

An anti-aliasing technique for voxel-based massive model visualization strategies

Gustavo N. Wagner, Alberto Raposo, Marcelo Gattass

Tecgraf, Computer Science Dept., PUC-Rio, Rua Marques de São
Vicente, 255, 22453-900 – Rio de Janeiro, Brazil
{gustavow, abraposo, mgattass}@tecgraf.puc-rio.br
<http://www.tecgraf.puc-rio.br>

Abstract. CAD models of industrial installations usually have hundreds of millions of triangles. For this reason they cannot be interactively rendered in the current generation of computer hardware. There are many different approaches to deal with this problem, including the Far Voxels algorithm, which uses a hierarchical level-of-detail structure. In this structure, voxels are used to create a coarse representation of the model when required. This strategy yields interactive rates for large data sets because it deals well with levels of detail, culling, occlusion and out-of-core model storage. The Far Voxels algorithm, however, has a severe alias problem when it is used to represent small or thin objects, which is especially visible during transitions between different levels of detail. This paper presents a new version of the Far Voxels algorithm that improves visual quality during model navigation.

1 Introduction

The visualization of CAD (Computer Aided Design) models is important for many engineering-related activities spanning from conception and design up to maintenance. There are, however, several problems in obtaining quality visualization of CAD models. One of these problems is the size of the visualization model, which can easily achieve hundreds of millions of triangles. As graphics engines become powerful enough to deal with large models, research on massive model visualization has received greater attention. Despite all recent advancements, most real CAD models still exceed the processing and memory capacity of existing hardware when interactive rates are required.

To speed up the rendering of this type of model, several algorithms have been proposed in the literature. Most of them are based on a combination of strategies that can be summarized as a combination of hierarchical levels of detail (HLOD), culling, occlusion, efficient use of the graphics pipeline and good management of disk, CPU and GPU memory. A promising algorithm to render massive models, called the Far Voxels, was presented in the paper by Gobbetti and Marton [3]. This algorithm uses a HLOD structure where intermediate (coarse) representations of sub-models are represented by voxels. This HLOD with voxel representation yields interactive rates for large data sets because it deals well with levels of detail, culling, occlusion and out-of-core model storage.

The Far Voxels algorithm, however, has a severe drawback when dealing with

CAD models. CAD models usually have lines and thin objects that, even with the most detailed representation, introduce very high spatial frequency that cannot be rendered without proper anti-aliasing treatment. The aliasing caused is very disturbing during model navigation, where these thin objects seem to move between consecutive frames.

In the original Far Voxels algorithm the temporal aliasing problem is aggravated when these thin objects are converted to voxels. The voxels used to represent these objects tend to create representations which are larger than the ones of the original geometric representation, as shown in Fig. 1. In addition to creating a representation that is very different from the original model, this distortion causes very noticeable popping artifacts in the transition between different levels of detail.

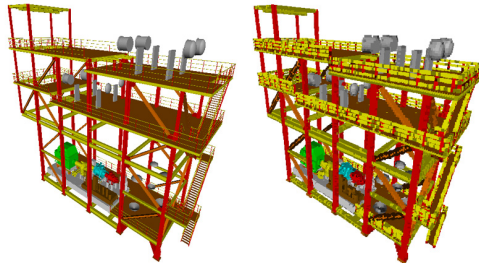


Fig. 1. Thin objects: (*left*) represented as geometry, (*right*) represented as voxels, causing visual artifacts.

In this paper we propose a method for detecting this type of voxel and an alternative voxel representation that seeks to achieve images of the same quality obtained with current 3D hardware's anti-aliasing using the detailed representation of the model (triangles and lines). This detection and alternate voxel representation yields a new version of the Far Voxels algorithm.

2 Related work

The literature on HLOD and point rendering is extensive and a complete review would be too long to be included here. For this reason, only key papers related to point rendering or improvement of its visual quality are discussed. Complete surveys on massive model rendering can be found in [3] and [10].

The idea of using points as graphics primitives is not new. In 1985, Levoy and Whitted [6] presented a seminal paper that proposes points as an efficient display primitive to render complex objects. In 1998, more than a decade later, Grossman and Dally [4] presented an efficient algorithm to render object from sampled points in the CPU. An advantage of representations based on points is that they do not require any effort to preserve the model's topology, as it would have to be done in the frontier between meshes with different resolutions.

More recent approaches began to make use of 3D graphics hardware. In 2000, Rusinkiewicz and Levoy [9] presented the QSplat system, which uses point primitives to render very complex laser scanned models. In this system, the model is converted

to a hierarchy of bounding-spheres. During visualization, the most appropriate spheres from the hierarchy are selected and rendered in 3D hardware as point primitives. In this hierarchy, points of a region of the model are grouped into a single point, which may be used instead of the cluster when viewed at distance.

In 2004 Gobbetti and Marton [2] presented an approach to organize point primitives in clusters. Their strategy reduces CPU processing, permits the clouds to be cached in graphics hardware and improves the performance of CPU-to-GPU communications. This idea evolved into the Far Voxels technique [3], which allows much larger models to be efficiently rendered with point-based primitives. The leaf nodes, which contain the most detailed representation of the model in the hierarchy, are still rendered as triangles.

Much effort has been placed in trying to improve the rendering quality of point primitives. In QSplat [9], different primitive shapes are tested to determine the one with the best quality/performance ratio. Alexa et al. [1], convert the model into a tree of higher order polynomial patches which is used to generate the point primitives in the proper resolution needed during visualization.

Zwicker et al. [11] presented a seminal paper about sampling issues in point rendering, based on the concept of Elliptical Weighted Average (EWA) resampling filters, which were introduced by Heckbert [5]. Ren, Pfister and Zwicker [8] proposed an implementation compatible with modern graphics hardware for the EWA screen filter [11], which permits high quality rendering of point-based 3D objects.

Here we approach the problem of visualizing large models using the Far Voxels algorithm, which is one of the most efficient techniques available for dealing with large CAD models. We present a solution that improves the appearance of a few specific voxels of the model, namely those representing objects whose dimensions are smaller than one voxel. Our approach to improve the rendering quality of point primitives is orthogonal to the abovementioned ones and could be used in conjunction with any one of them that uses voxels as impostors.

3 Opaque voxels

The Far Voxels algorithm is composed of two main phases: preprocessing and model visualization. The preprocessing phase computes visibility information that will aid in the creation of the simplified voxel-based representations of parts of the model. The visualization phase uses this information to efficiently navigate through the model.

3.1 Preprocessing phase

The preprocessing phase creates a representation of the model optimized for rendering by sampling the volume from all possible directions. This optimized representation is structured on an HLOD hierarchy, which has on its leaves the most detailed representation available for the model, and on the other nodes simplified representations of their respective regions. In this representation, nodes located near the hierarchy root will have the coarsest representations of the model, and are intended to be used to represent the model when it is viewed from far away.

The hierarchy is created by a recursive subdivision which uses axis-aligned planes

positioned according to the surface-area heuristic [7]. The subdivision starts working with all triangles of the model, which are assigned to the hierarchy's root, and then starts subdividing them. Each subdivision will create a new pair of nodes on the tree, with existing triangles split between both nodes. The subdivision process ends when a node with less than a predefined amount of triangles is created. This node becomes a leaf of the hierarchy.

Simplified representations of parts of the model are created using a voxel representation. Inside a node, voxels are organized on a uniform grid, where each voxel will have, under ideal conditions, the size of one pixel when projected on the screen. Each voxel may assume different representations depending on the direction from where it is visualized, as it attempts to reproduce the appearance of the original model as closely as possible.

The simplified representations of the model's regions are created in two steps. First, all visible surfaces in that region are sampled with the use of a CPU-based raytracer. Then, the radiance of all the hits obtained from the intersection of the rays with the region's surfaces is analyzed and the appropriate shader to represent them is selected. The use of a raytracer permits selecting only the surfaces of the model that are visible, which is important to avoid that hidden surfaces create artifacts on the generated simplified representation.

The raytracer starts by defining a volume V , which corresponds to the region being sampled, and a surface S , from where the sampling rays will be shot. The distance d_{\min} of surface S to the volume V is calculated as being equal to the minimum distance that the user has to be from V for this voxel representation to be used. As we can assure that the user will not be inside the region defined by surface S , all objects inside it can be used as occluders (Fig. 2a).

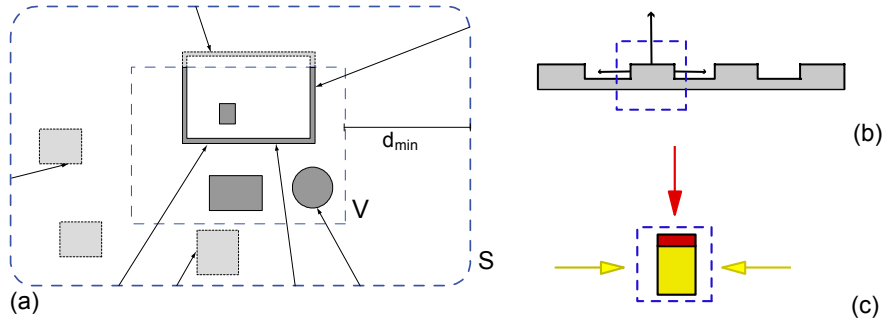


Fig. 2. (a) Example of a possible raytracing configuration. Voxel appearances may vary with viewing direction: (b) varying normals, (c) varying colors.

Many of the surfaces located inside features of the model are never hit by the sampling rays traced during the preprocessing phase and do not need to be represented, as it is assumed that this voxel representation is always viewed from outside. This is called Environmental Occlusion [3] and allows the creation of a more efficient simplified representation of the model. The amount of discarded voxels is reported to vary from 25% to 43% in the original paper and reached up to 57% in the models used here to test our implementation.

For all voxels that are hit by the sampling rays, the colors computed for all directions are stored and then analyzed to produce a radiance model for that point in space. This analysis results in a choice of the most appropriate shader model to render the voxel from the minimum distance established by the criterion that it should project in approximately one pixel. There are two types of shade models. The first one is intended to represent surfaces which are flat or almost flat, and is parameterized by a single normal and two materials, one for each direction from which the voxel may be seen. The second one is intended for more complex surfaces and is composed of 6 materials and 6 normals, each associated with one of the main viewing directions ($\pm x$, $\pm y$, $\pm z$). In Fig. 2b and 2c, there are examples where the normal and color of the surfaces are significantly different depending on the direction from where they are seen, and a single representation for the normal or color on the voxel would not generate satisfactory results. When the corresponding voxel is being viewed from an intermediate direction, the resulting material and normal used correspond to an interpolation of the material and normal associated with the nearest viewing directions weighted by their respective direction cosines.

The generated voxels are stored on disk to be loaded on demand during the model's visualization. The data file that is generated is divided in two parts. One part corresponds to the HLOD hierarchy, which contains all information necessary to traverse it and determine the appropriate configuration of nodes to be used. The other part contains the data that is used to represent the model, composed of lists of voxels for internal nodes of the hierarchy, and lists of triangles/triangle-strips for leaf nodes.

3.2 Visualization phase

During visualization, all nodes of the model's hierarchy are traversed. Each voxel node found on the hierarchy may be composed of voxels of the two existing types. For each type, the appropriate vertex shader is bound and all the voxels contained in it are drawn with a single call to `glDrawArrays`. The additional parameters necessary to create the voxel representation are transmitted using `VertexAttributes`.

Each node of the model's hierarchy contains a possible simplified representation for its corresponding region of the model. While traversing these nodes, the viewer has to decide whether to use the representation available at that node or to continue traversing the node's children in search of a more precise representation. This decision is based on a user-defined parameter that indicates the size that the voxel of a node must have when it is projected to the screen. The viewer keeps descending on the hierarchical structure until the existing voxels are smaller than the desired projected size or until a leaf node is found.

During the model traversal, branches of the hierarchy are tested for occlusion using Hardware Occlusion Queries. Parts of the model that are determined as being hidden do not need to be drawn.

The rendering algorithm may issue the Occlusion Query to determine if a certain branch of the graph is visible while rendering its bounding box or while rendering its full geometry. If only its bounding box is rendered and is found to be visible, its geometry will have to be rendered later. On the other hand, if the branch's full geometry is rendered, there is no advantage, at least for the current frame, in

determining that it is hidden.

The visibility information obtained is used on the next frame when the renderer needs to decide how to query a node's visibility. Nodes that were hidden on the previous frame have their bounding-boxes checked for visibility before being rendered. These nodes have a good chance of remaining hidden in the current frame. The nodes that were visible on the last frame are always directly rendered as geometry or voxels.

Nodes on the scene hierarchy are visited by order of proximity to the camera, starting from the nearest ones. The results for the issued Occlusion Queries are not available instantly, so the renderer must not stop rendering while it waits for them. All performed queries are stored in a list of pending queries, which is checked by the visualization algorithm every time it needs to select a new node to be rendered. If the result of any pending query becomes available and indicates that its associated geometry is visible and has to be rendered, it is rendered immediately.

4 Detecting problematic voxels

When voxels are used to represent parts of the model composed of small or thin objects, the "pure" Far Voxels algorithm tends to create representations which appear "bloated" when compared to the original geometric representation. Unfortunately, thin and small objects are very common in CAD models and the first tests conducted with the original Far Voxels algorithm yielded unsatisfactory results. In addition to creating distortions in the simplified model appearance, they cause a very noticeable popping effect in the transition between two different levels of detail.

A naive solution would be breaking the voxel into smaller voxels, which would create a representation that could adjust to the model with the required precision. As in all other aliasing problems, increasing the resolution is not the best method to solve the problems arising from the presence of high spatial frequencies. Furthermore, this increase contradicts the whole idea of having voxels with the approximate size of one pixel in the screen.

A better approach results from using the pre-processing stage as a sampling stage. Voxels with small objects are problematic because they enclose high spatial frequencies that must be filtered before the rendering reconstruction occurs.

A possible implementation for filtering and rendering voxels with small objects follows the same ideas for anti-aliasing currently implemented in graphic boards. In the sampling stage they determine an opacity factor to be attributed to the voxel. The blending of a partially transparent voxel yields the blurring that results from filtering high frequencies. Note that to achieve this result it is not just a matter of enabling the 3D hardware's anti-aliasing procedures with opaque voxels. Semi-transparent voxels must be used.

In order to determine which voxels have to be represented as semi-transparent voxels, we use the visibility information generated by the raytracer during the preprocessing phase (Fig. 3a). Depending on the geometry configuration inside a voxel, the rays shot against it may hit some geometry inside it or pass through it. When a voxel is trespassed by many rays in a certain direction, we may safely assume that the geometry contained in that voxel occupies a small cross-section area of the

voxel when it is viewed from that angle. Thus, for that angle, this voxel may be rendered with a transparency scale proportional to the ratio between the amount of rays that hit a surface inside the voxel and the total number of rays that were shot against it.

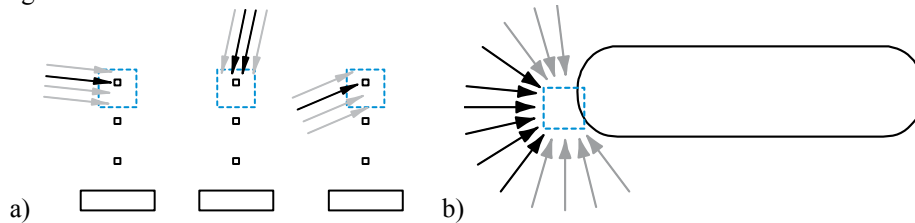


Fig. 3. a) Raytracer sampling a voxel to obtain its visibility. b) Voxels containing small parts of large objects may also have many rays passing through it.

We must be careful to use the transparency information in a different direction than the one in which it is computed. In very frequent cases, parts of large objects may occupy only a small fraction of the voxel (Fig. 3b) in a way it is hit by rays that come only from a few directions. This voxel may be rendered with transparency when viewed from the direction from where the rays that passed through it came from, but must remain opaque when viewed from all other directions. This procedure prevents the appearance of holes in large objects.

As storing visibility information for every possible viewing direction is unfeasible, the transparency information obtained from each sampling direction must be packed before it may be used in the visualization. Since the rays that crossed a certain voxel are not evenly distributed over all directions, we use a representation that allows independent transparencies for each of the six main viewing directions ($\pm x$, $\pm y$, $\pm z$). That is, we use a transparency model that is similar to the material color model explained in section 3.

The final voxel transparency will be an interpolation of the transparencies associated with the nearest viewing directions weighted by their respective direction cosines. If the transparency factor is the same for all directions, we may use a single transparency factor for the voxel, yielding a more efficient representation.

5 Rendering anti-aliased voxels

Correctly rendering any kind of transparent geometry using 3D hardware requires that it is drawn after the whole opaque geometry in the rendered scene. Also, transparent geometry has to be drawn from back to front in order to create a correct representation in cases where two or more transparent objects overlap on screen. This is exactly the opposite behavior expected for geometry being tested for occlusion with Occlusion Queries, which has to be rendered from front to back. For this reason, transparent voxels have to be handled in a different way.

As transparent voxels make up for only a small part of all voxels they may be rendered without being tested for occlusion after the whole opaque geometry, without significant impact on the overall performance. Also, most of the occluded transparent

voxels are going to be discarded before being rasterized by the Early Z Culling present in current graphics cards, as most occluders will have already been rendered.

Ordering all voxels from back to front could cause performance problems with larger models, as we would have to manage these objects on a per-primitive level. As transparent voxels are well distributed in small groups throughout the model, a more relaxed approach can be used. During rendering, only the nodes of the graph that contains transparent voxels are ordered from back to front, while letting the primitives that exist inside these nodes be rendered in any order. As transparent voxels occur only in well distributed parts of the models, visual artifacts that arise from this simplification are hardly noticed, and can be further reduced if the transparent voxels are rendered with Z-Buffer writing disabled.

6 Results

The navigation application used to validate and test the developed technique was implemented in C++ using OpenGL. The models used to test the algorithm were real offshore structure models provided by Petrobras, a Brazilian Oil & Gas Company. In this paper, we present the results obtained using a single real engineering model, the P-50 FPSO (Floating Production Storage and Offloading), with 30 million triangles and 1.2 million objects. This model was chosen because it contains many regions where small or thin objects create visual artifacts when represented using voxels.

Performance tests were made over a typical walkthrough of the model. The machine used was an Athlon X2 64 4200+, with 4 GB of RAM memory, nVidia GeForce 8800 GTX graphics card with 768 MB of memory running on Windows XP Professional 32-bit.

The model was preprocessed on the same machine, but using Windows XP Professional 64-bit to allow the preprocessor to have access to all the available system memory. The entire model was preprocessed in 27 hours on a single machine.

Transparency information, which was added to all voxels of the model, only increased the total processed model size in 5%, from 1.56 GB to 1.64 GB. The rendering performance of the developed method was evaluated during a walkthrough that followed the path outlined in Fig. 4.

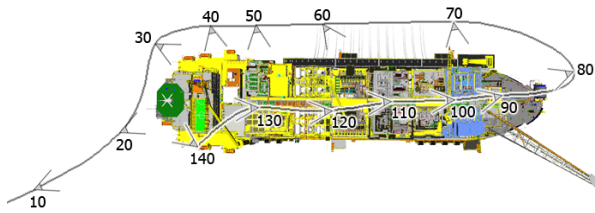


Fig. 4. Path followed on the walkthrough around the P-50 model.

Frame rate information was recorded for each frame rendered. The data recorded compares the performance of the walkthrough over a model that uses the anti-aliased voxels developed with the performance obtained on a model that uses only the voxels presented in the original Far Voxels technique and occlusion queries. The frame rate

obtained during the walkthrough over these two models is plotted in Fig. 5. Comparing the results obtained with these two different types of model allows verifying the efficiency of the developed anti-aliasing method.

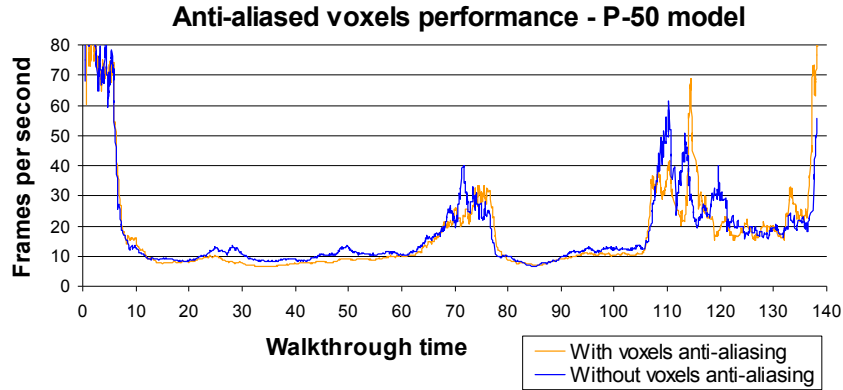


Fig. 5. Walkthrough performance with and without anti-aliased voxels.

The graph shows that the performance does not decrease significantly when the new types of voxels are used. This indicates that the developed voxel anti-aliasing technique is efficient enough to be applied on larger models. Regarding visual quality improvement, Fig. 6 illustrates the difference between the voxels with and without anti-aliasing, comparing them to the ideal representation which uses only triangles.

7 Conclusions

This paper evaluated the use of the Far Voxels technique on models of offshore structures. Due to particular characteristics of these models, their visualization presented problems that were not treated in the original algorithm. To improve the visual representation of very small and thin objects, a detection method and an alternative voxel representation were implemented. This technique was implemented with a small simplification in the way transparent voxels are ordered, aiming at not increasing the CPU processing required to prepare the scene for rendering.

The results were evaluated using real oil & gas platform models, and the algorithm performance with the anti-aliasing filter was not significantly inferior to the performance obtained with our implementation of the original Far Voxels algorithm, which by itself is far more efficient than using simpler optimizations. This is especially true for very large models.

References

1. Alexa, M., Behr, J., Cohen-or, D., Fleishman, S., Levin, D., Silva, C. T.: Point set surfaces. In: Proceedings of the conference on Visualization '01, IEEE Computer Society (2001) 21-28.
2. Gobbetti, E., Marton, F.: Layered point clouds: a simple and efficient multiresolution

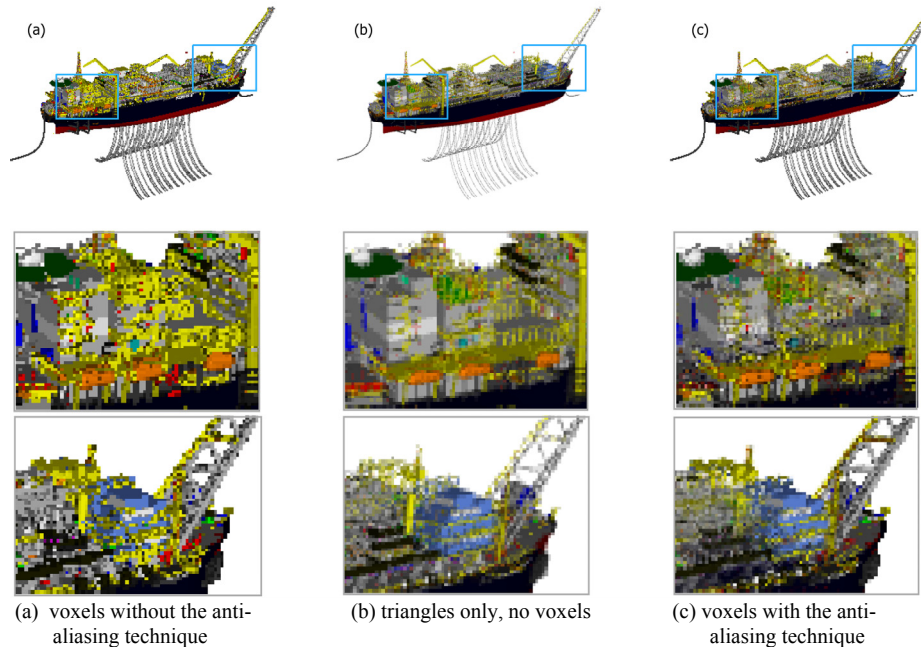


Figure 6. Results obtained using the developed technique compared with the ones obtained the original voxel strategy. The zoomed views are intended to be pixelated.

- structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics* 28, 6 (2004) 815-826.
3. Gobbetti, E. Marton, F.: Far voxels: a multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms. *ACM Trans. Graph.* 24, 3 (2005), 878-885.
 4. Grossman, J.P., Dally, W.J.: Point Sample Rendering, In: 9th Eurographics Workshop on Rendering, (1998), 181-192.
 5. Heckbert, P.: Fundamentals of texture mapping and image warping. M.sc. thesis, University of California, Berkeley, (1989).
 6. Levoy, M., Whitted, T.: The use of points as a display primitive, Tech. Rep. TR 85-022, University of North Carolina at Chapel Hill (1985).
 7. MacDonald, J. D., Booth, K. S.: Heuristics for ray tracing using space subdivision. *The Visual Computer* 6, 6, (1990) 153-165.
 8. Ren, L., Pfister, H., Zwicker, M.: Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering, *Computer Graphics Forum* 21, 3 (2002) 461-470.
 9. Rusinkiewicz, S. Levoy, M.: 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., (2000) 343-352.
 10. Yoon, S., Salomon, B., Gayle, R., and Manocha, D.: Quick-VDR: Interactive View-Dependent Rendering of Massive Models. In: *Proceedings of the IEEE Conference on Visualization '04* (2004). IEEE Computer Society, 131-138.
 11. Zwicker, M., Pfister, H., Van Baar, J., and Gross, M.: Surface Splatting. In: *Computer Graphics, SIGGRAPH 2001 Proceedings*, (2001) 371-378.