Computer-Aided Design 79 (2016) 48–59

Contents lists available at ScienceDirect

Computer-Aided Design

journal homepage: www.elsevier.com/locate/cad

Interactive inspection of complex multi-object industrial assemblies

O. Argudo, I. Besora, P. Brunet *, C. Creus, P. Hermosilla, I. Navazo, À. Vinacua

ViRVIG Research Group, Technical University of Catalonia, Spain[1]

ARTICLE INFO

Article history:
Received 10 February 2015
Accepted 22 June 2016

Keywords:
Computer graphics
Methodology and techniques

渲染评估，对象选取、替换、实时冲突检测

视图相关可视化

ABSTRACT

The use of virtual prototypes and digital models containing thousands of individual objects is commonplace in complex industrial applications like the cooperative design of huge ships. Designers are interested in selecting and editing specific sets of objects during the interactive inspection sessions. This is however not supported by standard visualization systems for huge models. In this paper we discuss in detail the concept of rendering front in multiresolution trees, their properties and the algorithms that construct the hierarchy and efficiently render it, applied to very complex CAD models, so that the model structure and the identities of objects are preserved. We also propose an algorithm for the interactive inspection of huge models which uses a rendering budget and supports selection of individual objects and sets of objects, displacement of the selected objects and real-time collision detection during these displacements. Our solution – based on the analysis of several existing view-dependent visualization schemes – uses a Hybrid Multiresolution Tree that mixes layers of exact geometry, simplified models and impostors, together with a time-critical, view-dependent algorithm and a Constrained Front. The algorithm has been successfully tested in real industrial environments; the models involved are presented and discussed in the paper.

图形质量和帧率

对象选取，对象的层次信息，允许实时注释和修改

场景编辑，实时冲突检测

1. Introduction

A number of algorithms for real-time visualization of huge digital 3D models have been proposed. While they are well suited for many applications, they do not meet the present user requirements in some industrial applications. Complex virtual prototypes are essential in many industrial endeavors involving large models, like in the automotive, aeronautic and ship-building industries. Moreover, the high cost of many of these designs, sometimes destined to be built only once, makes the use of physical prototypes unfeasible. Also, these models contain thousands of individual objects. Instead of relying only on standard visualization systems, designers are interested in addressing individual objects and specific sets of objects. We have identified these requirements for inspection applications in industrial design of complex multi-object assemblies:

- The View-Dependent Visualization algorithm should guarantee a certain frame-rate with good image quality.
- The System must support the selection of individual objects and of hierarchies of objects, during the navigation, to access information of these objects or to annotate and modify them.
- Limited scene editing (including displacement of the selected objects) and real-time collision detection during scene editing must be supported.

As far as we know, no present algorithm fulfills all of the above requirements. Furthermore, previous solutions often resort to substitutions, modifications or simplifications of the geometry that blur the scene structure and the individual objects. Instead, we are interested in preserving the design intent, and the structure of the design tree of the original CAD model, that is meaningful to the users. We show that we can achieve these goals while maintaining sustained frame rates for very large models. Fig. 1 shows the edition and annotation of such a model. These editions and annotations are saved for later incorporation into the CAD model if appropriate. Building on previous contributions in the literature, we propose a solution that preserves CAD hierarchies and object identities while allowing simple interactions. The main contributions of this paper are:

- A formal discussion of the front concept in multiresolution trees, and the characterization of the properties required for time-critical rendering.
- An object-aware scene simplification and multiresolution scheme that results in a multilayered, multiresolution tree

* Corresponding author.
E-mail addresses: oscar.argudo.medrano@gmail.com (O. Argudo), onesvenus@gmail.com (I. Besora), pere@lsi.upc.edu (P. Brunet), crazycraft@gmail.com (C. Creus), pit2500@gmail.com (P. Hermosilla), isabel@lsi.upc.edu (I. Navazo), alvar@lsi.upc.edu (À. Vinacua).
[1] http://virvig.eu.

http://dx.doi.org/10.1016/j.cad.2016.06.005
0010-4485/

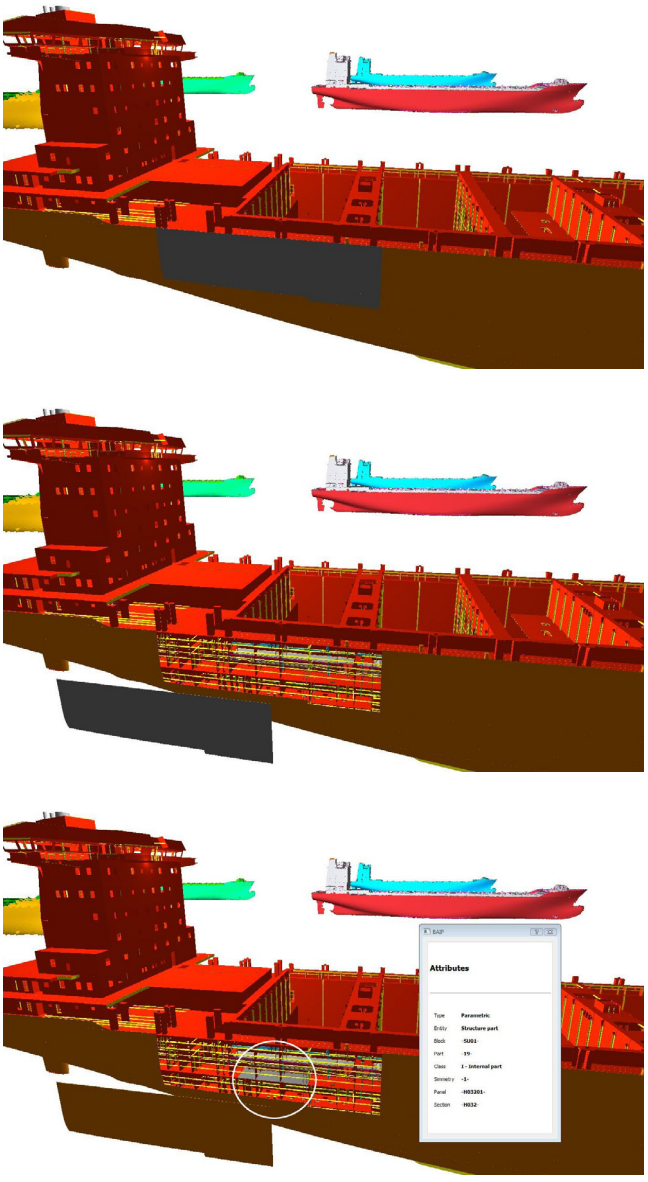

Fig. 1. The model used in the tests. The top image shows a portion of the hull selected by the user in gray. In the middle image the user has moved the selected portion of the hull, partially revealing the complexity of the model inside. The bottom image shows data associated to an object, that the user may annotate. Notice the ships in the background, which are rendered using omnidirectional relief impostors.

with cost and benefit functions per node, which is monotonic by construction. Layers mix geometry and impostors. The layered scheme is based on the results of an evaluation of user perception of several hierarchical representations.

- Support for the selection of individual objects and sets of objects, displacement of the selected objects and real-time collision detection during these displacements while in the interactive navigation, thanks to the object-aware nature of the representation.
- A time-critical, view-dependent visualization algorithm with constrained front update based on a greedy optimization per frame, usable in commodity hardware.

The rest of the paper is organized as follows. Section 2 reviews prior work. After an analysis of front-based rendering algorithms in Section 3, an overview of the algorithm is presented in Section 4, whereas Sections 5 and 6 detail the scene tree generation algorithm and its visualization. Section 7 discusses several results on an example scene, and Section 8 presents the main conclusions and outlines future research directions.

2. Previous work

In this section we discuss only a selection of the algorithms most relevant to our work. For an extensive survey, the reader can refer to [1] or [2]. These algorithms are usually based on the generation, in a preprocess step, of a scene data structure which represents the scene model at different levels of detail. During the interactive visualization, a suboptimal set of nodes is computed and rendered at each frame.

Multiresolution Trees are well-known data structures that represent scenes and assemblies at multiple resolutions. Multiresolution Trees can be binary [3], quaternary [4] or octal [5], and based on either a spatial subdivision [6] or a scene subdivision [4]. Binary subdivision structures like Kd-trees have been widely used because of their splitting flexibility [7].

Multiresolution Trees creation algorithms work in two steps. In the first step, the Tree structure is generated in a top-down way by recursively distributing the scene geometry from parent to son nodes. At the end of this first step, the algorithm has distributed the scene geometry among all tree leaf nodes, and leaf nodes have a size not exceeding a predefined value. The second step operates bottom-up by merging node information and simplifying the union of the informations in their son nodes in a way such that all internal nodes have a size again within the chosen limit.

The paper from Funkhouser and Sequin [8] was seminal in this area. Their scene structure consisted of a simple list (or array) of objects, which sufficed for scenes of only moderate complexity. The preprocess consisted in the computation of a 2D array of object representations, storing each of the N objects at M different levels of detail. Standard simplification algorithms were used for this purpose. In the kernel of the visualization scheme, cost and benefit functions were defined and computed for each object and for each of its M LODs. The cost (time to render the atomic object) was considered to be constant, and computed in the preprocess step. Benefit, however, was dynamic, depending on the camera position and on how the object was projected on the viewport. Funkhouser and Sequin further defined a constrained optimization problem per frame: the goal was to maximize the total benefit per frame, with the constraint that the total cost of the rendered primitives did not exceed the rendering budget. For this purpose, they proposed a greedy front update algorithm, applied at each frame. The object with the maximum benefit to cost ratio was refined, while one or more objects having the lowest ratio were coarsened to keep the total cost below the budget. Based on this work, Gobbetti and Bouvier [9] proposed a solution for this optimization using Lagrange multipliers.

The term *view-dependent visualization algorithms* was coined to refer to algorithms based on a hierarchy of objects (the multiresolution tree) and a dynamic rendering front that adapts itself during the visualization. View-dependent algorithms include FarVoxels, LayeredPointClouds, TetraPuzzles, Quick-VDR and others. We briefly review these algorithms in the next paragraphs. In them, the front update is based on a suitable benefit function, but in all these algorithms no information about the frame computing and rendering time is taken into account.

Far Voxels [3] uses hybrid multiresolution Kd-trees, with triangle strips of the original scene model in leaf nodes and approximates volume representations in internal tree nodes. These nodes are discretized into a fixed number of around 16K voxels. Voxels contain parameterized direction-dependent material models, generated by sampling the geometry in the node along rays emanating from 256K viewpoints around it. The rendering algorithm uses

a front update algorithm based on the size of the projection of the nodes in the viewport, performing reasonably well for inspection tasks in complex environments.

The Quick-VDR algorithm [6] uses a Cluster Hierarchy of progressive meshes (CHPM) organized in a tree. The algorithm is aimed at the interactive inspection of huge triangular meshes. Tree nodes contain progressive meshes, the least simplified version of the mesh in a node is the union of the best representations in its children. Dependencies between nodes are used to avoid artifacts between neighbor nodes with different levels of detail in the viewport. Apart from using dependencies, the dynamic front management is based on the standard view-dependent scheme. A related work is [10]. In this paper, the authors discuss an optimization algorithm to compute coherent mesh layouts, and use them to improve the efficiency of the view-dependent rendering and collision detection algorithms in [6].

The Tetra Puzzles approach [11] is an efficient technique for out-of-core construction and accurate view-dependent visualization of very large surface mesh models. The method uses a regular conformal hierarchy of tetrahedra (organized in diamonds) to spatially partition the model. Each tetrahedral cell contains a precomputed simplified version of the original model, represented using cache coherent indexed strips for fast rendering. The view-dependent algorithm uses out-of-core and batched rendering techniques, with metrics based on the visual quality but with no budget for the frame rendering time.

Gobbetti et al. proposed a suggestive approach in [12]. Layered Point Clouds is an efficient multiresolution structure for rendering very large point sampled models on consumer graphics platforms. Tree nodes contain partial point clouds that are combined to produce the rendered primitives per frame. Sample densities are locally and dynamically adapted, according to their projected size on the viewport. The progressive block-based refinement nature of the rendering traversal uses prefetching, view frustum and occlusion culling, as well as compression and view-dependent progressive transmission. Remarkably, the authors recognize that, for interactive applications, it is often useful to have a direct control on rendering time, instead of being only based on metrics and tolerances on the rendering quality. Lacking an *a priori* estimate of the cost as in [8], their proposed solution iterates tree-traversals while adjusting the threshold to meet the budget.

Giga Voxels, as proposed by Crassin et al. [13], is an algorithm to efficiently render large volume datasets. The solution is based on an adaptive data representation depending on the current view and occlusion information, coupled to an efficient ray-casting rendering algorithm. Filtering, occlusion culling, procedural data creation, and level of detail mechanisms are integrated in an efficient GPU voxel engine. Data production and streaming is guided from information extracted during rendering.

A view-dependent approach for the interactive rendering of large-scale urban models has recently been proposed in [4], based on the Omni-Directional Relief Impostors (ORIs, see [14]). The approach is oriented to medium-range distance visualization of massively photo-textured cities. The authors use an image-based approach in the multiresolution tree. For each node of the tree, a set of relief maps that provide a multiresolution representation of the urban scene is stored. The rendering algorithm combines relief mapping with projective texture mapping, using only a subset of the precomputed relief maps, and wavelet compression to simulate two additional levels of the tree. The scheme is claimed to run considerably faster than polygonal-based approaches. Our approach is related to this method, using nodes with relief impostors in one of the tree layers, but improves it by considering a cost function besides the benefit function in an optimization framework.

Other proposals to visualize very large models have been made, including [15,16] and [17], for example, but like the schemes above, they cannot distinguish the different objects that make up the scene. Moreover, we observe that Funkhouser's paper (which was focused on moderately large scenes) contained some key ideas that have not been used, nor adapted to scene trees. Most present algorithms do not consider view-dependent rendering as a constrained problem with an upper bound in the rendering cost of the primitives. Of the solutions discussed above, notice that only Far Voxels and the method in [4] lend themselves to treat multiple, distinct objects.

3. View-dependent, front-based rendering

In this section we propose a taxonomy of algorithms using rendering fronts in Multiresolution Trees by sorting the existing algorithms into two categories: those using visual contribution fronts and those adopting more advanced fronts. We then introduce new concepts: tree monotonicity – an extension of Funkhouser's list monotonicity – and constrained fronts, and discuss the properties and advantages of fronts fulfilling these conditions, and show how to achieve them.

Multiresolution geometric models supporting view-dependent rendering must encode the steps performed by a simplification or coarsening process in a compact data structure from which a virtually continuous set of variable-resolution models can be efficiently extracted, [11]. Multiresolution Trees are a well-established data structure for this purpose.

A **Multiresolution Tree** is a hierarchical scene representation which encodes parts of the scene at a full range of different resolutions. The leafs of any subtree constitute a representation of a portion of the scene, with possibly mixed resolution levels. Multiresolution Trees have been extensively used for the representation of huge triangular mesh models [11], huge assemblies [3] and volume models [5]. They can encode scenes with multiple objects or highly complex meshes.

We define the following concepts concerning Multiresolution Trees:

The **Visual Contribution** $v(n, \mathcal{C}) \geq 0$ of a tree node n viewed with camera $\mathcal{C}$ measures the benefit of rendering node n with that camera in terms of the final visual quality. Authors often define $v(n, \mathcal{C})$ as an empirical function of the complexity of the geometric information in n and a number of view-dependent parameters like the size of the node projection in display coordinates. See the next section for our choice of $v(n, \mathcal{C})$. The visual contribution of nodes outside the visualization frustum is zero.

Tree Monotonicity with respect to a function $f(n)$ is an essential property in Multiresolution Trees. A Multiresolution Tree is said to be monotonic with respect to the function $f(n)$ if, for any node n and for any rendering conditions, $f(n) \leq \sum_{m \in sons(n)} f(m)$. For example, tree monotonicity with respect to $v(n, \mathcal{C})$ ensures that we will get better visual qualities when we render deeper tree levels with camera $\mathcal{C}$, a strongly desirable property for Multiresolution Trees. Of course, the maximum visual quality is reached when all tree leaves with unsimplified geometry are rendered.

A **Staircase Subtree** is a subtree such that if a node is in the subtree, all its siblings are also included. Notice this implies that this subtree is rooted at the root of the original tree.

A **Front** F is the set of leafs of a Staircase Subtree S_F. In what follows, we will note n_F the number of nodes in F. Any front partitions the multiresolution tree. From the point of view of rendering quality, one wishes the trimmed subtree S_F to have interior nodes with insufficient visual contribution, while the multiresolution tree nodes trimmed out – those with an ancestor in the subtree – have unnecessarily high quality for the frame being rendered. Any front F is a representation of the scene for certain choice of resolution at each portion of the scene. Observe that

any multiresolution tree can generate a huge combinatorial set of potential view-dependent fronts F.

Two different approaches appear in the literature to prune the scene tree at rendering time. We call them Visual-Contribution Fronts and Constrained Fronts, which we define next.

Visual-Contribution Fronts F_v are based on the visual contribution function $v(n, \mathcal{C})$ and a quality threshold Q_v. The visual contribution front F_v is the set of leaves of a staircase subtree such that all the interior nodes have a visual contribution smaller than Q_v, and where the leaves have a visual contribution larger than or equal to Q_v, or are leaves of the complete tree. Usually, the front F_v is computed at each frame through a top-down tree traversal clipped by the frustum. At each frame, the new front is usually computed in the CPU and sent to the GPU for rendering, but avoiding sending information which is already residing in the GPU. Therefore algorithms using this approach do not take the frame-rendering time into consideration.

For time-critical rendering, we need to have an estimation of the cost of rendering a given node, which we shall denote by $c(n)$. We consequently call $c()$ the **Cost Function**. The cost of rendering a given front F is then $c(F) = \sum_{n \in F} c(n)$. In polygonal models, for example, the rendering cost is proportional to the number of polygons in the geometry of n, so the cost in this case can be measured as the number of polygons. It could also be measured in Bytes, as the memory size of the geometry in n is also proportional to the number of polygons.

Constrained Fronts F_c have a bounded cost $\sum_{n \in F_c}(c(n)) \leq \mathsf{MaxCost}$. They are required by time-critical visualization algorithms, which guarantee a predefined minimum frame rate.

At each frame, the time-critical algorithm must solve a constrained optimization problem, i.e. finding the constrained front which maximizes the total visual contribution:

$$\arg\max_{F_c} \left(\sum_{n \in F_c} v(n, \mathcal{C}) \right). \tag{1}$$

As observed by [8] this is a knapsack-type problem, and suboptimal solutions must be considered. A good option is the greedy front update scheme already proposed in [8].

To use Constrained Fronts, Multiresolution Trees benefit from being cost-monotonic (monotonic w.r.t. the cost function), to guarantee that the total cost can be decreased by moving up the tree. Cost monotonicity is usually ensured by the bottom-up construction process of internal nodes.

Several properties follow easily from these definitions:

Property 1. Visual-Contribution Fronts can guarantee a certain image quality, but cannot guarantee a given frame rate. Constrained Fronts, on the contrary, can guarantee a frame-rate, but at a variable image quality.

Notice that the property that characterizes a Constrained Front is a global one, and therefore Constrained Fronts cannot be computed by a top-down tree traversal, since the information about a node (given by the pair $(v(n, \mathcal{C}), c(n))$ is insufficient to ascertain if it needs to be refined or not. Thus, the constrained optimization transforms the problem into a global one, which may be solved using tree monotonicity with respect to cost and visual contribution, with a greedy front update per frame [8]:

Property 2. Constrained fronts F_c must be computed by updating the front of the previous frame. They cannot be obtained by a top-down tree traversal.

When the camera moves suddenly and drastically, an algorithm computing F_v faces the same task as in any other case: it will traverse the tree, starting from the root and collecting the necessary nodes in the front. It may be hindered only by the need to exchange more information with the GPU. On the other hand, an algorithm computing a Constrained Front F_c will usually face, in this case, an exceedingly large number of necessary front-update operations, which cannot be met within the budget. For this reason, rendering algorithms which use constrained fronts rely on lazy updates and CPU–GPU transmission algorithms. At each frame, only a few updates are performed and sent to the GPU:

Table 1
Classification of the papers discussed in Section 2 according to the kind of front they use, and whether the objects are distinguishable in the rendering data structures.

	Object identifiability	
	No	Yes
F_v	[4], [11], [13,12], [3], [6], [15] and [17]	–
F_c	[9]	[8], Our proposal

Property 3. Confronted with drastic camera movements, an algorithm computing F_c will still comply with the frame-rate, but will need several frames to maximize the image quality.

Because fronts F_v are usually computed at each frame from the root of the tree, whenever a node is occluded or outside the frustum it can be discarded, along with all of its progeny. In the case of a front F_c, however, one needs to keep in it a complete representation of the scene, to be able to perform incremental updates even when some new nodes of the tree become unoccluded or enter the view-frustum:

Property 4. Time-critical rendering algorithms require specific data structures to manage visibility; see for example Section 6, for our approach to this problem.

Summarizing, the four properties discussed above show that time-critical visualization with a rendering budget requires a monotonic Multiresolution Tree with respect to both $c(n)$ and $v(n, \mathcal{C})$, and should be based on a constrained front F_c, with local, incremental front updates at each frame, lazy transmission and updates, and visibility management. As Table 1 shows, the literature is lacking in contributions addressing all of these requirements. However, a sustained frame rate is desirable in many applications, and generally improves perceived responsiveness of the application, in spite of isolated decays in image quality when the camera is changed drastically. The approach presented in this paper is intended to address this need.

4. Hybrid multiresolution trees (*HMT*s)

To meet the requirements discussed in the introduction, we have developed a visualization algorithm based on a specific scene binary tree, a Hybrid Multiresolution Kd-tree (*HMT* in what follows). *HMT*s contain three different layers of nodes and associated object data. Tree leaves contain the exact geometry of the objects, and constitute the *Exact-layer*. Nodes above them, but at depths larger than three make up the *SP-layer*, and contain simplified polygonal representations of the portion of the scene described by all the leaves that are descendants of them. The nodes in the upper levels of the tree form the *RI-layer*, and contain the most aggressive simplifications of their subtrees. Based on our experiments, we observed that setting the RI-layer to the top three or four levels yielded a fair compromise between speed and image quality. To choose how to populate them, we studied the behavior of some of the existing algorithms for the visualization of complex scenes with many differentiated objects, we performed a set of tests to experimentally compare the direct visualization of the polygonal scene with the main alternatives. From the results of our tests (detailed in Section 7.1) we conclude that image-based representations (specially, ORIs, see [14]) are well behaved in nodes with extremely complex geometry, presenting a better efficiency (measured in frames per second) and a good perceptual visual quality. Because of their better behavior, we adopt ORIs as the representation for the *RI-layer*.

*HMT*s can support selection of objects and groups of objects by clicking anytime on any of their triangles, as discussed in Section 5, making *HMT*s object-aware.

Our algorithm to render *HMT*s implements a constrained front to address the requirement of a guaranteed frame-rate. Each tree node n has an associated cost $c(n)$ and an associated visual contribution $v(n, \mathcal{C})$. The cost and visual contribution satisfy these monotonicity properties:

- For any non-leaf node in the tree, its cost is lower than the sum of the costs of its direct children.
- For any non-leaf node in the tree, its visual contribution is lower than the sum of the visual contributions of its direct children, regardless of the camera location.

We generalize Funkhouser's algorithm to render the *HMT* data structure, and try to (suboptimally) solve a constrained optimization problem at each frame. In what follows we discuss our choices to define the cost and visual contribution of the nodes in the *HMT*. Readers already familiar with [8] may find it preferable to skip to Section 5 first, and return to these details later.

The cost is static, and is computed for each node during the tree generation. The cost of nodes in *SP* and *Exact* layers is defined as the number of triangles in the corresponding node representation. Therefore, in these nodes, the rendering time is roughly proportional to their cost. Cost in the *RI* layer is computed as an equivalent triangle count, based on their rendering time.

The visual contribution of a certain node is a function which estimates its contribution to the overall perceived image quality. It is dynamic, depending on some intrinsic node information and on the camera parameters during the navigation. The visual contribution $v(n, \mathcal{C})$ is defined as a base term $v_0(n, \mathcal{C})$ which is modulated by several other camera-depending functions. For a given node n that passes the frustum culling, we define the dynamic visual contribution function $v(n, \mathcal{C})$ by the following expression, inspired on the ideas from [8]:

$$v(n, \mathcal{C}) = v_0(n, \mathcal{C}) * \mathit{Centered}(n) * \mathit{Change}(n) * \mathit{Vis}(n).$$

The base visual contribution function $v_0(n, \mathcal{C})$ for a node n is a function of its projected size $p(n, \mathcal{C})$ and of its level in the tree $l(n)$ (where the root sits at level 0). It measures the contribution of the node to the overall visual quality of the image, when n is projected and remains in the center of the viewport. The projected size $p(n, \mathcal{C})$ is the surface area (in pixels) of the screen projection of the axis-aligned bounding box of n, *AABB(n)*. We define the base contribution as,

$$v_0(n, \mathcal{C}) = p(n, \mathcal{C}) * \mathit{Quality}(l(n)).$$

$v_0(n, \mathcal{C})$ models the fact that nodes with a large projection size will have a greater visual contribution to the final result. Moreover, nodes in deeper tree levels present a higher visual quality, as modeled by the increasing function *Quality*. In our implementation we have used $\mathit{Quality}(k) = \sqrt{k} + 1$, for which we show below that it results in monotonic visual contributions. Other increasing functions for which a monotonical contribution could be proved would be suitable replacements, but we have found that this gives good results.

The remaining three factors in the definition of $v(n, \mathcal{C})$ modify $v_0(n, \mathcal{C})$ by taking into account the camera parameters and their variation over time. They are controlled by coefficients set experimentally. For the first two, we have found that limiting their influence to a maximum reduction of 20% gives good results. This represents a compromise between the two extremes. If these functions are given much larger strength, then they may reduce the value of a nodes' contribution excessively, inducing in turn a simplification much larger than really necessary, and ultimately increasing the number of artifacts. If they are given to little influence, the result will be that nodes that are moving fast, or near the boundaries of the viewport, will be excessively detailed, producing an increase in cost, and since we run on a limited budget, inducing a decrease in quality somewhere.

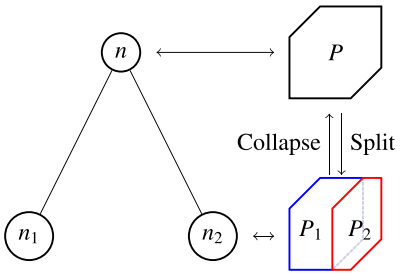

Fig. 2. Base Visual contribution is monotonic. The Split operation increases it, while the Collapse operation decreases it.

The first one is the Centered function:

$$\mathit{Centered}(n) = 0.2\left(1 - \frac{\mathsf{dist}}{\mathsf{vpsize}}\right) + 0.8$$

which decreases with the distance *dist* from the center of the viewport to the projection of *AABB(n)* onto the viewport (here vpsize is half the diagonal of the viewport). Thus, *Centered(n)* is one if the projection contains the center of the viewport, and if not, it decreases down to 0.8 near the boundary of the viewport. This factor attempts to capture the loss of visual acuity in the viewport periphery.

Change(n) is a decreasing function, measuring the rate of change in the projected position of the node. The main idea is that we have a reduced visual acuity for objects (nodes) presenting apparent movement in the viewport. In our implementation, we have found that a good compromise is to define

$$\mathit{Change}(n) = 0.2 \max\left(1 - \frac{\left\|C^n - C^n_{\mathsf{prevframe}}\right\|}{0.4 * \mathsf{vpsize}}, 0\right) + 0.8$$

where C^n is the center of the projection of the *AABB* of node n.

Finally, *Vis*(n) takes occlusions into account. For efficiency reasons, we use information from occlusion queries from the previous frame. We define $\mathit{Vis}(n) = \mathit{VisPixels}/\mathit{TotalPixels}$, where *TotalPixels* is the projected size $p(n, \mathcal{C})$ and *VisPixels* is the number of visible pixels of *AABB(n)*, both in the previous frame.

Let us prove now that base visual contributions are monotonic. To this end, let us now consider the base visual contributions of node n and of its two children. To simplify the notation, assume that p is the surface area of the screen projection of n, and p_1 and p_2 the areas corresponding to its children nodes n_1 and n_2. We do not indicate explicitly from here on the dependency on $\mathcal{C}$ of these surface areas, in the interest of more compact equations. To see that the base visual contribution is locally monotonic (see Fig. 2), notice that $p \leq p_1 + p_2$, so

$$\begin{aligned} v_0(n, \mathcal{C}) &= \left(\sqrt{l(n)} + 1\right) p \leq \left(\sqrt{l(n)} + 1\right)(p_1 + p_2) \\ &< \left(\sqrt{l(n) + 1} + 1\right) p_1 + \left(\sqrt{l(n) + 1} + 1\right) p_2 \\ &= v_0(n_1, \mathcal{C}) + v_0(n_2, \mathcal{C}). \end{aligned}$$

So

$$v_0(n, \mathcal{C}) < v_0(n_1, \mathcal{C}) + v_0(n_2, \mathcal{C})$$

no matter what camera $\mathcal{C}$ we are using.

The (approximate) optimization performed during rendering generalizes [8] to hierarchical data structures and is detailed in Section 6.

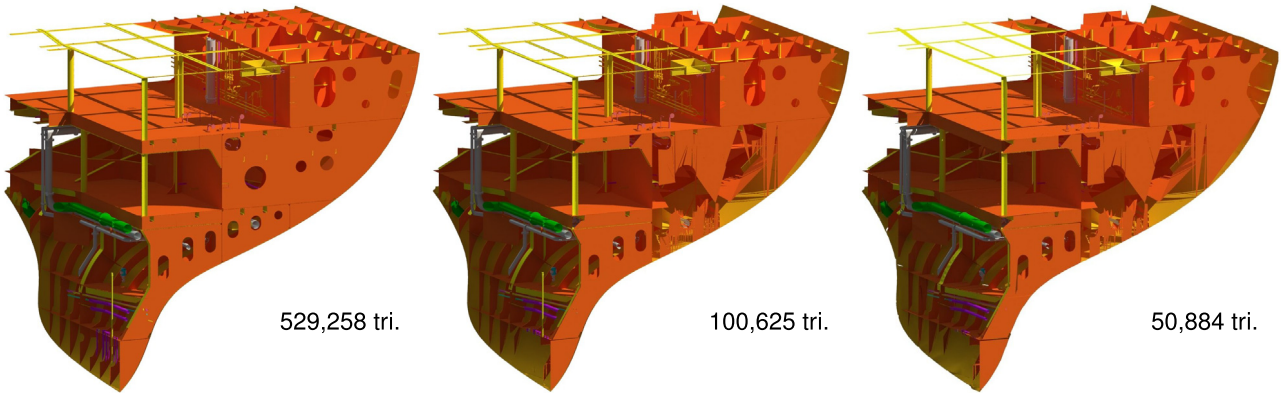

Fig. 3. Node simplification: The leftmost image shows the exact geometry corresponding to one of the nodes in the Kd-tree. In the center image, some of this geometry has been removed by the visibility culling (which is conducted including geometry from neighboring nodes, since the simplified versions are not meant to be used from close-up). The rightmost image shows the result of the simplification of this node. Notice that there are large holes in the front wall; they happen because those portions of the wall are occluded by geometry in the neighboring node, and will therefore never be visible when this simplification is used.

Monotonicity of dynamic visual contributions comes from the fact that the *Centered*, *Change* and *Vis* functions are not significantly different between any node n and its children. Moreover, the cost of *HMT* nodes is monotone by construction, as shown in next section. We can therefore conclude that *HMT*s are monotonic and support constrained fronts F_c. We now turn to the details involved in the construction of an *HMT*.

5. Generation of the *HMT*

*HMT*s can handle huge models distributed among several input files. These files contain triangles with color per vertex and data is structured in objects that point to triangle lists. Objects in the input files are assigned sequential integer labels, and a single triangle soup is created and stored in external memory. Triangles in this soup contain their object index as an attribute. The next step is the creation of the Kd-tree structure. This is done by a standard recursive space splitting algorithm. Every node in the Kd-tree represents an axis-aligned box and all triangles contained in it. The box of the root corresponds to the scene's bounding box. At each step, the recursive algorithm tests the three coordinate directions and chooses the best (most centered) orthogonal plane that splits the set of triangles in the father node into two sets with similar cardinality. Long triangles – whose extent into both sub-nodes is substantial – that are stabbed by the splitting plane are subdivided, whereas the rest of stabbed triangles are simply assigned to one of the two new regions. In practice, very few triangles need to be split, and at any rate this only impacts the pre-processing of the scene. Subdivision is repeated until nodes contain less than 50K triangles each (this number has been experimentally determined to be the size for VBOs that achieves good performance across diverse GPUs). At the end of this step, the Kd-tree structure has been generated and leaf nodes (the *Exact-layer* of the tree) already contain their final geometry.

Nodes in the *SP-layer* are computed by bottom-up simplification, starting by the parents of the already defined *Exact-layer* nodes. After many experiments and tests, we concluded that standard simplification algorithms were not adapted to huge assemblies with a large number of objects and many disjoint meshes. Our simplification scheme uses volumetric techniques with surface reconstruction per node, and proceeds as follows:

1. Visibility culling. The node is enlarged with one layer of neighbor leaf nodes, and it is then rendered from 320 directions around its center. Triangles which do not appear in any of the renders are marked as invisible. Invisible triangles in the node can be removed, as objects close to the camera will be always rendered as *Exact-layer* nodes, while nodes in the *SP-layer* will be used in medium distances (see Fig. 3, middle).
2. Complexity test. If the node, after visibility culling, has less than 50K triangles, we are done. If not, we proceed with the next steps:
3. Node voxelization. We extend each node's box by a 5% in each direction to avoid cracks between nodes, we fill it with geometry from neighbor nodes, and voxelize it. We denote by Nv the number of voxels in the longest extended box direction. The number of voxels in the other directions is computed to guarantee almost cubic cells. Voxels can be either void or full. We initialize $Nv = 200$.
4. Surface simplification. Our algorithm is based on [18] and on [19]. For every voxel v, we classify all vertices of the initial model in v. Vertex labels are assigned according to the index of their object and to the direction of their normal vector. We consider eight direction classes by packing the signs of the three normal components. Triangles having two or three vertices with the same label are considered dangling and removed. Then, all vertices in v having the same label are collapsed into their centroid, which becomes their representative vertex.
5. Clipping. The resulting geometry is clipped inside the box of the node (see Fig. 3, right).
6. Cost computation. The resulting number of polygons in the node (cost) is tested for monotonicity. If the cost is not lower than the sum of the children's costs, Nv is decreased by a 10% and steps 3... 6 are repeated.

The upper tree layer (the *RI-layer*) is computed with an algorithm based on [14], but modified for our purposes. For each node in the *RI-layer*, we render its geometry on a 300 × 300 viewport, from 102 almost isotropic directions and using orthogonal projection. Color, depth and normals per pixel are stored in one relief impostor per view direction (relief impostors are implemented as two textures). Neighbor nodes are also rendered, to complete the information in the impostors. We derive the 102 viewing directions from the regular subdivision of an octahedron, as axis-aligned viewing directions are relevant in industrial design. In our implementation, the *RI-layer* fills levels 0 . . . 2 of the *HMT*, and each node in this layer is represented by 30 or less relief impostors. A node-dependent rendering simulation is performed in order to discard poor impostors and find a set with (at most) 30 best impostors. This rendering step is based on [14]. It renders the node from 320 equally-distributed viewpoints, derived from the subdivision of an icosahedron. For each one of them, it tests different subsets of the eight relief impostors which are closest to the viewing direction. For each subset, the

rendered image of the node is obtained and a visual error metric is computed as the mean squared error in the per pixel perceptual comparison between this image and a polygonal rendering of the node geometry from the same viewpoint. The best subset for a certain viewpoint direction is the one giving the lowest error value. Errors are also accumulated in the contributing impostors. Impostors are not used in the rendering subset for any of the tested directions, and impostors with high accumulated errors are removed from the initial set, until the target of 30 *RIs* is reached. In short, we use 102 renders per node to compute candidate impostors, and 320 extra RI-renders per node to discard non-informative impostors. Nodes in the *RI-layer* are assigned a cost which is constant (because of the uniform number of impostors per node). Cost monotonicity is therefore guaranteed by construction. The cost value is computed as an equivalent triangle count, based on the node rendering time.

As a last test, cost monotonicity is checked between the upper *SP* layer and the lowest *RI* layer of the tree. If necessary, the parameter *Nv* of the upper *SP* layer nodes is further decreased until the whole tree is monotonic with respect to the cost. This update has never been necessary in the examples we have tested.

In order to make the *HMT* object-aware, we store for each triangle an integer identifier of the object it belongs to. In order to efficiently access material properties we store them in an $N \times N$ *Object Texture*, with N such that N^2 is greater than or equal to the total number of objects in the scene. Observe that a texture size of 1024×1024 supports more than 10^6 differentiated objects, which is sufficient for large current-day industrial assemblies. In our implementation we use one *RGBA* texture storing the color and transparency attributes of each object. Given an object index *oi* we use the equations $u = oi \bmod N$ and $v = oi \operatorname{div} N$ to compute the (u, v) coordinates of the texel storing the color attribute. Extra attributes could be stored, but if they do not participate in rendering, they can be directly accessed through other data structures in the application.

6. Visualization and interactive scene editing

The visualization algorithm is based on a dynamic constrained front. The front F_k is the list of *HMT* nodes to be rendered at frame k, together with the device coordinates of the center of $AABB(n)$ for each front node. The front also includes upper tree representations of regions outside the current frustum, to support efficient update in fast camera movements. The front structure is CPU-based, but front nodes are cached in the GPU as Vertex Buffer Objects when appropriate.

The rendering algorithm solves the constrained optimization problem in Eq. (1) at each frame, trying to maximize the total visual quality of the rendered image while ensuring that the total cost is not greater than *MaxCost*.

The optimization starts with the front at the previous frame and updates it. Front update is based on two operations, node split and node collapse (see Fig. 2). A node split refines a certain node n and increases the cardinality of F_k by one, increasing also the total cost and the total visual contribution of the front (because of the monotonicity property). Collapse acts on a pair of siblings and replaces them by their common parent node, decreasing the front cardinality by one, decreasing by the same token the front cost and visual contribution.

Front nodes are labeled as nodes *outside the present frustum*, *occluded nodes* or *visible nodes*. *Occluded nodes* are nodes that were completely occluded in the previous frame. *Visible nodes* are nodes neither occluded nor completely outside the frustum. In what follows, we will use the term *invisible nodes* for the union set of *occluded nodes* and nodes *outside the frustum*.

The front update implements a greedy suboptimal optimization and transforms F_k into F_k^u. This is achieved in three substeps. First, pairs of siblings such that both of them are *invisible nodes* are collapsed. This is repeated until no pair of invisible siblings exists. Then, the dynamic visual contribution of every *visible* node in F_k is computed. Afterwards, the *visible* node n with the maximum dynamic visual contribution $v_d(n)$ is detected and refined, being substituted in F_k by its two children, which become *visible*. Finally, candidates to collapse are computed and collapsed. Candidates to collapse are pairs of brother nodes, both in F_k, with at least one of them being *visible*. The candidate pair (n_1, n_2) with the lowest $v(n_1, \mathcal{C}) + v(n_2, \mathcal{C})$ is collapsed (both nodes are substituted by their parent, which is labeled as *visible*), and this collapse is repeated (with the next remaining candidate pair with the lowest joint visual contribution) until the total front cost is lower than *MaxCost*. The resulting updated front is named F_k^u. Next, the GPU cache is updated by transmitting the (few) changed nodes that have a *visible* label.

Finally, the front F_k^u becomes F_{k+1} by updating the occlusion labels. Occlusion queries are sent by all nodes in F_k^u intersecting the frustum. In the case of *visible* nodes, one query per node is sent, but in the case of *occluded* queries, they are sent in groups (we use four nodes per group) to reduce the total number of queries and the frame update time.

In our implementation, the number of polygons per node is fairly constant by construction. Therefore, constraining the total cost to be lower than *MaxCost* is basically equivalent to constraining the maximum front cardinality. Our experimental results have shown a stable number of nodes in the front. Furthermore (see Section 7, especially Fig. 10), we have seen that in most cases, the numbers of splits and of collapses of *visible* nodes per frame are similar.

Our experiments show (see Section 7, Fig. 11)) that the increments in visual contribution due to the splits are always greater than the decreases produced by the collapses, empirically validating our greedy optimization. Nonetheless, the total visual contribution remains bounded because of the visibility changes and because of nodes leaving the frustum.

This lazy algorithm with suboptimal constrained optimization shows a good performance and user acceptance, as will be discussed. Since time-critical requirements disappear when users stop the navigating camera, we detect these cases, save the current front and temporarily increase the value of *MaxCost*, but allow only splits, to avoid oscillations. When the camera starts again the navigation, we retrieve the previous front and *MaxCost* and continue the standard per-frame optimization.

While this rendering strategy successfully brings together and extends several different techniques to produce quality navigations of very complex models, its true singularity rests in doing so while preserving each and every object in the original CAD model, which makes these navigations meaningful in the middle of the design loop, and better supporting collaborative design discussions. Operations on sets of selected objects – optionally selected directly in the CAD design tree – use the Object Texture. Selection feedback is performed via a change of the objects material color. Previous attributes in the texel corresponding to the selected objects are saved, and a temporary selection color is assigned to them. Upon the end of selection, original colors and attributes are retrieved and restored. Interactive dragging of the set of selected objects is possible, in sets of moderate cardinality. After the selection, a temporary Vertex Buffer Object is constructed and transferred to the GPU with the triangles of the selected objects. Edition in this case is performed by making the objects transparent in the *HMT* (temporarily modifying the Object Texture) and rendering and dragging this additional VBO of the selected objects. The overhead of rendering twice the set of selected objects is negligible in practical editing operations. Edition results are saved

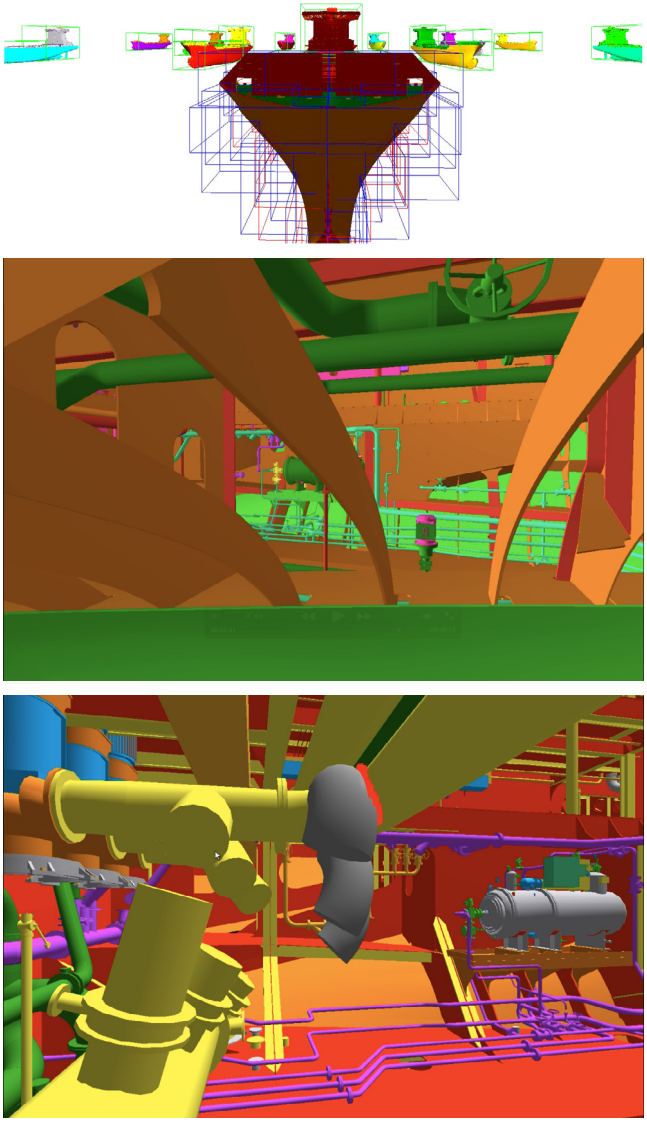

Fig. 4. The top snapshot shows a partial view of the fleet scene. Here we have drawn the AABB's of the nodes, color-coding their nature: green nodes are ORIs, red nodes contain simplified geometry, and blue nodes are leaves (with the exact geometry). The middle image shows a portion of the inside of the ship, with part of the hull removed. The image at the bottom shows the result of selecting and displacing three pipe elbows (in gray). Notice the red highlighting where the elbows interfere with other geometry. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

in a log file for ulterior batch update of the design and *HMT* trees. Selected objects can be made transparent in the same way, but in this case the Alpha component of the corresponding texel is modified. Unique global indices allow to log also other sorts of user annotations or modifications to the selected objects. Approximate collision detection is supported during edition and dragging of the selected objects in a hierarchical way (see Fig. 4, bottom). This is reminiscent of [20] in that bounding boxes are rotated with the objects. However, we deal with relatively simple objects and are therefore able to construct the collision hierarchy for the selected objects on the fly, instead of requiring a long pre-process. This is done at the leaf level only, since the user is assumed to focus his attention on these parts. To this end, we compute on the fly a six-level octree of the selected objects, and use it to check for collisions with a six-level octree of the leaf (which is computed during the pre-process and stored in each leaf of the *HMT*). The octree representation of the selected objects is computed in a second

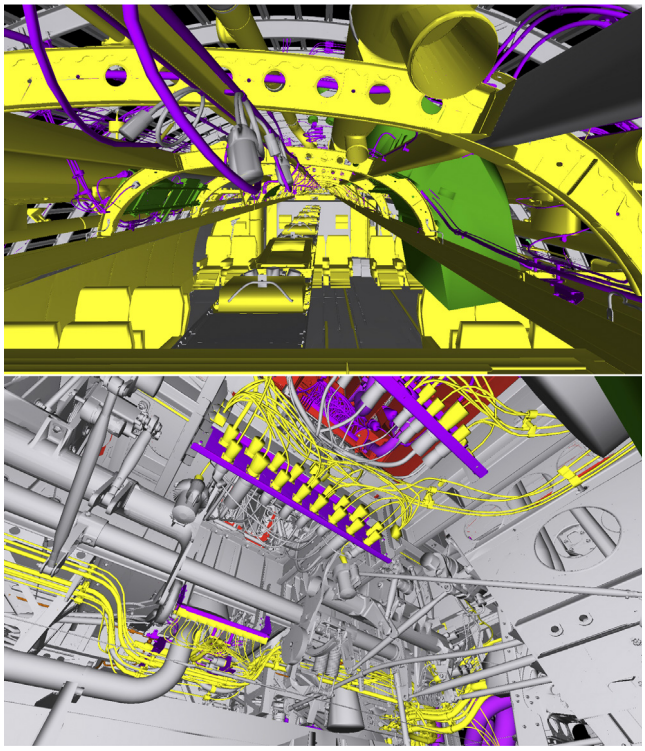

Fig. 5. Two frames from a navigation inside the Boeing 777-200. The top frame shows a view near the top of the cabin, where sight extends all the way to the rear end of the airplane. Notice that upon careful inspection, far elements, rendered as ORIs, can be distinguished as slightly more blurred. Nodes at a mid-distance are simplified, and geometry close to the camera is exact. The bottom figure shows a frame where we see that the model has preserved intact the details of the original for close inspection.

thread, while their polygonal representation is being transferred to the temporary VBO in the GPU. These two black-and-white octrees are stored in a compact, pointer-less depth-first representation, with two bits per node. They are bounding octrees: any point in the model is in the set of black nodes and any black node contains points of the model. This octree is transformed following the user interaction, so that the collisions are actually tested between an axis aligned octree (of the leaf-node geometry) and the resulting rotated octree. Obviously, the collisions are detected with an error up to the length of the diagonal of the leaf nodes of these octrees, which is sufficient for the applications considered. Changing the model would require a new pre-processing of the scene, so these changes are kept only as annotations that allow their reproduction, and may be afterwards processed in the CAD system if so desired.

7. Results

We have tested the performance of our algorithms using real models from the ship industry, which was our main focal application. The most complex ship model we were able to use in these tests – due to confidentiality restrictions – corresponds to a cargo ship of 219.124 m length overall, consists of some 14.4×10^6 polygons, and is shown in Fig. 1. This model contains some imperfections, where geometry interpenetrates other geometry, causing some artifacts, but our algorithm performs without any special repair, albeit displaying, of course, the same artifacts.

In order to test our algorithm under more stressing conditions, we have run it on a scene made up of a fleet of sixteen copies of this same ship model. We have also tested our results with the model of a Boeing 777-200 (see Fig. 5 and the second accompanying video).

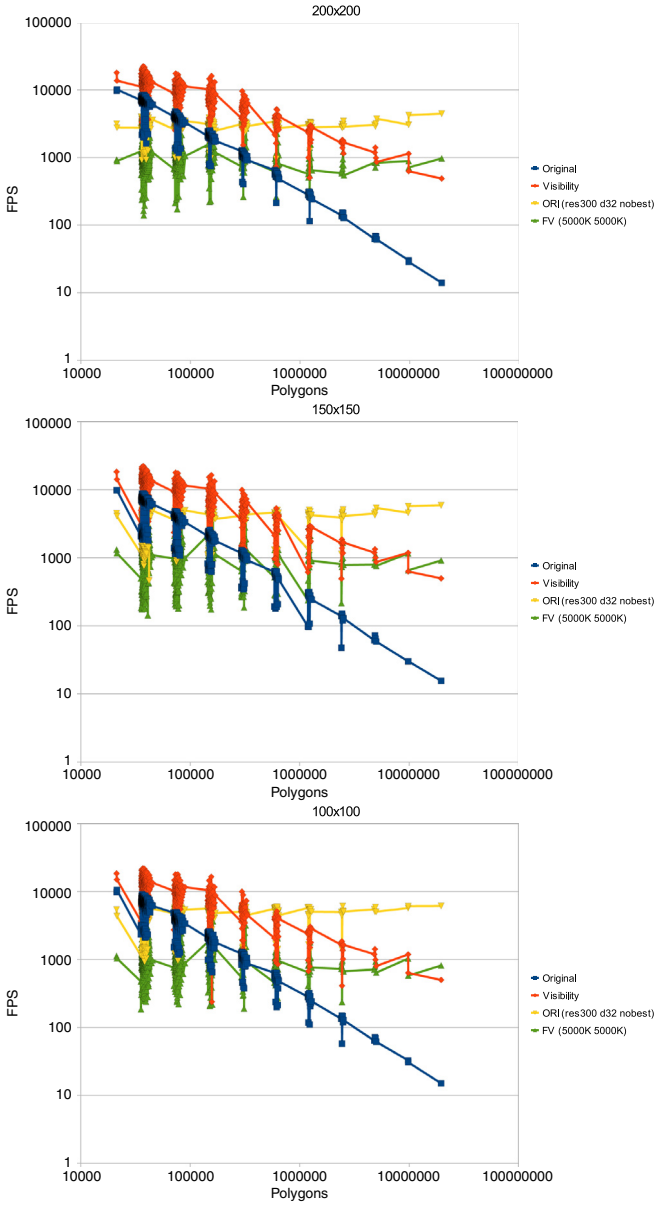

Fig. 6. Comparative speeds in different rendering techniques for large datasets in a 200×200 viewport (top), a 150×150 viewport (middle) and a 100×100 viewport. The test was run on a machine with a 4-core Intel i7 at 3.2 Ghz, with 12 Mb of memory and a GeForce GTX570 graphics card with 1.25 Gb of DDR5 memory, using Windows 7 Professional 64bits.

7.1. User-perceived visual quality in ORIs and far voxels

To understand the behavior of some of the existing algorithms in the visualization of complex scenes with many differentiated objects, we prepared a set of tests to experimentally compare the direct visualization of the polygonal scene with the Far Voxels approach [3] and with ORIs [14]. We selected these two approaches because of their ability to visualize huge scenes with distinct objects. We implemented a testing platform with different variants of these approaches and designed a suitable interface for the experiments.

Our first test was on rendering efficiency. We rendered different parts of a ship model with the previously mentioned visualization algorithms. Since we intend to use these algorithms to render nodes that are far from the camera, whose projection onto the viewport has a diameter of not more than 200 pixels, we performed three tests with viewport sizes of 100×100, 150×150 and 200×200 pixels. In each case, we rendered parts of the model of increasing complexity, see Fig. 6. The plots in this figure show the frame rates in four different cases: a polygonal rendering of the original geometry (no hardware occlusion culling); a polygonal rendering of the original geometry with internal, non-visible geometry removed; an ORI rendering; and finally, a Far Voxels rendering. The ORI consisted of 32 relief impostors from equally distributed directions, each of them having a 300×300 resolution. Far Voxels was implemented at a higher resolution than usual, with five million voxels per tree node to achieve similar visual quality at this resolution; five million rays were used for computing voxel material properties in each tree node.

From the results shown in Fig. 6, we can conclude that in our platform, rendering raw geometry is less efficient when the polygon count is larger than one million polygons, the consequence being a decrease in the frame rate. Moreover, the efficiency of ORIs is always higher than the corresponding one for the Far Voxels scheme. This is true independently of the viewport size.

To test the visual quality, we implemented a platform to visualize a chosen tree node simultaneously in six viewports, each one having 200×200 pixels. As we shall see in Section 7 (especially Fig. 12), nodes with ORIs are very rarely used at larger projection sizes. Users interact with the model and see exactly the same camera changes in all viewports. Users can load any tree node and experiment with it. The platform manager prepares the experiment by assigning different visualization schemes to each of the viewports. Fig. 7 shows the layout of the interface for this experiment.

The comparison of visual quality was performed by 20 users, in the aforementioned test platform. Each user could interact with eight randomly-chosen nodes from the full tree; for each node, the user could change the camera, and was shown simultaneously six viewports for 30 s with different algorithms (see Fig. 7). The six viewports displayed, from left to right, the reference polygonal model (ground truth), three ORI renderings obtained using 32 viewing directions and textures of 300×300 pixels (the leftmost is the original ORI algorithm, the next viewport is a blending between the original ORI render and a down-sampled version, and the third is upsampled with 4 rays per fragment). The fifth viewport displayed a repeat of the reference model in the first viewport, and the last viewport displayed a high resolution rendering with Far Voxels (with nodes of 5 million voxels, computed using 5 million rays). Each user was asked to grade the visual quality of each viewport with respect to the ground truth of the first one. The conclusion was that users clearly preferred the ORI rendering with respect to Far Voxels. Users gave the maximum grade to viewport 5, which they did not know was a repetition of viewport one, confirming their reliability in running the test.

We also compared the images themselves at a resolution of 200×200 pixels (the maximum resolution intended for them in our application). Given the images of the same node from the same viewpoint using Far Voxels (with the parameters above) and ORIs, we computed the RMS value of the deviations from the ground truth (represented by the exact geometry), measured for each pixel as the euclidean distance between the colors of the pixels in RGB space. The average was taken over the *relevant pixels* for each view, defined as those that were not background-colored in at least one of the three images. The results for some representative nodes are given in Table 2.

Two conclusions can be derived from the results of our tests. Nodes containing polygonal representations have a better performance when their cardinality is well below 1M polygons. Moreover, image-based representations (specially, ORIs) are well behaved in nodes with extremely complex geometry, presenting a better efficiency (measured in frames per second) and a good perceptual visual quality.

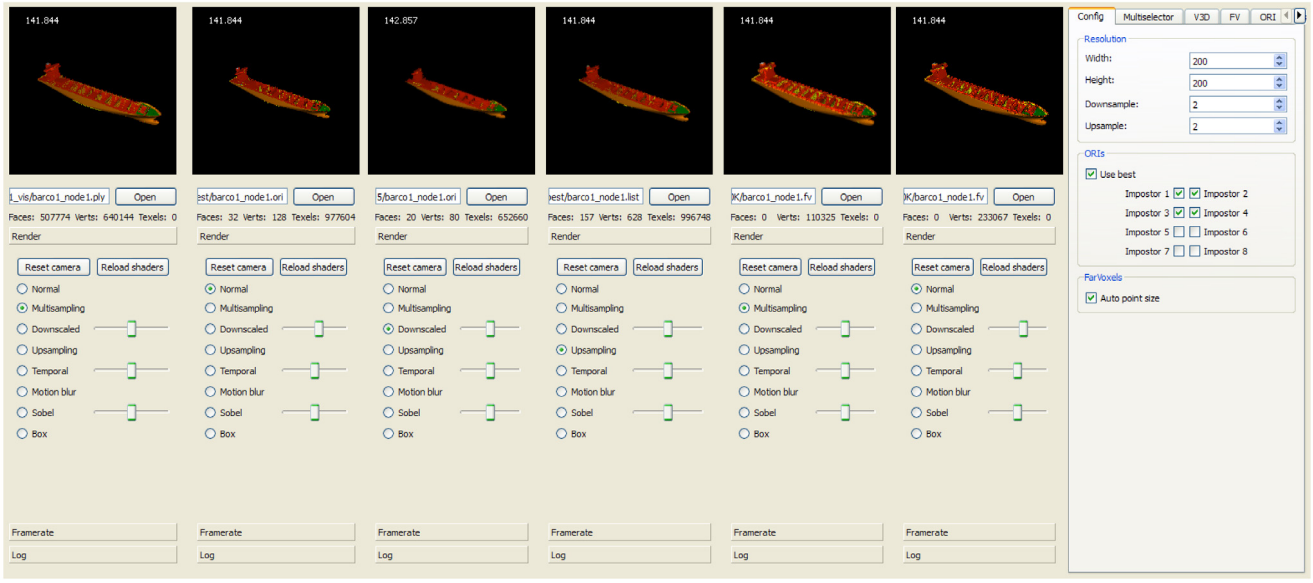

Fig. 7. Interface for the user perception study.

Table 2
Comparison of image quality for several nodes and random cameras. The root mean square averages are computed with respect to the relevant pixels, i.e. those that are not background.

Node	Relevant pxls	rmsE(ORI)	rmsE(FV)
1	4 419	0.1324	0.2657
2	10 019	0.1177	0.2434
6	9 048	0.1682	0.3141
740	13 653	0.2063	0.4340

7.2. Results analysis and discussion

Our algorithm applied to the model in Fig. 1 resulted in an *HMT* of 993 nodes and depth ten. Notice that a balanced tree of this size would have nine levels, so the *HMT* is close to balanced. This is a consequence of our Kd-tree construction algorithm, which enforces an as-balanced-as-possible distribution of primitives between siblings. The construction time for this Kd-tree took in this case just 58.84 s. In this *HMT*, the first three levels form the *RI-layer*. Each of the ORIs in this tree took between 3.59 Mb and 9.98 Mb. The simplified nodes occupied anywhere from 0.62 Mb to 4.47 Mb, and the leaf nodes with the exact geometry weighed in the range of 1.29 and 2.64 Mb. The pre-processing time for computing the ORIs was 3192 s and the simplified nodes took 8011 s to compute. The total number of triangles at the leafs of the resulting HMT was 17.5×10^6, due to the splitting of triangles at node boundaries. These timings correspond to running the pre-processing algorithm on a PC with an Intel Core Duo E7600 CPU at 3.06 GHz, with 8 Gb of RAM, and an nVidia geForce GTX280 graphics card, running Windows Vista Enterprise 64 bits.

The rest of the results with this model were run on the scene made up of sixteen copies of this model (each with distinct colors to distinguish them). Since the scene is, however, sparse, we represented it internally with a forest of sixteen *HMT*s, totaling 208×10^6 triangles in their leaf nodes. We have recorded a real-time execution of our application exploring this scene, which is shown in the first of the accompanying videos. The numbers given hereunder correspond to this execution on a PC with an Intel Core i7 with 4 cores, running at 3.2 GHz, and having 12 Gb of DDR3 RAM. The graphics card was an nVidia geForce GTX 570 with 1.25 Gb of dedicated memory.

We have attempted to use trajectories and operations that display the real potential of our proposal. Fig. 4 shows three snapshots from this video, presenting operations at the object level. Objects can also be annotated (see Fig. 1) as required in cooperative design.

Table 3
Size of each level of the HMT for the model in Fig. 5. The third column shows the number of nodes that contain ORIs, simplified geometry or full geometry. The last three columns show the minimum, average and maximum number of triangles per node for that level, and provide a coarse measure of the extent to which our algorithm produces a reasonably balanced tree.

Level	Num. Nodes	Num. ORIs/simp./leafs	Min	Avg	Max
0	1	1/0/0			
1	2	2/0/0			
2	4	4/0/0			
3	8	8/0/0			
4	16	0/16/0	43 360	63 243	78 565
5	32	0/32/0	53 036	68 793	96 291
6	64	0/64/0	28 004	60 765	79 155
7	128	0/128/0	51 649	66 157	89 414
8	256	0/256/0	30 878	62 802	76 425
9	512	0/512/0	39 886	65 408	74 954
10	1024	0/1024/0	22 325	57 707	74 969
11	2048	0/2045/3	29 906	64 150	89 841
12	4090	0/3489/601	23 079	61 932	91 321
13	6978	0/118/6860	21 909	48 572	78 094
14	236	0/0/236	33 454	44 190	69 893

The model of the Boeing 777 in Fig. 5 consists of more than 180×10^6 faces of different arity, and equivalent to more than 330×10^6 triangles. Table 3 details the sizes and characteristics of the resulting HMT. Notice that for this larger model we have used four levels for the RI-layer. Notice that in this case the first four levels are occupied by ORIs, levels four through thirteen contain simplified geometry, and leaves with exact geometry appear from level eleven onwards.

Fig. 8 shows the time spent in each of the frames of the first video. Notice that the vast majority of the frames are under 40ms, since we chose the cost to achieve a minimum frame rate of 25fps. This budget is only exceeded for isolated frames where drastic changes in the image content, and thus in the rendering front, occur (for example when the camera goes through a wall).

Next, we show in Fig. 9 the cost, as computed by the algorithm, of rendering each front. Notice that the plot reproduces the shape of the frame rate in Fig. 8, validating experimentally – in this case – our computation of the cost.

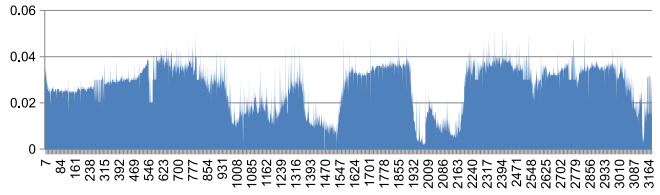

Fig. 8. Fleet scene: time for each frame.

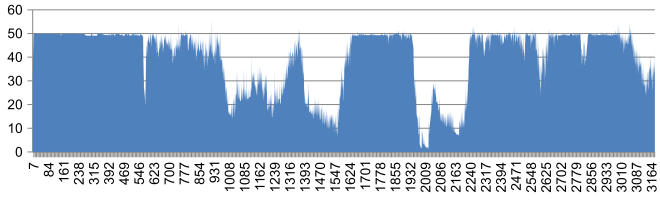

Fig. 9. The cost computed for rendering the front at each frame for the fleet scene.

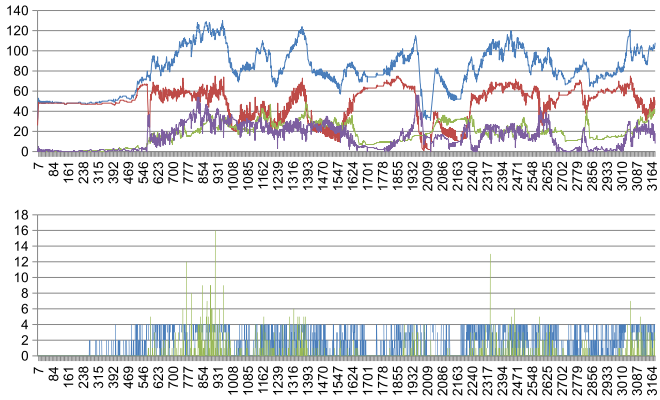

Fig. 10. The evolution of the front (top) and the effort to update the cache (bottom) in the video of the fleet scene. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

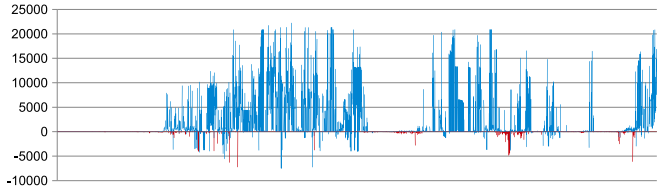

Fig. 11. Fleet scene: evolution of the total visual contribution of the front. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

In Fig. 10 we show at the top a plot of the size of the front at each frame. The curves show the number of nodes of each kind: total number of nodes in the front (in blue), visible nodes (in red), culled by the frustum (in green) and culled by the occlusion test (in purple). Notice this total is consistently below roughly 120 nodes. The plot at the bottom of the figure shows the number of nodes updated in the GPU cache at each frame, which is almost always eight or less. Blue bars correspond to nodes uploaded to the GPU due to split and merge operations. The green bars show nodes uploaded due to changes in the visibility of nodes. The few isolated spikes correspond again to instances where this visibility changes abruptly.

Next, we show in Fig. 11 the evolution of the visual contribution of the nodes in the front. In this plot, the blue bars represent increments in visual contribution due to splits, whereas the red plot represents (in negative values) the losses of visual contribution due to collapses. Notice that increases are predominant, thus validating, again at an experimental level, our heuristics.

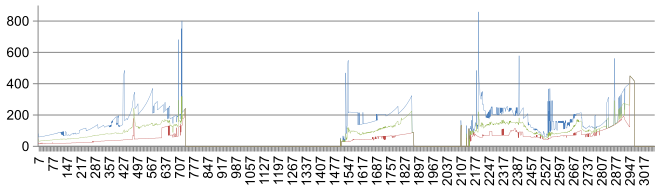

Fig. 12. Fleet scene: sizes of the projections of the ORIs onto the viewport per frame in the first test video of the fleet scene. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

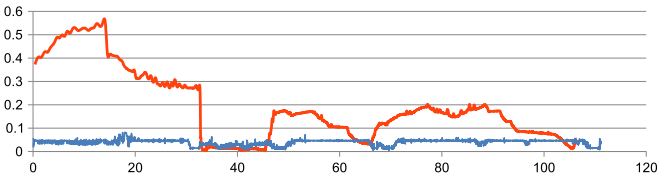

Fig. 13. Comparison of time spent per frame by drawing visibility-culled polygons (in red) and by our algorithm (in blue) along a test trajectory in our fleet scene. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

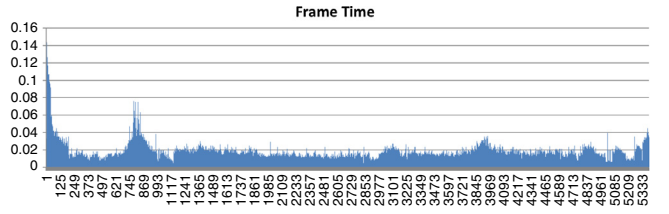

Fig. 14. Time to render each frame for a navigation inside the Boeing 777. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Fig. 12 shows the maximum (blue), minimum (red) and average (green) size of the nodes in the RI-layer (rendered using ORIs) that appear in each frame. While the maximum undergoes some sharp spikes, notice that average values (and even maximum values for most frames) roughly stay under the threshold of 200 pixels. Notice that these plots represent the size of the projection of the whole node. In most cases, large ORIs, in the few instances that occur, correspond to large nodes in the HMT that are almost completely occluded.

Fig. 13 offers a comparison of the time spent per frame at different points in the first accompanying video (the fleet) by two different algorithms. The abscissae in this case represent seconds of video. The red curve plots the time spent when drawing the full polygonal model using Coherent Hierarchical Culling [21] (culling polygons that are not visible, which accounts for the reasonable speed at certain points along the video). The blue line represents the time per frame spent by our algorithm, which remains downright flat.

The tests on the Boeing 777 model were run on a single-threaded application in a PC with an i7-5820K CPU and 32 Gb of RAM, and rendered using a GeForce 980Ti GPU. Fig. 14 shows the time it took to render each frame in a navigation inside this model. The first frames are much slower until data are updated to the GPU, and then sustain good frame rate. A slight decrease in frame rate near frame 800 corresponds to a sudden increase in the number of visible nodes after going through a wall. Finally, Fig. 15 shows, with the same color codes as the top graphic in Fig. 10, the evolution of the number of nodes in the front during the same navigation reported in Fig. 14. Notice that, despite the much higher complexity of the model, a very large proportion of it becomes culled by the frustum culling, as seen from the approximately constant gap between the total number of nodes and the number of nodes culled (blue and green curves).

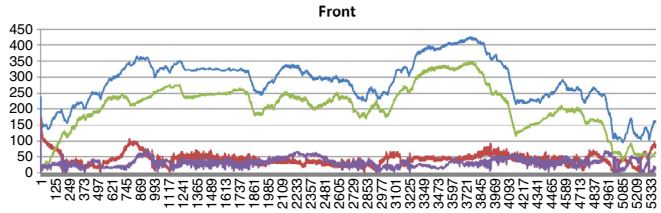

Fig. 15. Evolution of the front for the same navigation shown in Fig. 14.

8. Conclusions and future work

The paper contributes to the field by proposing a formalization of the front concept and a new object-aware algorithm for the interactive inspection of huge models which uses a rendering budget and supports selection of individual objects and sets of objects, displacement of the selected objects and real-time collision detection during these displacements.

As far as we know, no present algorithm addresses rendering such huge sets of objects with guaranteed frame rates, while allowing for the modification of individual objects during the inspection and attempting to optimize image quality. Our proposal addresses these needs, and is also able to verify on the fly possible collisions when moving objects around. The proposed algorithm has proved to successfully solve a precise need in the design of huge assemblies and is now being used in real ship design environments.

In our tests, we have not perceived differences between the quality of the images rendered by our algorithm with respect to the ground truth represented by the input models. Isolated frames may present transient artifacts when the visibility changes abruptly, since the updates exceed the budget, but they very quickly disappear.

Although we have not discussed it here, our current implementation supports textured polygons and selection through attributes. To be able to fully support textured models, however, we need to further improve the ORIs to handle them. This is intended as future work.

Because of the locality in space afforded by the Kd-trees, the whole pre-processing is amenable to being done out-of-core, and hence our approach should scale well for even larger models and modest hardware. Another avenue of improvement is therefore the implementation of this out-of-core approach in a fully automatic way.

Acknowledgments

The source 3D datasets were provided by and are used with permission of the Boeing Company and *Sener Ingeniería y Sistemas*; Sener also provided the initial industrial problem from which this work stems. The authors would like to thank for the partial support by grant TIN2014-52211-C2-1-R of the *Ministerio de Economía y Competitividad* with funds from FEDER, from the European Community and grant CENIT–BAIP2020 of the Spanish government.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at http://dx.doi.org/10.1016/j.cad.2016.06.005.

References

[1] Yoon S-E, Gobbetti E, Kasik DJ, Manocha D. Real-time massive model rendering. Synthesis lectures on computer graphics and animation. Morgan & Claypool Publishers; 2008.
[2] Pajarola R, Gobbetti E. Survey of semi-regular multiresolution models for interactive terrain rendering. Vis. Comp 2007;23(8):583–605.
[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 2005;24(3):878–85.
[4] Andújar C, Brunet P, Chica A, Navazo I. Visualization of large-scale urban models through multi-level relief impostors. Comput Graph Forum 2010; 29(8):2456–68.
[5] Gobbetti E, Iglesias Guitián JA, Marton F. Covra: A compression-domain output-sensitive volume rendering architecture based on a sparse representation of voxel blocks. Comput Graph Forum 2012;31:1315–24. 3pt4.
[6] Yoon S-E, Salomon B, Gayle R, Manocha D. Quick-VDR: Out-of-core view-dependent rendering of gigantic models. IEEE Trans Vis Comput Graph 2005; 11(4):369–82.
[7] Samet H. Foundations of multidimensional and metric data structures. Morgan Kaufmann; 2006.
[8] Funkhouser TA, Sequin CH. Adaptive display algorithm for interactive frame rates during visualisation of complex virtual environments. In: Proceedings of SIGGRAPH'93, 1993, pp. 247–254.
[9] Gobbetti E, Bouvier E. Time-critical multiresolution rendering of large complex models. Comput -Aided Des 2000;32(13):785–803.
[10] eui Yoon S, Lindstrom P, Pascucci V, Manocha D. Cache-oblivious mesh layouts. ACM Trans Graph 2005;24:886–93.
[11] Cignoni P, Ganovelli F, Gobbetti E, Marton F, Ponchio F, Scopigno R. Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. ACM Trans Graph 2004;23:796–803.
[12] Gobbetti E, Marton F. Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. Comput Graph 2004;28(6):815–26.
[13] Crassin C, Neyret F, Lefebvre S, Eisemann E. Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In: Haines E, McGuire M, Aliaga DG, Oliveira MM, Spencer SN, editors. Proceedings of the 2009 symposium on interactive 3D graphics, SI3D 2009, February 27–March 1, 2009, Boston, Massachusetts, USA. ACM; 2009. p. 15–22.
[14] Andújar C, Boo J, Brunet P, González MF, Navazo I, Vázquez P-P, Vinacua A. Omni-directional relief impostors. Comput Graph Forum 2007;26(3):553–60.
[15] Baxter III WV, Sud A, Govindaraju NK, Manocha D. Gigawalk: Interactive walkthrough of complex environments. In: Proceedings of the 13th eurographics workshop on rendering. EGRW'02, Aire-la-Ville, Switzerland, Switzerland: Eurographics Association; 2002. p. 203–14.
[16] Borgeat L, Godin G, Blais F, Lahanier C. Gold: interactive display of huge colored and textured models. ACM Trans Graph 2005;24:869–77.
[17] Peng C, Cao Y. A gpu-based approach for massive model rendering with frame-to-frame coherence. Comput Graph Forum 2012;31:393–402. 2pt2.
[18] Willmott A. Rapid simplification of multi-attribute meshes. In: Proceedings of the ACM SIGGRAPH symposium on high performance graphics. HPG'11, New York, NY, USA: ACM; 2011. p. 151–8.
[19] Rossignac J, Borrel P. Multi-resolution 3D approximations for rendering complex scenes. In: Falcidieno B, Kunii T, editors. Modeling in computer graphics: methods and applications. Berlin: Springer-Verlag; 1993. p. 455–65. Proc. of Conf., Genoa, Italy, June 1993. (Also available as IBM Research Report RC 17697, Feb. 1992, Yorktown Heights, NY 10598).
[20] Yoon S-E, Salomon B, Lin M, Manocha D. Fast collision detection between massive models using dynamic simplification. In: Scopigno R, Zorin D, editors. Symposium on geometry processing. The Eurographics Association; 2004.
[21] Bittner J, Wimmer M, Piringer H, Purgathofer W. Coherent hierarchical culling: Hardware occlusion queries made useful. Comput Graph Forum 2004;23(3): 615–24.