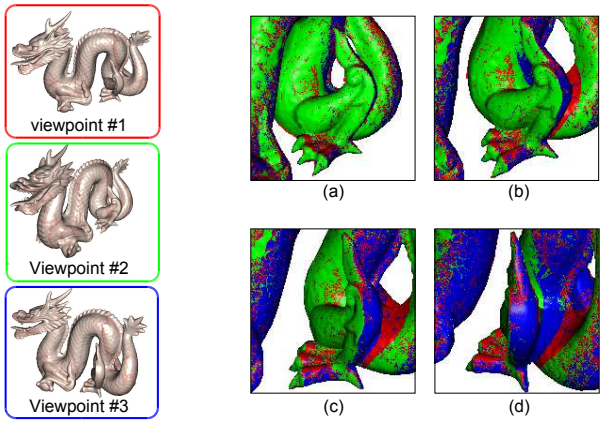


Figure 5: Per-pixel visibility algorithm: selected viewpoints used to compute pixels color.

5.3 Discussion

As shown in figure 4, this method produces an error δ_{err} depending on the proximity of simplified surface from real surface ($\delta_{simplif}$). But in practice this error is hardly noticeable even with heavily simplified objects (we highlight these reconstruction errors in section 7). This is mainly due to the high quality of geometrical simplification algorithm we employ [Hoppe 1996], but also because our selection criterion provides the closer optical ray (in angular term) from the optical ray of the rendered pixel (that is $[OP]$).

For our method, when reference viewpoints entirely covers the object surface, the worst case in term of texture coordinates deviation [5] would be when two of the three viewpoints are occluded because in this case only one viewpoint will provide useful relief informations. Similarly, the best case would be when the three viewpoints "see" the pixel because we will use the one which provides the smallest texture deviation. So, when the user positions the viewpoints, he must take care of theses considerations. This is an important factor for the quality of the reconstruction.

[5] That is δerr projected onto a viewpoint

6 Real-time Implementation

This section deals with technical details about our real-time implementation. We first describe vertex and pixel shader, followed by texture filtering consideration. Then we briefly describe self shadowing effects we used in results section.

One thing to note is that all our computations are driven in object-space. This simply requires to transform viewer camera and light(s) positions into each object space before rendering. We implemented our algorithm using shaders through OpenGL fragment program extensions and NVIDIA CG compiler.

6.1 Shaders

As for all GPU programs, our shaders are splitted into two parts: a vertex program and a pixel (or fragment) program [6]. A shader program is characterized by its constant parameters, its input and output data. Constant parameters means data which will not be modified during object rendering.

Vertex program In our case, *constant parameters* are camera transformation matrix and the three viewpoints transformation matrices.
Input data are vertices data: 3D position and normal, in object space, and texture coordinates.
Output data are values that are interpolated along triangle vertices. Here, they are position in camera space, position in object space and three texture coordinates channels corresponding to projective texture coordinates of the vertex into each reference viewpoints.

Thus, operations carried out in this stage are only vertex projections.

Fragment program *Constant parameters* are data associated to the three selected viewpoints. For each viewpoint, data are: colors texture, normals texture, depth texture and inverted transformation matrix. We have to add eye transformation matrix, its corresponding inverted matrix, light position and object material parameters (such as specular coefficient).
Input data are vertex program output parameters.
Output data are pixel color and its depth.

Operations carried out in this stage are detailed in section 5.2. Main operations are point space transformations: from object space to camera space to screen space. Thus we have to switch between NDC[7] and NWC[8] when fetching texels.

6.2 Texture filtering

Because we use projective texture mapping, we principally have to deal with texture magnification artifacts (see [Everitt 2001]). In this case, graphics cards linear interpolation can reduce artifact but for very close views we do not have much to do: surface looks blurry. This kind of blur is less noticeable than VDTM blur because it comes from linear interpolations among pixels colors from the *same* image, and not from blending of completely different images. Nevertheless, to reduce these artifacts, user must take care of maximizing screen space when grabing viewpoints in order to use at best screen resolution.

6.3 Shadow Projection

We have implemented shadow projection and self-shadowing using Williams method [Williams 1978]. It requires one more projection into fragment shader stage: when pixel extraction is computed, we project the computed point into light camera and compare its depth with the one stored into light depth texture.
Thus, for a given object, we have to render its simplified mesh from light camera and store obtained z-buffer. Render-to-texture graphics cards capabilities makes this task easy and cheap to use. Finally, we have to add depth texture, light transformation matrix and its inverted matrix to constant parameters of the fragment shader. In scene space, in order to maximize z-buffer precision, we simply modify light camera frustum parameters to zoom-in objects.

[6] We provide shaders source code on our website, see Results section.
[7] Normalized Device Coordinate
[8] Normalized Window Coordinates

7 Results

For all results presented in this section, we used a resolution of 512x512 pixels for display as well as for textures. We employ NVIDIA GeForceFx 6800GT to implement our shader algorithm. All our experimentations were driven with a 2GHz AMD Athlon XP PC. The shader programs were generated using NVIDIA Cg compiler.
To compare results of our method with conventional rendering, we employ ARB_VERTEX_BUFFER_OBJECT OpenGL extension to render complex objects because it provides the highest performance available to display geometrical meshes.

Please note that shaders source code and videos showing our method in action are located on our web site: http://msi.unilim.fr/∼porquet

Complete scene Figure 6 and 10 show our method results applied to a complete scene composed of multiples instances (clones) of the same mesh, in order to stress the graphics card.

In figure 6 we render 14 meshes composed of 1.1 M triangles each: we get approximately 2 FPS. Using our method, we obtain 67 FPS while using 14 simplified meshes composed of 10 K triangles each.

In figure 10, we render 140 instances of the Dragon mesh, composed of approximately 871K triangle each, at 0.3 FPS. Using our method and simplified meshes composed of 4000 triangles, we obtain 32 FPS.

As shown, our reconstructed scenes are hardly discernible from original scenes.

Self Shadowing Closer look to the shadow projection. Shadows are deformed in flat triangles since we reconstruct the geometry of the real scene, as shown in right image of figure 7. In this image, we used a simplified mesh composed of 2000 triangles.
Left image show our shadow projection applied to the dragon mesh (10 K triangles).

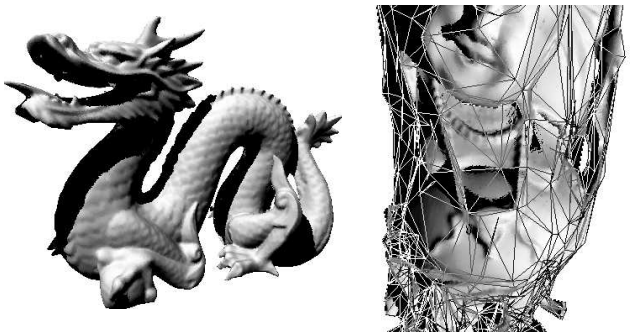
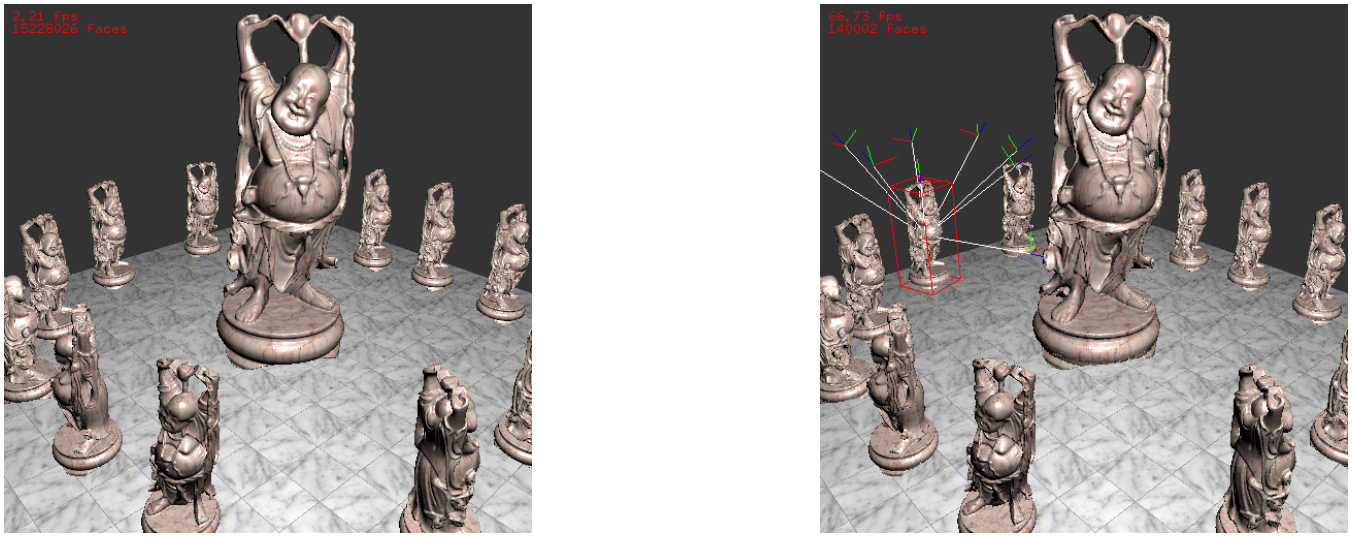


Figure 7: Shadow projection.



(a) Original scene: 15 M triangles, 2.21 FPS

(b) Reconstructed scene: 140 K triangles, 66.73 FPS

Figure 6: Scene composed of multiples clones of the Buddha mesh. The white lines around the mesh in the background show the directions of reference viewpoints.

Rendering quality Figure 9 highlight the visual differences between the original mesh and several simplified meshes. We simply substract colors of the original mesh with colors of simplified meshes: the greatest is the difference, the darker are the pixels. As shown, the main difference between the complex and the simplified mesh images are located onto the meshes borders.

This is emphasized in figure 8, which gives a close view of figure 1. As shown, meshe borders reveals the simplified mesh.

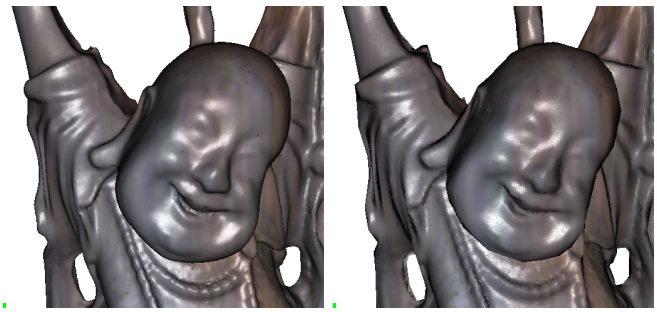
Memory cost While not using any optimizations, memory cost is acceptable: 3 texture per viewpoint use 2 Mb of video memory (using a definition of 512x512 pixels). Nevertheless, it would be straightforward to greatly improve memory cost while using texture compression for depths and colors map, and an indexed normals map. Note that if the object is textured using texture coordinates based on vertices position, we do not need to store reference viewpoints colors map.

8 Conclusion and Future Work

We have presented a simple and efficient system for real-time visualization of complex static geometrical objects. Our work main advantage is to provide such a frame rate almost independently from the geometrical complexity of the scene, yet providing high quality images (e.g. nearly undistinguishable from usual z-buffer rendering).

We extended View-Dependent Texture Mapping (VDTM) while simplifying preprocessing step and improving rendering quality (no blurring effects coming from image blending). Our method is fast and simple to implement using latest graphics cards. It is mainly adapted to interactive rendering applications such as games, because surface reconstruction errors, although small, can be unacceptable for some kind of applications. In games, these visual errors are less noticeable because priority is given to fast interactions instead of visual accuracy.

One objective of our future works is to extend the rendering process in order to obtain detailed borders silhouette even on very simplified meshes. In the same way, we have to improve the algorithm to reduce surface deviation: image warping approach would be a solution. Another objective is to find a method that decrease the memory consumption while capturing automatically the smallest viewpoint set and also while minimizing empty parts in viewpoint textures.



(a) Original object

(b) Reconstructed object

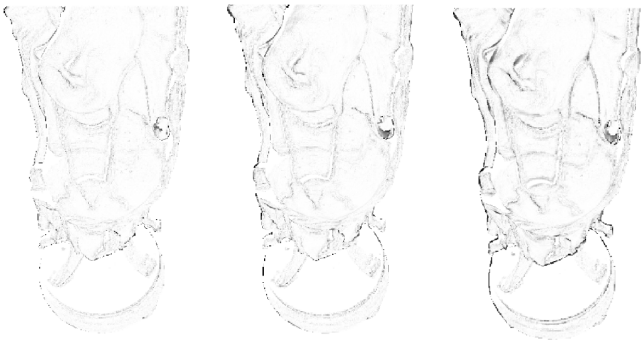
Figure 8: A close view of the reconstructed surface.

References

ALIAGA, D. G., AND LASTRA, A. A. 1998. Smooth transitions in texture-based simplification. *Computers and Graphics 22*, 7 (January), 71–81.

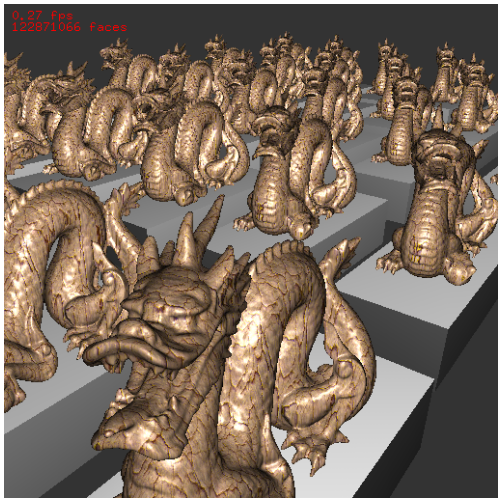
ALIAGA, D. G. 1996. Visualization of complex models using dynamic texture-based simplification. *IEEE Visualization '96*.

BUEHLER, C., BOSSE, M., MCMILLAN, L., GORTLER, S., AND COHEN, M. 2001. Unstructured lumigraph rendering. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, 425–432.

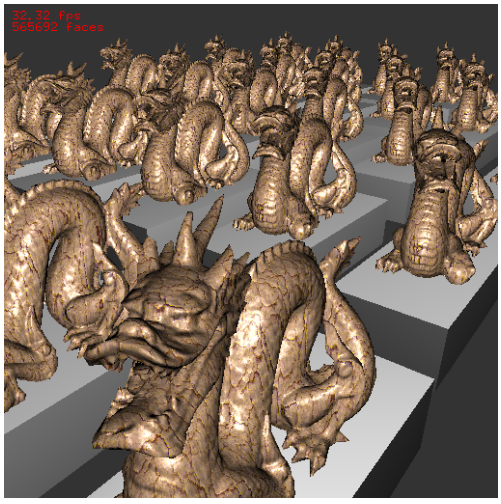


(a) 12 K tri. (b) 6 K tri. (c) 3 K tri.

Figure 9: Visual difference between differents simplified meshes and the original mesh.



(a) Original scene: 122 M tri., 0.3 FPS



(b) Reconstructed scene: 565 K tri., 32 FPS

Figure 10: Complete scene composed of Dragon mesh clones.

CATMULL, E., AND SMITH, A. R. 1980. 3-d transformations of images in scanline order. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, ACM Press, 279–285.

CHANG, C., BISHOP, G., AND LASTRA, A. 1999. Ldi tree : A hierarchical representation for image-based rendering. *Proc. SIGGRAPH 99* (August), 291–298.

CHEN, S., AND WILLIAMS, L. 1993. View interpolation for image synthesis. *Proc. SIGGRAPH 93* (August), 279–288.

CHEN, W.-C., BOUGUET, J.-Y., CHU, M. H., AND GRZESZCZUK, R. 2002. Light field mapping: efficient representation and hardware rendering of surface light fields. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, 447–456.

CHEN, S. 1995. Quicktime vr - an image-based approach to virtual environment naviguation. *Proc. SIGGRAPH 95* (August), 29–38.

COHEN, J., OLANO, M., AND MANOCHA, D. 1998. Appearance-preserving simplification. In *Computer Graphics, Annual Conference Series, ACM SIGGRAPH*, ACM Press, 115–122.

COOK, R. L. 1984. Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM Press, 223–231.

DARSA, L., COSTA SILVA, B., AND VARSHNEY, A. 1997. Navigating static environments using image-space simplification and morphing. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, ACM Press, 25–34.

DEBEVEC, P., TAYLOR, C., AND MALIK, J. 1996. Modeling and rendering architecture from photographs : A hybrid geometry- and image-based approach. *Proc. SIGGRAPH 96* (August), 11–20.

DEBEVEC, P., YU, Y., AND BOSHOKOV, G. 1998. Efficient view-dependent image-based rendering with projective texture-mapping. In *9th Eurographics Rendering Workshop*.

DÉCORET, X., DURAND, F., SILLION, F.-X., AND DORSEY, J. 2003. Billboard clouds for extreme model simplification. *Proc. SIGGRAPH 2003* (August).

DEERING, M., WINNER, S., SCHEDIWY, B., DUFFY, C., AND HUNT, N. 1988. The triangle processor and normal vector shader: A VLSI system for high performance graphics. In *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics (SIGGRAPH 88)*, ACM Press, Atlanta, Georgia, USA, R. J. Beach, Ed., 21–30.

EVERITT, C. 2001. *Projective Texture Mapping*. NVIDIA, http://developer.nvidia.com/. White paper.

GARLAND, M., AND HECKBERT, P. S. 1997. Surface simplification using quadric error metrics. In *Computer Graphics (SIGGRAPH'97 Proceedings)*.

GORTLER, S., GRZESZCZUK, R., SZELISKI, R., AND COHEN, M. F. 1996. The lumigraph. *Proc. SIGGRAPH 96* (August), 43–54.

GUENNEBAUD, G., BARTHE, L., AND PAULIN, M. 2004. Deferred Splatting. In *Computer Graphics Forum*, vol. 23. European Association for Computer Graphics and Blackwell Publishing, PO Box 805, 108 Cowley Road, Oxford, United Kingdom, septembre, 1–11.

HARRIS, M., AND HARGEAVES, S. 2004. *Deferred Shading*. NVIDIA, http://developer.nvidia.com/. White paper.

HECKBERT, P. S., AND GARLAND, M. 1997. Survey of polygonal surface simplification algorithms. Tech. rep.

HIDALGO, E., AND HUBBOLD, R. 2002. Interactive rendering of complex and animated scenes : Hybrid geometric - image based rendering. *Computer Graphics Forum (Eurographics 2002)*.

HIRCHE, J., EHLERT, A., GUTHE, S., AND DOGGETT, M. 2004. Hardware accelerated per-pixel displacement mapping. In *Proceedings of the 2004 conference on Graphics interface*, Canadian Human-Computer Communications Society, 153–158.

HOPPE, H. 1996. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM Press, 99–108.

HORRY, Y., ANJYO, K., AND ARAI, K. 1997. Tour into the picture: using a spidery mesh interface to make animation from single image. *Proc. SIGGRAPH 97*, 225–232.

JESCHKE, S., AND WIMMER, M. 2002. Textured depth meshes for real-time rendering of arbitrary scenes. *Eurographics Workshop on Rendering*.

LEVOY, M., AND HANRAHAN, P. 1996. Light field rendering. *Proc. SIGGRAPH 96*, 33–42.

LUEBKE, D., AND ERIKSON, C. 1997. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 199–208.

MARK, W. R., MCMILLAN, L., AND BISHOP, G. 1997. Post-rendering 3d warping. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, ACM Press, 7–16.

MCMILLAN, L., AND BISHOP, G. August 1995. Plenoptic modeling: An image-based rendering system. *Proceedings of SIGGRAPH 95*, 39–46. ISBN 0-201-84776-0. Held in Los Angeles, California.

MCMILLAN, L. 1995. Computing Visibility Without Depth. Tech. Rep. TR95-047, Department of Computer Science, University of North Carolina - Chapel Hill, Oct.

MCMILLAN, L. 1999. An image based approach to three-dimensional computer graphics. *Technical report, Ph.D. Dissertation, UNC Computer Science TR97-013*.

MOULE, K., AND MCCOOL, M. 2002. Efficient bounded adaptive tessellation of displacement maps. In *Proceedings of the Graphics Interface 2002 (GI-02)*, Canadian Information Processing Society, Calgary, Alberta, 171–180.

OLIVEIRA, M., G, G. B., AND MCALLISTER, D. 2000. Relief texture mapping. *Proc. SIGGRAPH 2000*, 359–368.

PAULO, W., MACIEL, C., AND SHIRLEY, P. 1995. Visual navigation of large environments using textured clusters. *Symposium on Interactive 3D Graphics.*, 95–102.

PFISTER, H., ZWICKER, M., VAN BAAR, J., AND GROSS, M. 2000. Surfels: elements as rendering primitives. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 335–342.

PIGHIN, F., HECKER, J., LISCHINSKI, D., SZELISKI, R., AND SALESIN, D. H. 1998. Synthesizing realistic facial expressions from photographs. In *Proceedings of the ACM Conference on Computer Graphics (SIGGRAPH-98)*, ACM Press, Orlando, FL, USA, 75–84. ISBN 0-89791-999-8.

PULLI, K., COHEN, M., DUCHAMP, T., HOPPE, H., SHAPIRO, L., AND STUETZLE, W. 1997. View-based rendering: Visualizing real objects from scanned range and color data. In *Eurographics Rendering Workshop 1997*, Springer Wein, New York City, NY, J. Dorsey and P. Slusallek, Eds., Eurographics, 23–34. ISBN 3-211-83001-4.

RUSINKIEWICZ, S., AND LEVOY, M. 2000. Qsplat: a multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 343–352.

SANDER, P. V., SNYDER, J., GORTLER, S. J., AND HOPPE, H. 2001. Texture mapping progressive meshes. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, 409–416.

SEGAL, M., KOROBKIN, C., VAN WIDENFELT, R., FORAN, J., AND HAEBERLI, P. 1992. Fast shadows and lighting effects using texture mapping. *SIGGRAPH Comput. Graph. 26*, 2, 249–252.

SEITZ, S. M., AND DYER, C. R. 1996. View morphing. In *Proceedings of the ACM Conference on Computer Graphics*, ACM, New Orleans, LA, USA, 21–30. ISBN 0-201-94800-1.

SHADE, J., GORTLER, S., WEI HE, L., AND SZELISKI, R. 1998. Layered depth images. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM Press, 231–242.

WANG, L., WANG, X., TONG, X., LIN, S., HU, S., GUO, B., AND SHUM, H. 2003. View-dependent displacement mapping. *Proc. SIGGRAPH 2003*.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, ACM Press, 270–274.

WIMMER, M., WONKA, P., AND SILLION, F. 2001. Point-based impostors for real-time visualization. *Eurographics Workshop on Rendering*.

WONKA, P., AND SCHMALSTIEG, D. 1999. Occluder shadows for fast walkthroughs of urban environments. *Computer Graphics Forum (Eurographics '99)*, 51–60.

YAMAKASI, S., SAGAWA, R., KAWASAKI, H., IKEUCHI, K., AND SAKAUCHI, M. 2002. Microfacet billboarding. *Proc. of the 13th Eurographics Workshop on Rendering.*, 175–186.

ZHANG, H., MANOCHA, D., HUDSON, T., AND HOFF, K. 1997. Visibility culling using hierarchical occlusion maps. *Proc. SIGGRAPH 97*, 77–88.